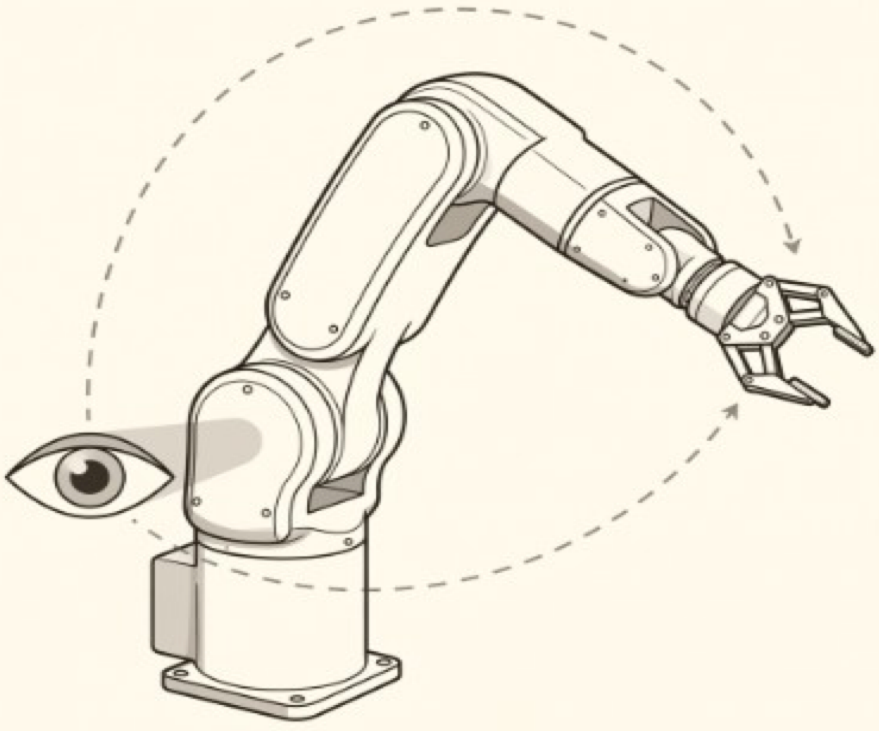
OPEN-ACCESS TEXTBOOK

Action Models for Robot Learning

Pavan Kumar Kandapagari



vision · language · action

Contents

Chapter 1 — The robot learning problem **6**
1.1 Why "action" is the hard part of robotics . . . 6
1.2 Anatomy of an action model: inputs, outputs, training signal . . . 8
1.3 A short history, from STRIPS to π0 . . . 11
1.4 The four families of action models . . . 14
1.5 What you will and will not find in this book . . . 17
1.6 Summary . . . 19
1.x Hands-on exercise + chapter references . . . 21

Chapter 2 — Your first VLA, end-to-end **25**
2.1 What we are going to build, and what is hidden inside . . . 25
2.2 Setting up the environment (OpenVLA weights, LIBERO simulator) . . . 27
2.3 Walking through the inference loop one line at a time . . . 31
2.4 When it works and when it does not . . . 34
2.5 What is left for the rest of the book . . . 36
2.6 Summary . . . 38
2.x Hands-on exercise + chapter references . . . 40

Chapter 3 — Math and ML prerequisites in 30 minutes **44**
3.1 Vectors, matrices, gradients, and why the chain rule rules robotics . . . 44
3.2 Random variables, expectations, KL divergence . . . 46
3.3 A 50-line PyTorch training loop, annotated . . . 49
3.4 Three loss families: supervised, RL, self-supervised . . . 53
3.5 Debugging a model that will not train . . . 57
3.6 Summary . . . 60
3.x Hands-on exercise + chapter references . . . 62

Chapter 4 — Classical action models: planning and inverse dynamics **66**
4.1 Symbolic actions: STRIPS, PDDL, and action schemas . . . 66
4.2 Geometric actions: inverse kinematics and motion planning . . . 69
4.3 Inverse dynamics and computed-torque control . . . 71
4.4 Where classical methods are still load-bearing in modern robots . . . 74
4.5 Summary . . . 76
4.x Hands-on exercise + chapter references . . . 79

Chapter 5 — Learning from rewards: MDPs and reinforcement learning **83**
5.1 States, actions, rewards, and policies . . . 83
5.2 Value iteration and policy iteration . . . 85
5.3 Q-learning and the role of exploration . . . 89
5.4 Why reward design is the hardest part . . . 92
5.5 The MDP-to-robot translation problem . . . 94
5.6 Summary . . . 97

5.x Hands-on exercise + chapter references . . . 99

Chapter 6 — Learning from demonstrations: behavior cloning and imitation learning 104
Why imitation is the dominant signal in modern robotics . . . 104
Behavior cloning, step by step . . . 106
Compounding error and DAgger . . . 109
A glance at IRL and adversarial imitation . . . 112
Choosing between BC, IRL, and RL . . . 114
6.6 Summary . . . 116
6.x Hands-on exercise + chapter references . . . 119

Chapter 7 — Deep RL for control: DQN to SAC and PPO 123
7.1 Function approximation: from Q-tables to Q-networks . . . 123
7.2 Policy gradients and the variance problem . . . 125
7.3 PPO in 100 lines . . . 128
7.4 Off-policy actor-critic: DDPG, TD3, SAC . . . 130
7.5 Sim-to-real: domain randomization in one slide . . . 133
7.6 Summary . . . 135
7.x Hands-on exercise + chapter references . . . 138

Chapter 8 — Sequence models meet control 142
8.1 The transformer in two pages, for control . . . 142
8.2 Decision Transformer: control as conditional sequence modeling . . . 144
8.3 Trajectory Transformer and beam-search planning . . . 147
8.4 What gets tokenized: states, actions, returns, language . . . 149
8.5 Bridge to foundation action models, and to the SSM alternative (RoboMamba) . . . 151
8.6 Summary . . . 153
8.x Hands-on exercise + chapter references . . . 155

Chapter 9 — World models and model-based learning 159
9.1 What is a world model, really . . . 159
9.2 Latent dynamics: RSSM and Dreamer . . . 161
9.3 Planning in latent space . . . 163
9.4 Video-prediction world models (Genie, V-JEPA) . . . 166
9.5 World models vs. VLAs: the architecture debate . . . 168
9.6 Summary . . . 170
9.x Hands-on exercise + chapter references . . . 173

Chapter 10 — Diffusion and flow models for action generation 177
10.1 A 10-minute introduction to diffusion models . . . 177
10.2 Diffusion Policy and ACT . . . 179
10.3 Flow matching and rectified flow for action . . . 182
10.4 Trade-offs: latency, multimodality, smoothness . . . 184
10.5 Action-head choices in modern VLAs . . . 186
10.6 Summary . . . 189
10.x Hands-on exercise + chapter references . . . 191

Chapter 11 — The VLA recipe: from CLIP to RT-1 195
11.1 CLIP and the multimodal pretraining moment . . . 195
11.2 Language-conditioned imitation: BC-Z, RT-1 . . . 197
11.3 Action tokenization: a small idea with large consequences . . . 199
11.4 What RT-1 changed and what it did not . . . 201
11.5 The data side: when does scale start to pay off . . . 203
11.6 Summary . . . 205
11.x Hands-on exercise + chapter references . . . 208

Chapter 12 — Scaling up: RT-2, OpenVLA, and Octo 212
12.1 RT-2: a VLM that also outputs actions . . . 212
12.2 OpenVLA: an open-source 7B-parameter VLA . . . 214
12.3 Octo: a generalist policy with a diffusion head . . . 216
12.4 Open X-Embodiment: the dataset that made all of this possible . . . 218
12.5 What "emergent" really means in this context . . . 221
12.6 Summary . . . 223
12.x Hands-on exercise + chapter references . . . 225

Chapter 13 — Smooth control: π0 and flow-matching action heads 229
13.1 The trouble with discrete action tokens . . . 229
13.2 π0's architecture, end to end . . . 231
13.3 Flow matching as a control objective . . . 233
13.4 What π0 can do that earlier VLAs cannot — and the π0.5 ⟶ π0.6 ⟶ π0.7 lineage . . . 235
13.5 Open questions in continuous-action foundation models . . . 238
13.6 Summary . . . 240
13.x Hands-on exercise + chapter references . . . 242

Chapter 14 — Dual-system architectures: Helix and GR00T N1 247
14.1 Why a single forward pass is not always enough . . . 247
14.2 Helix: a high-level VLM and a low-level sensorimotor model . . . 249
14.3 GR00T N1 and its N1.5–N1.7 successors: humanoid-flavored dual systems . . . 251
14.4 Latency budgets and real-time control . . . 253
14.5 Deployment case studies . . . 255
14.6 Gemini Robotics 2: unified whole-body loco-manipulation . . . 257
14.7 Summary . . . 259
14.x Hands-on exercise + chapter references . . . 262

Chapter 15 — Datasets, benchmarks, and evaluation 267
15.1 What a robot dataset looks like, by example . . . 267
15.2 Open X-Embodiment in detail — and what has (and has not) succeeded it . . . 270
15.3 LeRobot: Hugging Face's dataset format, hub, and community-contributed robot data . . . 272
15.4 Sim benchmarks (LIBERO, CALVIN, RoboCasa, SimplerEnv) and red-team-style suites such as RoboArena . . . 274
15.5 Real-robot evaluation: variance, success rate, time-to-completion . . . 276
15.6 Building your own evaluation . . . 278
15.7 Summary . . . 281
15.x Hands-on exercise + chapter references . . . 283

Chapter 16 — Fine-tuning a VLA for your robot 288
16.1 Picking a base model: OpenVLA, Octo, or a smaller distilled checkpoint . . . 288
16.2 Building a teleop dataset that does not waste your time . . . 290
16.3 LoRA vs. full fine-tuning vs. action-head-only . . . 291
16.4 Sim-to-real fine-tuning loops . . . 293
16.5 A recipe card for new embodiments . . . 295
16.6 Summary . . . 297
16.x Hands-on exercise + chapter references . . . 298

Chapter 17 — Evaluation, safety, and deployment 302
17.1 Safety as a layer, not a property . . . 302
17.2 Runtime monitors and shielding . . . 304
17.3 A/B evaluation on hardware . . . 306
17.4 Logging, alerting, and rollback . . . 307
17.5 What we still cannot certify . . . 309
17.6 Summary . . . 311

17.x Hands-on exercise + chapter references . . . 313

Chapter 18 — Open problems and what comes next **317**
18.1 Generalization across embodiments . . . 317
18.2 Long-horizon and dexterous tasks . . . 319
18.3 Video-pretrained action models . . . 321
18.4 Reasoning + action: where LLM-style chains of thought meet control . . . 323
18.5 What to read next, and how to contribute . . . 325
18.6 Summary . . . 327
18.x Hands-on exercise + chapter references . . . 329

Appendix A — Linear algebra refresher **332**
Appendix A. Linear algebra refresher . . . 332

Appendix B — Probability and information theory **336**
Appendix B. Probability and information theory . . . 336

Appendix C — PyTorch and JAX primer **340**
Appendix C. PyTorch and JAX primer . . . 340

Appendix D — Setting up a robotics simulator **345**
Appendix D. Setting up a robotics simulator . . . 345

Appendix E — Canonical references (full bibliography, by chapter) **350**
Appendix E. Canonical references and the offline source mirror . . . 350

Appendix F — VLA model zoo **355**
Appendix F. Model zoo . . . 355

Chapter 1 — The robot learning problem

1.1 Why "action" is the hard part of robotics

Start with an admission: action is the part of robotics that has resisted us the longest. The bottleneck of the field has moved several times over the last forty years, and each time it moved, the people working on the previous bottleneck declared the problem solved. In the 1980s the bottleneck was perception. Cameras were noisy, compute was scarce, and getting a robot to reliably segment a coffee cup from a table counted as a research result on its own. Deep learning took care of that in the 2010s, and the bottleneck shifted to language understanding and reasoning, because a robot still couldn't be told, in plain English, to "make me coffee" — the gap between an instruction and a sequence of joint commands stayed unbridged. Language understanding fell in turn, to a different generation of large neural networks, and the bottleneck settled back onto what it had probably always been: turning intent into motion. Call it the action problem.

This chapter explains why that problem is hard, why it resisted progress that perception and language did not, and why we finally have something worth writing a textbook about. The rest of Section 1.1 lays out four properties of the action problem that make it structurally different from what came before. Section 1.2 anatomizes a modern action model and gives you the vocabulary the rest of the book uses. Sections 1.3 and 1.4 trace the field's history and the four families of action models we'll study.

A concrete scene

Picture a robot arm with a parallel-jaw gripper in front of a tabletop holding a coffee cup, a sponge, and a small puddle of spilled milk. You tell the robot, in English, "clean up the spill." What does it actually have to do? Roughly: locate the spill, recognize the sponge as the relevant tool, estimate where to grasp it so the wrist clears the cup, plan a path from the gripper's current pose to that grasp pose, close the fingers with the right force, lift, traverse to the puddle, lower the sponge until it contacts the table at the right pressure, drag it across the puddle at the right normal force and surface velocity, lift again, drop the sponge somewhere reasonable, and stop. That's twenty-odd decisions. Half of them are analog. Several are contact-rich. Every one depends on the one before it.

A modern vision-language model can look at this scene and describe exactly what should happen. It can write you a paragraph of plain English laying out the plan. What it can't do, on its own, is *do it.* The gap between the paragraph and the motion is the territory of this book.

Four reasons action is structurally hard

Why is that gap so wide? Four properties of the action problem set it apart from its perceptual and linguistic cousins.

1. The output space is high-dimensional, continuous, and constrained

A modern image classifier picks one of a thousand labels. A language model picks one of fifty thousand tokens. A robot policy picks a vector of, typically, six to thirty real-valued numbers, and it picks a new one every twenty milliseconds.

The numerical structure matters here. The classifier's output space is a probability simplex over a finite set, so you train it by minimizing cross-entropy and evaluate it by counting hits. The robot's output space is a six- or

thirty-dimensional Euclidean manifold (or its tangent, if you parameterize in velocities or torques), constrained by kinematics, joint limits, torque limits, and contact dynamics. Two policies that look numerically close under any standard distance metric can produce wildly different physical behaviors. A millimeter of vertical error matters when you're picking up a coin and doesn't when you're stacking boxes. What counts as "close" is task-dependent in a way that "close" between two images simply isn't.

Geometry complicates things further. The space of rigid-body rotations is the 3-sphere SO(3), not Euclidean 3-space, and learning a policy that respects this without being told to sits at the edge of what current methods can do. Most action models cheat: they parameterize rotations as axis-angle deltas in some local frame, accept the singularities, and hope the training data avoids them. That works until it doesn't, and "until it doesn't" is exactly the distribution-edge case where students get confused and engineers get paged.

2. The training signal is sparse, expensive, and biased

The internet has been a free training corpus for perception and language for the last decade. Tens of billions of labeled images exist, and even more text. Robot actions have no equivalent. A single robot demonstration, one trajectory lasting a few seconds, typically costs a human ten to thirty seconds of teleoperation, plus the capital cost of the robot itself. Until very recently, all the robot data in the world fit on a small server.

The Open X-Embodiment dataset, a 21-institution collaboration released in 2024, collected over one million episodes across 22 different embodiments, and that counts, in 2026, as a watershed moment in the field. ImageNet, by comparison, had fourteen million labeled examples back in 2009 and was considered a minimum viable resource at the time. Robot data is three orders of magnitude smaller than vision data was when modern computer vision got going, and that scaling gap is unlikely to close soon, because acquiring robot data requires physical hardware, not a web crawler.

What data does exist is biased in ways perceptual data isn't. Demos come from a particular set of robots, in a particular set of labs, performing a particular set of tasks under particular lighting, run by particular teleoperators with particular ergonomic habits. The trajectories tend to run too smooth, too uniformly successful, too short. Behavior cloning on data like that produces policies that work beautifully inside the narrow regime of the training distribution and fail in interesting ways outside it. So much of the modern VLA literature is, structurally, dataset engineering. Chapter 15 spends a long time on what robot data actually looks like under the hood.

3. The environment is not a function of your action — it's a process

A perception model maps an input to an output. So does a translation model, and so does, in a deliberately simplified view, a chat model. At core, the relationship is a function.

A robot policy is part of a control loop instead. It produces an action, the environment responds, and the new environment state determines what the next action should be. This control-theoretic structure changes every part of the machine-learning pipeline. Loss functions need to account for trajectories, not single predictions. Evaluation needs to account for compounding error, since a tiny mistake at step one becomes a large mistake at step one hundred. The training data is, in spirit, meant to be closed-loop (you want the policy to learn what to do after it's just done something slightly wrong), but almost all of it gets collected open-loop, because the demonstrator was doing things correctly the whole time. Chapter 6 returns to this at length when we cover compounding error and DAgger. For now it's enough to know that this closed-loop/open-loop mismatch is the single biggest reason imitation learning is harder than supervised learning, even when the data looks similar on paper.

There's a second consequence of the closed-loop structure: a robot policy has to run in real time. A language model with forty seconds of per-token latency is a research curiosity. A robot policy with forty seconds of per-action latency is a non-functional product. Inference compute budgets for robot policies get measured in tens of milliseconds, not seconds, and that constraint reaches back into architecture decisions the rest of deep learning never has to think about. The dual-system architectures covered in Chapter 14, Helix and GR00T N1 among them, exist specifically because a single 7B-parameter forward pass runs too slow for the inner control loop of a humanoid robot.

4. The cost of being wrong is asymmetric

A misclassified image causes embarrassment. A wrong word in a chat response causes confusion. A wrong robot action, at the wrong instant, with the wrong magnitude, breaks the cup, breaks the robot, or breaks the person standing nearby.

That asymmetry carries real technical and methodological weight. Technically, it means any deployed robot policy needs a safety layer that isn't trained from data and that has well-understood worst-case behavior; Chapter 17 is devoted to this. Methodologically, it means the field's success metric can't just be average performance, it has to include worst-case behavior across edge cases. A 95%-success-rate policy that fails by dropping objects on the floor is a productivity tool. A 95%-success-rate policy that fails by closing the gripper on a human hand is a recall notice. Their summary statistics might look identical.

This is the one property, of the four, where machine learning progress has helped the least. The other three (high-dimensional output, sparse data, closed-loop dynamics) are problems the deep-learning toolkit has been chipping away at for a decade. Asymmetric cost is more or less unchanged since 1985, and the modern toolkit barely touches it. When you read a paper claiming a VLA "generalizes" to a new task, the first question worth asking is what the failure mode looked like for the cases that didn't work.

So why is *now* the moment?

Given those four properties, it's fair to ask why a textbook on action models is worth writing in 2026 at all. The answer comes in three pieces.

First, foundation models have closed the language-understanding part of the problem at industrial scale. The "make me coffee" instruction is now a tractable input rather than a research result, because the same vision-language model powering your favorite chat assistant can be reused as the front end of a robot policy. Grounding that instruction in a specific scene is no longer gatekept by how hard the grounding step used to be.

Second, the data wall isn't falling, but it's getting climbable. Open X-Embodiment is the watershed moment; the next generation of multi-institution collaborative datasets, steadily improving teleoperation interfaces, and increasingly photorealistic synthetic data from simulators all push in the same direction. The Stanford AI Index Report 2025 noted that peer-reviewed papers on foundation-model-based robot control grew over 60% between 2022 and 2024, outpacing every other applied-AI subfield. That growth was, structurally, made possible by the dataset side of the ecosystem finally catching up.

Third, the methods that bridge perception, language, and action, the methods this book is about, have crystallized into a recognizable recipe: tokenize the action space, pretrain on a vision-language corpus, co-train or fine-tune on robot demonstrations, then decode. The recipe has variants — discrete action tokens in RT-2 and OpenVLA, continuous flow-matching heads in $\pi 0$, dual-system decompositions in Helix and GR00T N1 — but it is a recipe, and a textbook can teach a recipe.

What a textbook can't do, and here's the honest version of the answer, is make any of the four structural problems above go away. They're still hard. They still bite. A reader who finishes this book won't have made action easy; they'll have learned the vocabulary, the techniques, and the failure modes of the current generation of approaches, and they'll be positioned to push the field forward exactly where the methods don't yet work. That's what we mean by *robot learning*, and the next section gets specific about what its central object, the action model itself, actually is.

1.2 Anatomy of an action model: inputs, outputs, training signal

Section 1.1 argued that turning intent into motion is structurally hard and named four reasons why. This section gets concrete. We'll take a modern action model apart and label its components in language the rest of the book will keep using. By the end, you should be able to look at any paper in the field, RT-1, OpenVLA, $\pi 0$, Helix, GR00T N1, whatever gets published the week after you read this, and identify the three slots that define it: what goes in, what comes out, and what training signal told it the difference between right and wrong.

Three slots. They look simple, and most of the design space of the field lives in the choices made at each one.

Slot 1 — Inputs: what the policy is allowed to look at

An action model takes some observation of the world and produces something the robot can do with it. The first design choice is what counts as "observation."

The minimum is one RGB camera and a clock. Most modern systems use more. A typical input bundle for a manipulation VLA looks like this:

- One or more RGB images, usually from cameras mounted on the robot: a wrist-mounted camera that sees the gripper, and a stationary "agentview" camera that sees the scene. Image resolution is typically 224×224 or 256×256 after preprocessing, small enough for a pretrained vision encoder to process in milliseconds, large enough that table-scale geometry stays visible.
- A natural-language instruction, usually a short imperative sentence such as "pick up the red block," "open the drawer," or "stack the cups." The instruction is optional in classical control and central in foundation-model-based approaches.
- Proprioception: the robot's own measurements of joint angles, joint velocities, end-effector pose, and gripper width. This data is cheap and always available. Modern VLAs use less of it than you'd expect, though, because they're designed for cross-embodiment generalization, and proprioception schemas differ from robot to robot.
- A short history of recent observations and actions. Some policies condition on the last image, some on the last k images, some on a sliding window across the full observation-action trajectory. Long context is more expressive; short context is faster. The choice interacts with the closed-loop/open-loop problem we'll return to in Chapter 6.

A few things about this slot recur throughout the book. The input modality mix isn't fixed by the problem, it's a design choice: the same task can be done by a single-camera policy or a five-camera one, and comparing them isn't trivial, since more cameras means more compute and more surface area for the model to overfit. The inputs are also heterogeneous — pixels, tokens, floating-point joint angles — and combining them is its own architectural question, one we treat in Chapter 8 on tokenization and Chapter 11 on the CLIP-to-RT-1 lineage. And what you decide *not* to look at matters just as much as what you do. A policy with access to a force-torque sensor will learn to use it. A policy without one will quietly substitute visual approximations of contact instead. Both can work. They fail differently.

Slot 2 — Outputs: what the policy is allowed to do

The second design choice is the action space, the set of things the policy is allowed to output. Few decisions carry more downstream consequences, because this choice determines what kind of motion the policy can express and how it gets trained.

Action spaces come in two cuts: by *type* and by *frame*. The type of an action is what the number physically means. The frame is what coordinate system the number is expressed in.

Four common types, ordered roughly from lowest- to highest-level:

- **Torques**. Direct motor commands, one number per joint, updated at the control loop rate (typically 200 Hz to 1 kHz for an arm). This is the most expressive action space; anything the robot can physically do, you can in principle express as a torque sequence. It's also the hardest to learn. Almost no VLAs operate at this level, though classical computed-torque controllers and some specialized legged-locomotion RL policies do.
- **Joint or end-effector velocities**. "Move joint 3 at 0.2 rad/s," or "move the end-effector at 5 cm/s in the +x direction." One layer above torques, with a low-level controller smoothing the velocity command into actual motor commands. Most teleoperation interfaces use this space, and many imitation-learning policies match it.
- **Pose deltas**. "Move the end-effector 1 cm in +x, 2 mm in -z, rotate the wrist by 0.05 rad, close the gripper." Modern VLAs almost universally emit this kind of action: a small relative motion per timestep, applied by an inverse-kinematics controller that works out the joint commands. OpenVLA's seven-dimensional action (three translation, three rotation, one gripper) is a canonical example.
- **Waypoints or trajectories**. "Move to this pose, then this pose, then this pose." A single inference produces a sequence of future poses rather than one delta. Diffusion policies (Chapter 10) and the continuous-action heads in $\pi 0$ and Octo (Chapters 12 and 13) work this way.

The frame distinction cuts across all four types. Most action spaces get expressed relative to the current end-effector pose: "go 1 cm forward from where you are" rather than "go to absolute world coordinate (0.45, 0.10, 0.30)." Relative actions generalize better, since they don't depend on how the robot's base frame happens to be calibrated, and they're easier to compose. Nearly every contemporary VLA uses relative actions. The exceptions are systems with strong global scene grounding, some 3D-aware VLAs like LEO in Chapter 15, and end-to-end driving policies like OpenDriveVLA, where the relevant frame is the world rather than the agent.

A second axis cuts across the type: discrete versus continuous representation. Even within "pose deltas," a model can output continuous floating-point numbers, the natural representation, or discretize each dimension into a fixed number of bins and predict bin indices instead. RT-1, RT-2, and OpenVLA all chose discretization (256 bins per axis is standard), because it lets them reuse a language-model decoder head and a cross-entropy loss. $\pi 0$ and Octo went the other way, with continuous outputs from a diffusion or flow-matching head. We'll spend Chapter 10 on that trade-off. For now, what matters is that the same physical action, "move 1 cm in +x," gets represented and trained differently depending on which side of this choice the architecture landed on.

Slot 3 — Training signal: how the model learns to fill the gap

The third slot is what tells the model that one mapping from inputs to outputs beats another. This is where action models split into the four families we'll name in Section 1.4, and it's the slot where methods have changed the most over the last fifty years.

Three training-signal types dominate the field today, and they aren't mutually exclusive. Most contemporary systems combine at least two.

- **Supervised imitation from demonstrations**. The signal is a dataset of (observation, action) pairs collected by a human teleoperator. The loss is whatever you'd use for a regular sequence-prediction problem: cross-entropy if the action space is discrete, mean-squared error or a diffusion-style loss if it's continuous. This is the dominant signal in modern VLAs, partly because the data is easiest to collect at scale and partly because it works well enough, even though it inherits the closed-loop/open-loop mismatch flagged in §1.1.
- **Reinforcement learning from rewards**. The signal is a scalar reward function, hand-designed or learned, that says this trajectory accomplished the task and that one didn't. The model learns to maximize expected return. RL fits naturally where a demonstration is hard to produce but a success criterion is easy to write down, locomotion being the clearest case. It also fits fine-tuning a policy already trained by imitation: you start with something that mostly works, then use RL to polish the parts that don't. Chapter 5 develops this in detail.
- **Self-supervised pretraining from large unlabeled corpora**. The signal is a pretext task, predicting masked image patches, predicting the next text token, contrastively aligning images and text, that needs no robot data at all. The model learns useful representations from internet-scale vision and language data, then gets adapted to robots through a smaller second stage. CLIP (Chapter 11) is the canonical example, and every modern VLA inherits most of its capacity from this stage.

The modern recipe, almost without exception, layers all three: self-supervised pretraining on internet data, supervised imitation on robot demonstrations, and optionally reinforcement-learning fine-tuning to close the last gap. When you read a paper, the useful question isn't "which signal does it use?" It's "in what proportions, and in what order?"

A worked instance: OpenVLA in three slots

The anatomy sticks better once it's pinned to a real model. Take OpenVLA (Kim et al. 2024, arXiv:2406.09246), the open-source VLA you ran in Chapter 2.

- **Inputs**. One RGB image, resized to 224×224, encoded by a combination of SigLIP (a vision-language pretraining objective) and DINOv2 (a self-supervised visual representation). A natural-language instruction, tokenized by the Llama-2 tokenizer. No proprioception, no history beyond the current frame.
- **Outputs**. Seven discrete action tokens, one per axis of an end-effector pose delta: three translation, three rotation (axis-angle), one gripper. Each axis is binned into 256 buckets fitted to the per-axis range of the training data. The tokens come out of the same decoder that would otherwise produce text, repurposed from the bottom 256 entries of Llama's vocabulary.

- **Training signal.** Two stages, both supervised. Stage one: the Llama-2 + SigLIP + DINOv2 backbone arrives pretrained on internet vision-language data, self-supervised, no robot data involved. Stage two: imitation on 970,000 robot trajectories from the Open X-Embodiment dataset, with cross-entropy loss over the discrete action tokens.

Three slots, four design choices each, one paragraph per model. Run this exercise on anything in the Model Zoo (Appendix F) and the shape of the design space falls right out.

Where this differs from a perception model and from a planner

Two contrasts close out this section, because the entire premise of action models, and of this book, is that they're a distinct object from the two things they get confused with most often.

A perception model (an image classifier, an object detector, a vision-language model) has Slot 1 and Slot 3 but no real Slot 2. Its outputs are labels, segments, or natural-language responses, not commands a robot will execute. A perception model can be a component of an action model, and every modern VLA has one embedded in it, but the embedding isn't free. The perception model has to get wired into a head that emits actions, and a training signal that grounds those actions has to be added on top. Most of the engineering effort in OpenVLA, RT-2, and $\pi 0$ lives in exactly that wiring.

A planner, think STRIPS, PDDL, motion planners like RRT or PRM, has Slot 2 but typically lacks Slot 1 and Slot 3 in the sense we've been using. It takes a symbolic or geometric description of the world rather than raw sensor input, produces an action or trajectory, and doesn't learn from data at all, since the rules are written by hand. Classical planners are extremely good at certain things action models struggle with, and Chapter 4 makes that case in detail. They're bad at certain things action models handle well, which is why the two coexist in modern robotic stacks instead of one replacing the other.

The action models this book is about sit in the middle. They accept raw high-dimensional sensor input like a perception model, produce executable actions like a planner, and learn the mapping from data rather than having it written down. That combination is what makes them new, and what makes them hard. Section 1.3 traces how the field arrived at that combination, and Section 1.4 names the four families that share the slot structure but differ in how they fill it.

1.3 A short history, from STRIPS to $\pi 0$

Writing the history chapter of a textbook on a hot new field tempts you to treat the past as a series of failed attempts, now superseded. Resist that. The history of action models isn't a history of failures. It's a history of the bottleneck moving, through perception, planning, language, control, data, deployment, and at each move, the methods of the previous era kept solving the problems they'd always been good at. Almost every modern VLA still relies on a classical inverse-kinematics solver somewhere in the stack. Almost every legged-locomotion controller still relies on a hand-tuned reinforcement-learning reward. The eras compound. They don't replace each other.

What follows is a selective tour through six eras, meant to give you the names and the conceptual shifts. It isn't a complete bibliography (the Model Zoo in Appendix F and the chapter-by-chapter references in Appendix E.2 do that job). Read this section as a map you can hold in one hand, not a survey.

Era 1 — Symbolic planning (1970s)

The original action models were symbolic. STRIPS, Fikes and Nilsson's 1971 system, was developed at SRI's Shakey project, the first mobile robot built to navigate a physical lab and execute tasks described in natural language. The representation was logical: the world was a set of predicates ("the door is open," "the robot is in room B"), actions were operators with preconditions and effects, and a plan was a sequence of operators transforming an initial state into a goal state.

For a 2026 textbook on learned action models, STRIPS looks ancient, and in many ways it's the opposite of what fills the rest of this book. So why include it? Because the abstraction it introduced, that a robot's job is to search over a sequence of operators that transform world states, is still the right one. PDDL (McDermott et al., 1998),

STRIPS's modern descendant, is alive and well in benchmarks like the International Planning Competition. Modern hierarchical-RL systems and many task-and-motion-planning frameworks still use it to represent the task layer even when the motion layer is fully learned. Chapter 4 argues that the real dividing line between Era 1 and Eras 2 through 6 isn't "symbolic versus learned." It's "manipulating a discrete set of operators versus emitting continuous low-level commands," and modern stacks do both, at different levels of the same system.

Era 2 — Geometric and dynamic robotics (1980s–1990s)

The 1980s and 1990s were the era of control: kinematics, dynamics, and motion planning treated as branches of applied mathematics. The canonical references, Spong, Hutchinson, and Vidyasagar's *Robot Modeling and Control*, Murray, Li, and Sastry's *A Mathematical Introduction to Robotic Manipulation*, LaValle's *Planning Algorithms*, are all still in print, and the methods they describe are still load-bearing in modern robots. Forward and inverse kinematics, computed-torque control, RRTs and PRMs for collision-free path planning, impedance and force control: none of this is a legacy technique. It's the substrate learned policies operate on top of.

Era 2's contribution to action models in the modern sense is mostly indirect, since none of these methods learn from data; they're derived from a model of the robot and the task instead. But they established two ideas the rest of the book will use without spelling out again. First, the action space is structured: joint angles, joint velocities, end-effector poses, and torques relate to each other through known geometry, and switching among them just means choosing the right level of abstraction. Second, some problems are easier to write down than to learn. Inverse kinematics for a six-degree-of-freedom arm is a small system of equations. Collision checking in a known environment is a graph search. When a modern VLA produces an end-effector pose delta, an inverse-kinematics solver from Era 2 turns it into joint commands, and an Era-2 collision checker decides whether to execute it at all. Chapter 4 treats this hand-off as the load-bearing point it actually is.

Era 3 — First learned approaches (late 1980s–2000s)

The first credible attempt to learn an action model from data was Pomerleau's ALVINN (1988), a fully connected neural network with one hidden layer that mapped a 30×32 camera image of a road to a steering angle. ALVINN drove a van across the United States in 1995, nearly thirty years before the deep-learning revolution, and it's the direct ancestor of every modern behavior-cloning policy.

The 1990s and 2000s saw parallel growth in reinforcement learning for robotics. The single best reference for this era is the Kober, Bagnell, and Peters survey (IJRR, 2013), which captures everything from policy-search methods on humanoids to motor-primitive learning on industrial arms. The era produced impressive demonstrations: pendulum swing-up, ball-in-cup, robot table tennis. What it didn't produce was generalization. Each system was hand-engineered for its task, the policies were small (often a few hundred parameters), and transfer between tasks stayed a research result rather than a default. The data problem from §1.1, sparse, expensive, biased, was acute, and the methods of the time had no good answer for it.

Era 3's contribution is the existence proof: a robot's action model can be learned. The specific methods that did the learning would later get displaced by deep learning, but the framing, policy as a function approximator, training as optimization against a reward or a demonstration, is exactly the framing the rest of the field inherited.

Era 4 — Deep reinforcement learning (2013–2017)

The arrival of deep learning in robotics is usually dated to DeepMind's DQN paper (Mnih et al., Nature 2015), not because the paper was about robotics but because it showed a convolutional neural network could serve as the function approximator inside a Q-learning agent and learn to play Atari games from raw pixels. The lesson generalized fast. Within two years, Levine, Finn, Darrell, and Abbeel had end-to-end visuomotor policies trained with guided policy search. Within four, the algorithmic toolkit for deep RL on robots, TRPO (Schulman et al. 2015), DDPG (Lillicrap et al. 2016), PPO (Schulman et al. 2017), SAC (Haarnoja et al. 2018), had stabilized into the list of names you still see in every paper today.

Era 4 also discovered, expensively, the limits of deep RL on real robots. RL needs environments to interact with, and interacting with a real robot for millions of episodes just isn't practical. Simulation became the workhorse,

and sim-to-real, the problem of training in simulation and deploying on hardware, turned into a subfield of its own. Domain randomization (Tobin et al., 2017) was the era's signature trick, and the legged-locomotion successes of the late 2010s, ANYmal, Cassie, Spot, all rest on it.

The era's contribution to modern action models runs conceptual rather than direct. Deep RL isn't, in 2026, the dominant training signal for foundation action models; most VLAs train by imitation, with optional RL fine-tuning on top. But the recognition that a deep neural network could be the policy came from this era, and the algorithmic infrastructure it produced, replay buffers, target networks, advantage estimation, is still what you reach for when imitation isn't enough. Chapter 7 develops all of this in depth.

Era 5 — Deep imitation and the sequence-model turn (2017–2021)

While Era 4 was working out deep RL, a quieter parallel track was rediscovering imitation. The argument was pragmatic: if reward design is the hardest part of RL, and demonstrations are cheaper than reward functions for most tasks anyone actually cares about, maybe the right move is to scale up the ALVINN idea with modern architectures. BC-Z (Jang et al., CoRL 2021) was the strong demonstration, a CNN-based behavior-cloning policy that generalized to unseen tasks given language conditioning. The language thread matters here; it's the first point in this lineage where natural-language instructions became inputs to a policy at all. The pattern only accelerated from there.

In parallel, the transformer architecture started showing up in control. Chen et al.'s Decision Transformer (NeurIPS 2021) showed you could cast reinforcement learning as a sequence-prediction problem: feed in (state, action, return-to-go) tuples, ask a transformer to predict the next action. The Trajectory Transformer (Janner, Li, Levine, NeurIPS 2021) did something similar from a planning angle. Neither one was a robot system. Both gestured toward unifying perception, language, and control inside one architecture, and that unification arrived in the next era.

Era 5's contribution is the recognition that the closed-loop trajectory is the natural unit, and that a transformer is the natural architecture for sequences built from mixed-modality tokens. The vocabulary the next era's foundation action models would use, action tokens, return-to-go, language conditioning, got assembled here first.

Era 6 — Foundation action models (2022 → now)

The era we live in starts with RT-1 (Brohan et al., 2022, arXiv:2212.06817), a transformer policy trained on 130,000 demonstrations across 700-plus tasks at Google, using language conditioning and an action-tokenization scheme that would later become canonical. RT-1 was a generalist by 2022 standards; by 2026 standards it reads as a primitive baseline. The trajectory from there moves fast.

RT-2 (Brohan et al., 2023, arXiv:2307.15818) made the conceptual leap: take an off-the-shelf vision-language model, PaLI-X, PaLM-E, and reuse it as the policy backbone, with discrete action tokens slotted into the bottom of the vocabulary. Suddenly a policy inherited the world knowledge of a trillion-token language model trained on internet data. This "VLM-as-policy" moment is what gives the term Vision-Language-Action model its current meaning.

The open-source response arrived in 2024 with OpenVLA (Kim et al., arXiv:2406.09246), a 7-billion-parameter VLA built on Llama-2, SigLIP, and DINOv2, trained on 970,000 trajectories from the Open X-Embodiment dataset. OpenVLA matters not because it's the strongest VLA (it isn't) but because it's the first one a graduate student can actually run, fine-tune, and modify. The rest of this book leans on it as the canonical running example for exactly that reason.

At the same time, a parallel track abandoned discrete action tokens for continuous heads. Octo (UC Berkeley, 2024, arXiv:2405.12213) plugged a diffusion head onto a transformer backbone. $\pi 0$ (Physical Intelligence, 2024, arXiv:2410.24164) plugged a flow-matching head onto a PaliGemma backbone and trained on ten thousand hours of robot data, producing dexterous behavior, laundry-folding, egg-packing, that earlier discrete-token VLAs simply couldn't manage.

The 2025 generation added a third axis: dual-system architectures, where a slow VLM "System 2" handles scene understanding and a fast sensorimotor "System 1" handles real-time control. Figure AI's Helix runs on the Figure 02 humanoid at BMW's Spartanburg plant; NVIDIA's GR00T N1 (arXiv:2503.14734) is the open-research counterpart. Both architectures exist because a single 7B-parameter forward pass runs too slow for the inner control loop of a humanoid robot, the exact latency argument previewed back in §1.1.

And the era isn't finished. As of mid-2026, the model zoo (Appendix F) lists 24 named systems, including efficiency-frontier checkpoints (SmolVLA, TinyVLA, RoboMamba) that run on consumer GPUs, long-horizon variants (LiLo-VLA, Long-VLA, Embodied-R1), 3D-grounded models (LEO), and cross-domain ones (OpenDriveVLA for driving). The recipe itself, pretrain on internet vision-language data, fine-tune on robot demonstrations, decode, has stabilized. The variants are where the field's research energy lives now.

The through-line

Six eras, fifty-five years, one direction of travel. Each era took on a weaker form of supervision than the one before it: Era 1 took rules, Era 2 took a model of the robot, Era 3 took a hand-crafted reward or a small dataset, Era 4 took rewards plus simulation, Era 5 took demonstrations and language, and Era 6 takes whatever it can get, internet-scale vision-language data, cross-embodiment demonstration corpora, a small amount of robot-specific fine-tuning. The trend runs monotone, and nothing suggests it's about to stop.

What stays constant across eras is the three-slot anatomy from §1.2. Every system has inputs, outputs, and a training signal. What changes is which parts of each slot get designed by hand and which get learned. Section 1.4 picks out the four broad families this lineage produced and gives each one a name.

1.4 The four families of action models

The history in §1.3 was chronological. This section is taxonomic. Take the same fifty-five years of work and regroup it into four families of action model, named so you have vocabulary for the chapters ahead: classical/analytical, reinforcement-learning, imitation, and foundation/VLA. They aren't mutually exclusive, and almost every deployed modern robot is a stack containing at least two of them. But the differences matter, because each family fills the three slots from §1.2, inputs, outputs, training signal, in a structurally different way.

A taxonomy that doesn't earn its keep is just an organizing trick. This one earns it for two reasons. The right family is often determined by the data and supervision available for a task, not by the task itself, so once you can name what kind of supervision you actually have, the family follows. And when a modern system fails, and they all do somewhere, the failure mode is usually characteristic of one family inside the stack. Knowing which family is which shortens the debugging considerably.

Family 1 — Classical / analytical action models

The first family treats the action problem as applied mathematics. Given a model of the robot (its kinematics, dynamics, and contact geometry) and a model of the task (a goal pose, a path constraint, a stability criterion), you derive the action sequence rather than learning it. There's no training data, no neural network involved. Just a system of equations and an algorithm that solves it.

Examples in this family are the workhorses of industrial robotics. STRIPS and its descendant PDDL, the Era-1 systems from §1.3, compute symbolic plans: sequences of operators that transform one logical world state into another. Inverse-kinematics solvers, closed-form for some arm geometries, numerical (Jacobian pseudo-inverse, damped least squares) for the rest, turn a desired end-effector pose into joint angles. Motion planners like RRT, RRT*, and PRM (LaValle 2006) compute collision-free paths through configuration space. Computed-torque controllers and operational-space controllers turn a desired trajectory into the motor torques that track it. Each of these is a classical action model, and none of them learns from data.

In §1.2's slot anatomy: Slot 1 (inputs) is a structured, low-dimensional representation of the world, a list of predicates, a goal pose, a known obstacle map. Slot 2 (outputs) is a symbolic plan, a joint trajectory, or a torque command, depending on the controller's level. Slot 3 (training signal) doesn't exist, since the system is derived, not trained.

The argument against this family is exactly what motivated everything in Eras 3 through 6: it doesn't scale to scenes that haven't been hand-described. Classical methods need a clean model of the world, and the world is usually messy. The argument for the family is strong nonetheless. When the model is clean, the methods are correct, fast, and certifiable. Industrial pick-and-place lines that move billions of parts a year run almost entirely on classical methods, because the model is clean (a fixturized part on a known conveyor) and the cost of being wrong is high.

Chapter 4 develops the family in detail and makes the case that the right question for a modern roboticist isn't "classical or learned," it's where the dividing line goes in this particular system.

Family 2 — Reinforcement-learning action models

The second family treats the action problem as optimization against a reward function. You define a reward, a scalar describing how well the robot is doing, and train a policy to maximize the expected sum of future rewards. Training data comes from the robot's own interactions with an environment (simulated, real, or a mixture), and the supervision signal is the reward itself.

This is the family that produced the deep-RL successes of Era 4 (DQN, TRPO, PPO, SAC) and that powers most of the modern legged-locomotion stack. ANYmal, Cassie, and Spot all use RL controllers for their gait policies, trained massively in simulation and deployed via sim-to-real transfer with domain randomization. The locomotion success isn't an accident: walking is the canonical case where a reward function is easy to write (forward velocity, upright posture, energy efficiency) but a demonstration is hard to provide, since you can't puppeteer a quadruped through a stumbling-on-gravel recovery the way you can puppeteer an arm through a pick.

In slot terms: Slot 1 (inputs) is whatever the policy observes from the environment, proprioception, often a depth image, sometimes a heightmap. Slot 2 (outputs) is typically joint targets or end-effector velocities, executed by a low-level controller running at higher rate. Slot 3 (training signal) is the reward, which isn't a target output but a scalar evaluation of one, and the machinery converting that scalar into a gradient (policy gradients, Q-learning, actor-critic) is what most of Chapters 5 through 7 cover.

The cost of belonging to this family is the cost of reward design. Writing a reward function that produces the behavior you actually want, without producing degenerate strategies that game the reward, ranks as the second-hardest problem in robot learning (the data problem still takes first place). Reward hacking is a recurring failure mode: a quadruped that learns to fall forward and slide because forward velocity gets rewarded and standing up doesn't, a manipulator that learns to flick the object off the table because "object is no longer visible" accidentally became part of the success criterion. Chapter 5 spends a full section on this. For now, what matters is that the family's strength, just write down what you want, doubles as its weakness, because what you write down is rarely what you actually want.

Family 3 — Imitation action models

The third family treats the action problem as supervised learning from demonstrations. Instead of a reward, you have a dataset of (observation, action) pairs collected by a human teleoperator, and the loss is the prediction error between the policy's action and the demonstrator's.

This is the largest family in modern manipulation by a wide margin, for the practical reason §1.1 already named: for most tasks anyone cares about, demonstrations are cheaper than reward functions. ALVINN (1988) was the prototype. Behavior Transformer (Shafiullah et al., 2022), Diffusion Policy (Chi et al., 2023), ACT (Zhao et al., 2023), and BC-Z (Jang et al., 2022) are the modern instances. All of them share a Slot-1/Slot-2 structure inherited from supervised learning, observations in, actions out, and a Slot 3 that's simple in spirit (match the demonstrator) but pathological in practice, thanks to the closed-loop/open-loop mismatch from §1.1.

The defining problem of this family is compounding error. A policy trained to match demonstrations sees, during training, only the states the demonstrator visited. At deployment, it sees the states its own imperfect actions led to, which sit slightly off-distribution. That slight drift produces slightly worse actions, which produce more off-distribution states, which produce worse actions still. By the end of an episode the policy is operating in a regime it never saw during training, and the trajectory diverges. DAgger (Ross, Gordon, Bagnell, 2011) is the canonical fix: query the demonstrator on the policy's own visited states and fold those into the dataset. But DAgger is expensive, and modern practice rarely uses it as written. Instead it leans on more demonstrations and better architectures (history conditioning, diffusion heads, action chunking), accepting compounding error as the price of admission to this family.

A clean example of how seriously the field takes this trade-off is Action Chunking with Transformers (ACT, Zhao et al., 2023). Rather than predicting one action at a time, the policy predicts the next k actions in a single forward pass.

Predicting a chunk makes the policy more robust to single-step errors and produces smoother behavior, at the cost of slower reaction to surprises. The choice of k, typically 8 to 16, is one of the architectural levers separating modern imitation systems from naive behavior cloning, and Chapter 6 returns to it.

Family 4 — Foundation / VLA action models

The fourth family is the subject of the second half of this book. A foundation action model, a Vision-Language-Action model in the dominant subgenre, takes the imitation-learning recipe of Family 3 and adds two ingredients: a large-scale pretraining stage on internet vision-language data, and a cross-embodiment training stage on robot data aggregated across many labs and robot types. The result is a policy you can prompt in natural language, one that inherits world knowledge from a text-and-image backbone and is meant to generalize across tasks, scenes, and, with fine-tuning, robot embodiments.

The members of this family are the systems Era 6 produced. RT-1 (Brohan et al., 2022, arXiv:2212.06817) is the canonical first instance, an 8-task transformer trained on 130k demonstrations. RT-2 (Brohan et al., 2023, arXiv:2307.15818) marks the moment a vision-language model got reused as the policy backbone. OpenVLA (Kim et al., 2024, arXiv:2406.09246) is the open-source 7B-parameter VLA built on Llama-2, SigLIP, and DINOv2, trained on 970k Open X-Embodiment trajectories. Octo (arXiv:2405.12213) and $\pi 0$ (Black et al., 2024, arXiv:2410.24164) are the continuous-action-head counterparts. The Sapkota survey (arXiv:2505.04769) and the Pure-VLA survey (Zhang et al., 2025, arXiv:2509.19012) catalogue the rest, and the Model Zoo in Appendix F names twenty-four of them along with parameter counts.

In slot terms: Slot 1 (inputs) is one or more RGB images plus a natural-language instruction, often with proprioception and a short history added. Slot 2 (outputs) is a pose delta or trajectory, represented either as discrete tokens (RT-2, OpenVLA) or as continuous output from a diffusion or flow-matching head (Octo, $\pi 0$). Slot 3 (training signal) comes layered: self-supervised pretraining on internet data, supervised imitation on robot demonstrations, sometimes RL fine-tuning stacked on top. The slot diagram for a VLA looks like an imitation policy with a much larger Slot 3 and a far more heterogeneous Slot 1.

What makes this family genuinely new, and worth a textbook, is the compounding of training signals. A VLA inherits language understanding from a corpus of trillions of tokens, scene understanding from billions of images, and action grounding from roughly a million demonstrations. None of the previous three families can claim that stack. Whether the stack delivers what it promises, true cross-embodiment generalization, robustness to novel objects, long-horizon task execution, is the empirical question driving Chapters 11 through 15.

Reading the four together

A worked exercise closes the section. Consider four ways to build a robot that empties a dishwasher.

A classical solution writes a PDDL plan ("pick plate from rack, place plate in cabinet, repeat") and an inverse-kinematics-plus-RRT motion planner that realizes each step. It needs a clean model of the kitchen.

A reinforcement-learning solution defines a reward, say +1 for each plate reaching the cabinet, −1 for each plate broken, and trains a policy in simulation through millions of episodes before sim-to-realing it to hardware. It needs a simulator that captures the dishwasher's geometry and the plates' contact dynamics.

An imitation solution collects fifty teleoperated demonstrations of an operator emptying the dishwasher and trains a Diffusion Policy on them. It needs the operator's time and a robot to demonstrate on.

A foundation/VLA solution takes OpenVLA, fine-tunes it on those same fifty demonstrations, and prompts it with "empty the dishwasher." It needs the demonstrations and the OpenVLA checkpoint both, but in exchange it inherits enough world knowledge to recognize plates it hasn't seen before, handle a misaligned rack, and respond to the operator's natural-language correction mid-task.

The fourth option is the most flexible and the most expensive at training time. The first is the most reliable when its assumptions hold. The middle two trade off in different directions depending on what's cheap for you to collect. A real deployed system, Figure 02's kitchen demos, GR00T-enabled humanoid pilots, the warehouse arms running $\pi 0$

fine-tunes, usually contains pieces of all four. Section 1.5 sets out which of these the rest of the book covers in depth, and which it doesn't.

1.5 What you will and will not find in this book

A book about a moving field has to draw lines somewhere. The lines drawn here aren't the only reasonable ones, but after a year of teaching this material, they produced the smallest gap between what students wanted to know and what they could actually do by the end of a semester. This section names those lines and, more usefully, explains why they sit where they do. The goal is to save you from reading three other books just to discover that this one doesn't cover what you actually need.

Short version: this book treats action models as a class of model, with the foundation/VLA family as the dominant case. It isn't a robotics textbook in the LaValle sense, a controls textbook in the Khalil sense, or a deep-learning textbook in the Goodfellow sense. It assumes you've already read, or are willing to read alongside it, the chapters of those books you personally need. In exchange, this book goes deep on what those others don't yet cover: how modern action models, diffusion policies, action-tokenized transformers, flow-matching heads, vision-language-action stacks, actually work, how they're trained, and how they fit into a deployed robot.

What the book covers

Four commitments shape the table of contents.

The first is the four families, treated as one. Most existing texts pick a family and stay inside it. Sutton and Barto stay inside RL. LaValle stays inside classical planning. Argall's survey (RAS 57(5), 2009) stays inside imitation. The VLA surveys (Sapkota, arXiv:2505.04769; Zhang, arXiv:2509.19012) stay inside the foundation family. Each treatment is excellent on its own terms, but staying inside one family means you never see the trade-off between them, and that trade-off is what actually determines what you should build. Chapters 4 through 7 walk through the classical, RL, and imitation families in enough depth that you can read a paper from any of them. Chapters 11 through 14 do the same for the foundation family, so that by the end of Part 4 you can read RT-1, OpenVLA, $\pi 0$, Helix, and GR00T N1 as variations on a small number of design choices rather than a disconnected list of model names.

The second is modern building blocks before modern systems. The transformer, the world-model recurrence, the diffusion or flow-matching action head: each is a piece of machinery that recurs across many VLAs, and each is easier to learn in isolation than inside a 7-billion-parameter system whose paper barely has room to explain it. Part 3 (Chapters 8 through 10) is where the building blocks live. Read it before Part 4, even if you're tempted to skip ahead.

The third is citing the canonical paper rather than the survey. With one exception (the Sapkota survey, the broadest VLA reference available at the time of writing, cited in several chapters as a map of the field), every model and method gets a single canonical citation: the original paper or the official technical report. Appendix E.2's reading list and Appendix F's Model Zoo collect everything in one place, so the chapter sections themselves can just point you to the one paper you actually need next.

The fourth is the engineering of building one, not just the theory. Part 5 (Chapters 16 and 17) covers what happens after the model is published. How do you fine-tune an OpenVLA checkpoint (arXiv:2406.09246) on your own teleoperated data without burning a week per attempt? How do you build a teleop dataset that doesn't waste your time? How do you A/B test on real hardware when every episode costs three minutes and a fresh setup? How do you wrap a safety layer around a learned policy you can't certify? Most existing books stop where the loss curve flattens. The interesting failures all happen after that point, which is where Part 5 tries hardest to be concrete.

What the book does not cover

Several reasonable topics are missing on purpose: a better existing reference already covers them, including them would double the book's length without serving its core argument, or, in two cases, the topic is moving too fast for a 2026 textbook to say anything useful.

Foundational RL theory beyond what an action model needs. Sutton and Barto (2018) remains the canonical introduction; the second edition is freely available and covers convergence proofs, contraction-mapping arguments, and function-approximation theory in more depth than this book attempts. Chapter 5 gives you enough RL to read modern locomotion and RL-fine-tuning papers, and Chapters 6 and 7 add the deep-RL machinery (DQN, PPO, SAC). If you want linear-MDP regret bounds, read Sutton and Barto, then Agarwal, Jiang, Kakade, and Sun's notes.

Classical motion planning beyond an introductory chapter. LaValle (2006) runs 1,000 pages on this topic alone and remains the right reference. Chapter 4 gives you STRIPS, IK, RRT, and computed-torque control at a level that lets you read a modern paper combining them with learning; it won't turn you into a motion-planning researcher. If your application lives in a domain where classical methods dominate, industrial pick-and-place, high-precision assembly, surgical robotics, read LaValle alongside this book.

Hardware design, electronics, and mechanical engineering. This book treats the robot as a controllable system with a known action space and a known observation interface. How you build the arm, choose the motors, calibrate the cameras, or design the gripper sits outside the scope here. Lynch and Park's *Modern Robotics* (Cambridge, 2017) is one of the cleaner mechanical-side references, and Siciliano et al.'s *Robotics: Modelling, Planning and Control* (Springer, 2010) covers the dynamics side.

Computer vision and language modeling, except where they get used directly. A vision-language-action model sits, by construction, on top of a vision encoder and a language model. Chapter 11 explains CLIP at the level of "what is the contrastive objective and what does the resulting embedding buy you," but doesn't develop the design space of vision transformers and doesn't derive the scaling laws of language models. For those, Goodfellow, Bengio, and Courville (2016) and Murphy (2022) remain the best textbook references, and the *Building LLMs from Scratch* community has produced several excellent recent treatments.

Multi-robot systems, swarm robotics, and most of mobile robotics. The action models in this book assume one robot and one task. Multi-agent coordination, market-based task allocation, and SLAM-heavy mobile robotics are their own subfields with their own textbooks. The action-model ideas here transfer fine (a foundation policy is a foundation policy whether the robot has legs or wheels), but the surrounding engineering differs enough that one chapter would have served worse than zero.

Two topics would have made the cut if the field had held still long enough. The first is video-pretrained action models without robot data: V-JEPA-style self-supervised video models that are turning out to be useful priors for action prediction. Chapter 9 sketches the world-model side and §18.3 returns to the open question, but a confident chapter-length treatment is still six months off. The second is reasoning-plus-action systems in the LLM-chain-of-thought sense, RoboBrain-style agents, Embodied-R1 (arXiv:2508.13998), where the model produces an explicit plan in text before producing actions. §18.4 catalogues the state of play, though a stable treatment is more like a year away.

What you should read alongside this book

A short reading list, in priority order, for the gaps left open.

No machine-learning background? Read the first six chapters of Murphy's *Probabilistic Machine Learning* (MIT Press, 2022) before Chapter 3. This book's own Chapter 3 is a thirty-minute refresher, not a first introduction.

No robotics background? Read the kinematics and dynamics chapters of Lynch and Park's *Modern Robotics* (Cambridge, 2017) before Chapter 4. Terms like joint space, task space, Jacobian, and end-effector pose get used without re-definition starting in Chapter 2.

Want deeper RL background? Read Sutton and Barto (2018) alongside Chapters 5 through 7. This book's RL chapters aim to give you the model-design vocabulary; Sutton and Barto gives you the theory underneath it.

Want to follow the VLA literature as it evolves past this book? Keep two surveys open in a browser tab: Sapkota et al. (arXiv:2505.04769) and the Pure VLA Survey (arXiv:2509.19012), read in that order since they overlap. The Embodied Arena leaderboard (arXiv:2509.15273) is the closest thing the field has to a community benchmark, and it's worth checking every few months.

Code, repositories, and what runs on what

The book pairs with a public code repository. Every chapter with a non-trivial code listing has a corresponding folder in the repo containing the full, runnable version. Two extended hands-on exercises, fine-tuning OpenVLA on a small LIBERO-derived dataset (Chapter 16) and building a safety-monitor wrapper around a deployed policy (Chapter 17), are the points where running the code beats reading it.

The default training environment is PyTorch with the HuggingFace `transformers` stack, since that's what every open-source VLA at the time of writing ships in (OpenVLA, SmolVLA arXiv:2506.01844, Octo arXiv:2405.12213, π0's open weights from arXiv:2410.24164). JAX shows up in two places, the Octo reference implementation and the π0 official release, and the JAX-specific differences get flagged where they matter. The hands-on exercises are sized to run on a single 24 GB consumer GPU (an RTX 4090 or similar) for the fine-tuning case, and on a CPU for everything else. The chapters describing pretraining at the scale of Open X-Embodiment (arXiv:2310.08864) are explicitly describing it, not asking you to reproduce it. That workload assumes a cluster.

A note on the field's pace

The first draft of this book named π0 (Black et al., 2024, arXiv:2410.24164) as the most recent flow-matching VLA. By the time the fourth draft was finished, GR00T N1 (arXiv:2503.14734) was out, the FAST action tokenizer (arXiv:2501.09747) had appeared, the Efficient VLA Survey (arXiv:2510.24795) had been written, and Helix-02 had been announced. The chapters that name specific models will date. The chapters that name design choices, discrete versus continuous actions, single-system versus dual-system, language-conditioned versus goal-image-conditioned, shouldn't. If a model name in this book is two years old and looks unfamiliar, treat it as a representative of its design choice rather than the current state of the art, and look it up in whichever survey the appendix points you at most recently.

With the scope of the book settled, §1.6 closes the chapter by collecting the four ideas you should leave it carrying.

1.6 Summary

Chapter 1 was, by design, the chapter with the least code in the book. It laid out the vocabulary, the history, and the shape of the field so the chapters that do contain code, and there's a lot of it coming, can move fast. This summary collects the load-bearing ideas in one place. If you've read the chapters in order, treat what follows as a checklist for what you should be able to do before turning the page. If you skipped ahead and came back, treat it as a refresher.

The four ideas worth carrying forward

Action is structurally different from perception and language. The output space of a robot policy, a vector of continuous joint commands or pose deltas emitted every twenty milliseconds, isn't the output space of a classifier or a token predictor. §1.1 named four properties that set it apart: the output space is high-dimensional, continuous, and constrained; the policy runs in a closed loop where each action changes the next observation; training distribution and deployment distribution diverge by construction (compounding error); and the cost of being wrong gets paid in physical hardware rather than benchmark points. Every architectural choice in the rest of the book, action chunking, diffusion heads, flow matching, runtime safety monitors, responds to one of these four properties. When you read a paper and ask why this design and not the obvious one, the answer is almost always one of the four.

An action model is a function with three slots. §1.2 introduced the anatomy: Slot 1 is the input (observations and instructions, in whatever modality), Slot 2 is the output (the action representation, symbolic, joint target, end-effector delta, torque, or a tokenized version of any of these), and Slot 3 is the training signal (the loss or reward that produced the function). The anatomy is deliberately model-agnostic. It applies to a STRIPS planner (structured predicates in, symbolic plan out, no training at all), to a SAC policy (observations in, action distribution out, reward as signal), and to OpenVLA (image plus text in, action tokens out, imitation loss on cross-embodiment data). Every paper for the rest of the book can be re-read as a set of choices for these three slots, and that re-reading is more useful than it sounds. It's how you notice that two systems with different names are making the same architectural choice, and that two systems with similar names aren't.

The history is one of bottlenecks moving, not methods replacing each other. §1.3 walked through six eras, symbolic planning, classical control, deep RL, end-to-end imitation, the foundation moment, the VLA scale-up, and the point of that walk was that each era's methods survived the transition into the next one. STRIPS-descendant planners still run inside warehouse robots that ship a VLA on top of them. Inverse-kinematics solvers still convert pose deltas to joint commands at the bottom of the OpenVLA stack. RL still produces the locomotion policies that VLAs ride on. The eras compound rather than replace. The corollary for an engineer: "should I use a classical method or a learned method" is rarely the right question. The right question is where the dividing line goes in this particular system, and the rest of the book is, in large part, a catalogue of where that line has been drawn in published systems and why.

The four families are a vocabulary, not a taxonomy of mutually exclusive camps. §1.4 named them: classical/analytical, reinforcement-learning, imitation, foundation/VLA. They're useful because they correspond to qualitatively different kinds of supervision, derivation from a model, reward maximization, demonstration matching, and pretrained-plus-fine-tuned demonstration matching, and once you can name the supervision, the design space narrows fast. A deployed system in 2026 typically contains pieces of at least two of these families, often three: a classical IK solver inside a learned policy, a VLA prompting a low-level RL controller, an imitation policy fine-tuned with RL on safety-critical edge cases. Reading the family vocabulary fluently is what lets you take a stack like that apart.

What you should be able to do now

Four concrete things, in increasing order of how much they matter for the rest of the book.

You should be able to read the abstract of a modern VLA paper, RT-1 (arXiv:2212.06817), OpenVLA (arXiv:2406.09246), $\pi 0$ (arXiv:2410.24164), and identify, sentence by sentence, which slot of the action-model anatomy each claim is talking about. Abstracts that say "we train on diverse data" are talking about Slot 3. Abstracts that say "we propose a new tokenization" are talking about Slot 2. Abstracts that say "we condition on language and a third-person camera" are talking about Slot 1. It's a small skill, but it changes how fast you triage a reading list, and a working roboticist reads on the order of two new VLA papers a week. The time you save on the ones that don't matter is time spent on the ones that do.

You should be able to place a named system in the right family and the right era. ALVINN (Pomerleau, 1988) is imitation, Era 3. DQN-Atari (Mnih et al., 2015) is RL, Era 4. RT-1 is foundation/VLA, Era 6. ANYmal's locomotion controller is RL by method, Era 4, but contemporary by deployment. This sounds pedantic, yet it's exactly how the rest of the book is organized, since Chapters 4 through 14 are structured by family and era, and being able to place a system on that map quickly tells you which chapter it'll show up in and what its likely failure modes are.

You should be able to sketch, on a napkin, the data and supervision pipeline for a new robot task you've just been handed. Say the customer wants the robot to empty a dishwasher. The four-family sketch runs: classical (PDDL plan plus IK plus motion planning, assuming a clean kitchen model), RL (a reward function, a simulator, sim-to-real transfer), imitation (a teleoperation rig, fifty demonstrations, a Diffusion Policy or ACT model), and foundation/VLA (an OpenVLA checkpoint, those same fifty demonstrations as fine-tuning data, a natural-language prompt). Within the first hour of a project you should be able to draw all four and name which one you'd actually build, and why. The real answer is usually some combination, and that combination is the design of the system.

You should be able to predict the failure mode of an action model from which family it sits in. Classical methods fail when the world model is wrong, an obstacle the planner never knew about. RL methods fail by reward hacking, the policy gets the reward in a way you didn't intend. Imitation methods fail by compounding error, the trajectory diverges as soon as the policy hits a state the demonstrator never visited. Foundation methods fail in all three ways at once, plus a fourth: they confabulate, producing actions that look plausible but are wrong because the pretraining prior over-extrapolated. When a production system starts failing, recognizing which family the failure belongs to is the first step toward fixing it, and that recognition is what the rest of the book teaches.

Where the chapter has set up the rest of the book

Three forward references are worth naming again here, since they structure what comes next. The three-slot anatomy from §1.2 is the spine of Chapter 2 (the first end-to-end VLA you train) and of every model walkthrough from Chapter 11 onward. The four families from §1.4 are the spine of Chapters 4 through 7 (the lineage) and Chapters 11 through 14

(the foundation family in depth). The compounding-error problem from §1.1 is the subject of §6.3 (DAgger), §6.5 (BC versus IRL versus RL trade-offs), and §17.2 (runtime monitoring as a deployment-time response to the same problem). When one of those threads resurfaces in a later chapter, the cross-reference isn't decorative. It's the second half of an argument this chapter started.

The chapter has not set up two things you might have expected. It hasn't defined "intelligence," "embodiment," or "generalization" with the rigor those words deserve. Each is a load-bearing concept in the field, and each gets its own section later on, §11.5 on when scale starts to pay off, §12.5 on what "emergent" actually means, §18.1 on cross-embodiment generalization, where they can be discussed against specific empirical results rather than treated as philosophical abstractions. If those are the questions you came to this book for, the first ten chapters are the infrastructure that makes those discussions productive instead of vacuous.

The chapter also hasn't given you any code. Chapter 2 changes that. The sample chapter already sitting in the project folder (Sample_Chapter_02_Your_First_VLA.docx) walks an OpenVLA fine-tune end to end, on a single GPU, using a small LIBERO-derived dataset (arXiv:2603.28301), and it's where the vocabulary established here first earns its keep.

§1.x closes the chapter with one hands-on exercise, designed to finish in under an hour with no GPU required, and the full reading list for the chapter.

1.x Hands-on exercise + chapter references

The Chapter 1 exercise needs no GPU, no code, and no robot. It's a reading-and-classification exercise, designed to take about ninety minutes, meant to make the four-family vocabulary from §1.4 and the three-slot anatomy from §1.2 actually stick. Resist the urge to skip it. Students who do it report afterward that it changed how they read the rest of the book. Students who skip it tend to do it anyway, implicitly and more slowly, somewhere around Chapter 11.

Exercise 1.x.1 — The four-paper triage

Pick four papers, one from each family, read each in abstract-plus-figures-only mode (twenty minutes per paper, no full read), and fill in a table. Recommended choices for a first pass:

- *Classical/analytical:* LaValle (2006), Chapter 5 of *Planning Algorithms*. RRT and its variants. (A textbook chapter rather than a paper because the classical family does not have a single "canonical paper" the way the others do.)
- *Reinforcement-learning:* Mnih et al. (2015), "Human-level control through deep reinforcement learning." *Nature* 518.
- *Imitation:* Pomerleau (1988), "ALVINN: An Autonomous Land Vehicle in a Neural Network." NeurIPS 1988. Or, if you want a modern instance, Chi et al. (2023), "Diffusion Policy," arXiv:2303.04137 (not in the local source mirror but freely available on arXiv).
- *Foundation/VLA:* Kim et al. (2024), OpenVLA, arXiv:2406.09246.

For each paper, fill in seven cells:

1. *Family* (one of the four).
2. *Era* (one of the six from §1.3).
3. *Slot 1 — input.* What does the model see? Be specific: image resolution, number of cameras, presence of proprioception, language conditioning.
4. *Slot 2 — output.* What does the model emit? Symbolic plan, joint targets, end-effector deltas, action tokens, continuous chunks? At what rate?
5. *Slot 3 — training signal.* Derivation, reward, demonstration, or pretraining-plus-fine-tune? On what data?
6. *Compounding-error response.* What does the paper do (if anything) to handle the closed-loop / open-loop mismatch from §1.1?
7. *One sentence on why this paper mattered.* No hedging.

A worked example, for ALVINN:

Cell	Value
Family	Imitation
Era	3 (first learned approaches, 1988 prototype)
Slot 1	30×32 grayscale camera, single front-facing
Slot 2	Steering angle (a single scalar), at ~10 Hz
Slot 3	Supervised regression on human-driving demonstrations
Compounding-error response	None explicitly; trajectory divergence was the dominant failure mode
Why it mattered	First demonstration that a neural network could output a control signal directly from pixels, on real hardware

Doing this for four papers takes ninety minutes the first time and goes faster on the second pass. The table is itself the artifact, so don't write prose around it.

Exercise 1.x.2 — The dishwasher problem, written down

Re-do the dishwasher worked example from §1.4 in writing, for a robot and kitchen you've actually seen. Choose any household task, emptying a dishwasher, sorting recycling, putting laundry in a basket, and write four short paragraphs (under 150 words each), one per family, describing how you'd build a robot to do it. For each, name the data or model the family would need (a clean kitchen geometry, a reward function, a teleoperation rig and some number of demonstrations, a base VLA checkpoint), the most likely failure mode (a missing obstacle, reward hacking, compounding-error trajectory divergence, a confabulated action), and one concrete reason you would or wouldn't pick this family for the task.

The exercise is the writing itself, not the answer. There's no scoring rubric. The value comes from noticing where you hesitate, and the places you hesitate are exactly the places later chapters will fill in.

Exercise 1.x.3 — Read one survey, end to end

Open Sapkota et al. (arXiv:2505.04769) and read it in one sitting. It's long but readable, and the local source mirror has it cached. Mark, with a pencil or a highlighter, three places where it refers to a model or method that doesn't yet make sense to you. Those three places will get answered by specific chapters of this book, usually one of Chapters 11 through 14. At the end of each of those chapters, come back to the survey and re-read the marked passages. The before-and-after delta is the single best self-assessment for whether the book is working.

If you can't find three places to mark, meaning every sentence of the Sapkota survey already makes sense, you've read the prerequisites this book assumes and can probably skip Chapter 3 (the math refresher) and parts of Chapter 8 (transformers for control), heading straight to Part 4 instead. Most readers will mark more than three.

Exercise 1.x.4 — A short language exercise

Pick five terms from the chapter and write a one-sentence definition for each without looking back: compounding error, the three-slot anatomy, action tokenization, the dividing line (between classical and learned components in a stack), and cross-embodiment generalization.

Then look back and check. The point isn't the score, it's noticing which of the five you can't define cleanly. Each one is a recurring concept, and the ones you can't yet define are the ones to keep a finger on while reading the next chapter.

Chapter 1 reading list

The works below are the ones cited or referenced across §1.1 through §1.6, grouped by purpose. Full bibliographic entries for everything cited in the whole book live in Appendix E.2; this list is just the chapter-local subset.

Foundational references

- Fikes, R. E., & Nilsson, N. J. (1971). "STRIPS: A new approach to the application of theorem proving to problem solving." *Artificial Intelligence* 2(3–4), 189–208.
- Pomerleau, D. A. (1988). "ALVINN: An Autonomous Land Vehicle in a Neural Network." *NeurIPS 1988*.
- LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press. Freely available at planning.cs.uiuc.edu.
- Argall, B. D., Chernova, S., Veloso, M., & Browning, B. (2009). "A Survey of Robot Learning from Demonstration." *Robotics and Autonomous Systems* 57(5), 469–483.
- Kober, J., Bagnell, J. A., & Peters, J. (2013). "Reinforcement Learning in Robotics: A Survey." *International Journal of Robotics Research* 32(11), 1238–1274.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

Deep RL and policy learning

- Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). "Human-level control through deep reinforcement learning." *Nature* 518, 529–533.
- Lillicrap, T. P., Hunt, J. J., Pritzel, A., et al. (2016). "Continuous control with deep reinforcement learning." (DDPG.) *ICLR 2016*.
- Schulman, J., Wolski, F., Dhariwal, P., et al. (2017). "Proximal Policy Optimization Algorithms." arXiv:1707.06347.
- Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). "Soft Actor-Critic." *ICML 2018*.

Imitation and behavior cloning

- Ross, S., Gordon, G., & Bagnell, D. (2011). "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." (DAgger.) *AISTATS 2011*.
- Jang, E., Irpan, A., Khansari, M., et al. (2022). "BC-Z: Zero-Shot Task Generalization with Robotic Imitation Learning." *CoRL 2022*.
- Shafiullah, N. M., Cui, Z., Altanzaya, A., & Pinto, L. (2022). "Behavior Transformers: Cloning k modes with one stone." arXiv:2206.11251.
- Chi, C., Feng, S., Du, Y., et al. (2023). "Diffusion Policy: Visuomotor Policy Learning via Action Diffusion." arXiv:2303.04137.
- Zhao, T. Z., Kumar, V., Levine, S., & Finn, C. (2023). "Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware." (ACT.) arXiv:2304.13705.

Foundation / VLA models

- Brohan, A., Brown, N., Carbajal, J., et al. (2022). "RT-1: Robotics Transformer for Real-World Control at Scale." arXiv:2212.06817.
- Driess, D., Xia, F., Sajjadi, M., et al. (2023). "PaLM-E: An Embodied Multimodal Language Model." arXiv:2303.03378.
- Brohan, A., Brown, N., Carbajal, J., et al. (2023). "RT-2: Vision- Language-Action Models Transfer Web Knowledge to Robotic Control." arXiv:2307.15818.
- O'Neill, A., Rehman, A., Maddukuri, A., et al. (2023). "Open X-Embodiment: Robotic Learning Datasets and RT-X Models." arXiv:2310.08864.
- Octo Model Team (2024). "Octo: An Open-Source Generalist Robot Policy." arXiv:2405.12213.
- Kim, M. J., Pertsch, K., Karamcheti, S., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246.
- Black, K., Brown, N., Driess, D., et al. (2024). "π0: A Vision-Language- Action Flow Model for General Robot Control." arXiv:2410.24164.
- Pertsch, K., et al. (2025). "FAST: Efficient Action Tokenization for Vision-Language-Action Models." arXiv:2501.09747.
- NVIDIA (2025). "GR00T N1: An Open Foundation Model for Generalist Humanoid Robots." arXiv:2503.14734.

Surveys and field-state references

- Sapkota, R., et al. (2025). "Vision-Language-Action Models: Concepts, Progress, Applications and Challenges." arXiv:2505.04769.
- Zhang, et al. (2025). "A Survey on Pure Vision-Language-Action Models." arXiv:2509.19012.
- "Embodied Arena: A Benchmark for Foundation Models in Embodied AI." arXiv:2509.15273.
- "Efficient Vision-Language-Action Models: A Survey." arXiv:2510.24795.
- Stanford HAI (2025). *AI Index Report — Robotics chapter.*

Background textbooks worth keeping nearby

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* MIT Press.
- Murphy, K. P. (2022). *Probabilistic Machine Learning: An Introduction.* MIT Press.
- Lynch, K. M., & Park, F. C. (2017). *Modern Robotics: Mechanics, Planning, and Control.* Cambridge University Press.
- Siciliano, B., Sciavicco, L., Villani, L., & Oriolo, G. (2010). *Robotics: Modelling, Planning and Control.* Springer.

Chapter summary

Chapter 1 set out a vocabulary for thinking about action models: the three-slot anatomy of inputs, outputs, and training signal, the six-era history that moved from STRIPS to $\pi 0$, and the four families, classical, RL, imitation, foundation/VLA, that organize the rest of the book. With that vocabulary in hand, you can read a modern VLA abstract and pull out its design choices, place a published system in the right family and era, sketch four candidate solutions to a new robot task, and predict an action model's failure mode from the family it belongs to. Chapter 2 is where all of this first earns its keep: a complete, end-to-end fine-tune of an OpenVLA checkpoint on a small benchmark, on a single GPU, in roughly seven pages.

Chapter 2 — Your first VLA, end-to-end

2.1 What we are going to build, and what is hidden inside

Chapter 1 was the vocabulary chapter. This one is the running-code chapter. By the end of it you'll have a Vision-Language-Action model, specifically OpenVLA-7B (Kim et al., 2024, arXiv:2406.09246), running on your machine: taking an RGB image of a simulated tabletop plus a natural-language instruction ("put the red block in the bowl") as input, and emitting low-level robot commands that drive a successful execution in the LIBERO simulator (Liu et al., 2023, arXiv:2306.03310). You'll also have it failing, repeatedly, in three characteristic ways that the rest of the book is structured to explain.

The order matters here: working first, theory after. The hard ideas in Chapters 3 through 10, what the chain rule means for a thirty-dimensional action space, how a policy gradient differs from a supervised gradient, why a transformer trained on text turns out useful for motor control, read easier once you've already watched a VLA do the right thing on your own laptop and the wrong thing on your colleague's. This chapter is the first-encounter artifact those later chapters keep referring back to. Don't skip the code. The muscle memory of standing up a model isn't something you acquire any other way.

The end state, in one paragraph

Here's concretely what you're about to run. An RGB image of a simulated tabletop scene, call it 256×256×3, uint8, comes off the LIBERO simulator. OpenVLA tokenizes the image (through a fused SigLIP and DINOv2 visual backbone) and the natural-language instruction (through a Llama-2-7B text tokenizer). The combined token sequence passes through a 7-billion-parameter transformer, which produces a short sequence of discrete action tokens, seven of them, one per axis of an end-effector pose delta plus a gripper bit. A detokenizer maps those tokens back into a 7-vector of continuous values. LIBERO steps the simulator with that 7-vector as the commanded action, returns the next image, and the loop repeats at roughly 5 to 15 Hz. Twenty to two hundred steps later, the task either succeeds, the episode times out, or the robot does something interesting and wrong. There's no magic between the picture and the motion, just a transformer doing next-token prediction over a vocabulary whose bottom 256 entries happen to mean "discretized action bins."

That paragraph is the spine of the chapter. §2.2 covers the environment setup so the spine can run. §2.3 walks the inference loop line by line. §2.4 covers what happens when the loop fails, which is most of the time on the first try. §2.5 maps each piece of the loop to the chapter that explains it in depth. §2.6 closes things out.

What is hidden inside a VLA

A modern VLA isn't a monolith labeled "AI." It's a stack of commitments, and much of the rest of this book is a guide to choosing among the alternatives at each layer. Four of those commitments matter at the level of the inference loop you're about to run.

The first commitment is how images become tokens. OpenVLA uses a fused SigLIP-plus-DINOv2 visual encoder: two pretrained vision transformers running in parallel, their patch embeddings concatenated, projected down, and fed into the language-model trunk. Other VLAs choose differently. RT-2 (Brohan et al., 2023, arXiv:2307.15818) uses

PaLI-X's native vision encoder; π0 (Black et al., 2024, arXiv:2410.24164) uses a separate vision trunk with a flow-matching head. This commitment determines how the model sees objects it hasn't been shown before. Chapter 11 dissects vision-language pretraining, and Chapter 12 compares OpenVLA's choice against its contemporaries.

The second commitment is how the policy turns intent into action. OpenVLA belongs to the discrete-token family: actions are a fixed-length sequence of categorical tokens, predicted with ordinary cross-entropy loss and decoded greedily at inference time. The alternative is a continuous-action head, a small diffusion or flow-matching model conditioned on the transformer's hidden state, sampling a smooth action chunk instead. RT-1 and RT-2 are discrete; π0 and Octo (arXiv:2405.12213) are continuous. Each choice trades off differently between simplicity, smoothness, and inference latency. Chapter 10 covers these heads in depth. For now, just know which family the model you're running belongs to (discrete), and that discrete-versus-continuous is the single largest architectural fork in modern VLAs.

The third commitment is what dataset taught the model physics. OpenVLA was pretrained on 970,000 robot trajectories drawn from Open X-Embodiment (O'Neill et al., 2023, arXiv:2310.08864), an aggregated corpus of teleoperated demonstrations from 22 institutions across 22 distinct robot embodiments. This mixture matters because the action vocabulary gets calibrated on the empirical distribution of motions inside it: bins sit where the training data spent its time. A motion common in the training data gets fine-grained resolution; a rare motion gets a single bin and a coarse output. Chapter 12 returns to Open X-Embodiment in detail. For §2.4, the practical upshot is that failures are often distributional failures, not modeling failures.

The fourth commitment, and the one most people miss, is what evaluation looks like. A cherry-picked clip of a robot succeeding once is easy to produce. A success rate measured across twenty seeds, three initial conditions, and two camera variations is hard to produce, and it's the only thing that should actually make you believe a result. Chapter 15 covers evaluation in depth; the version you'll do in §2.4 is small, a handful of LIBERO rollouts with logged seeds, but the discipline is the same discipline, and it scales.

A first concrete example

Suppose the LIBERO task is `libero_object` task 0, and the instruction is "put the apple in the bowl." A single inference step looks like this:

```
input image:   (256, 256, 3)  uint8   tabletop with apple, bowl, distractor
input prompt:  "In: What action should the robot take to put the apple in the bowl?\nOut: "
visual tokens: 256 patches × 2 encoders → 256 fused image tokens
text tokens:   ~20 tokens from the Llama-2 tokenizer
forward pass:  7B-parameter decoder; greedy decode for 7 steps
output tokens: [31882, 31894, 31881, 31905, 31888, 31875, 31999]
detokenized:   dx=-0.012, dy=+0.034, dz=-0.018, droll=+0.001, dpitch=-0.002, dyaw=+0.020, grip=1.0
sim step:      env.step([-0.012, 0.034, -0.018, 0.001, -0.002, 0.020, 1.0])
```

Those seven output tokens aren't text. They're integer IDs that happen to sit in the bottom 256 entries of the Llama-2 vocabulary, repurposed during training to mean "discretized action bin 0 through 255." When token 31882 gets decoded, the detokenizer subtracts a vocabulary base, gets a bin index in [0, 255], and maps that bin to a continuous value using the per-axis minimum and maximum from the training-data normalization statistics. The `unnorm_key` parameter in OpenVLA's `predict_action` call selects which embodiment's statistics to use. Pass the wrong key, and your output silently goes off-scale.

That seven-token output is the entire policy decision. Everything else, the billion parameters, the vision encoder, the language model, is upstream machinery whose whole job is choosing those seven integers well. When this chapter teaches you to debug, the seven tokens are the first thing you print.

Three load-bearing ideas you will meet again

Hidden inside this chapter's cheerful demo are three ideas that resurface again and again through the rest of the book.

Action tokenization turns control into a sequence-modeling problem. The architectural simplicity that follows, same transformer, same cross-entropy loss, same training infrastructure as a language model, is the reason a 7-billion-parameter VLA was feasible to train in the first place. The cost is discreteness: a 256-bin bucket over ±10 cm of translation gives you about 0.78 mm of resolution, fine for tabletop pick-and-place, coarse for threading a needle. Chapters 10 and 13 cover what reclaiming continuous resolution buys and what it costs.

Pretraining on internet-scale vision-language data buys semantics that pure robot datasets can't. OpenVLA's language understanding doesn't come from its 970,000 robot trajectories; it comes from the trillions of tokens of text and billions of images its backbones saw before any robot data ever touched them. That's what lets it parse "put the apple in the bowl" rather than only "task 17 of suite 4." Chapter 11 retraces the path, CLIP, BC-Z, RT-1, that made this transfer possible.

Evaluation is easy to fake in a cherry-picked clip and hard to do honestly across seeds, cameras, and initial conditions. A VLA that succeeds 9 out of 10 times on one initial pose isn't the same as one that succeeds 5 out of 10 across twenty randomized poses. §2.4 introduces the discipline of logging seeds and initial conditions, and Chapter 15 generalizes it. The temptation to publish the good clip is a permanent feature of this field. The only protection against fooling yourself is the rigor of the logs you actually keep.

What the chapter does not cover

A short list of things explicitly out of scope here. None of the training happens in this chapter; you'll run a pretrained checkpoint and won't modify its weights. Fine-tuning waits for Chapter 16. None of the math behind the transformer gets derived here, since Chapter 8 handles that. None of the safety wrapping a real deployment would require shows up here either, that's Chapter 17's job. Real-robot transfer isn't discussed at all; everything in this chapter runs in simulation. These omissions exist to keep the first VLA you run small enough to hold in your head. The rest of the book widens the aperture from here.

§2.2 begins the practical setup: GPU and software requirements, the weights download, and the LIBERO smoke test you should run before the model touches anything.

2.2 Setting up the environment (OpenVLA weights, LIBERO simulator)

Most "the VLA does not work" bug reports are setup bugs. The model loads, the simulator renders, and somewhere between them a driver mismatch or a half-downloaded weight file turns the whole pipeline into a generator of plausible-looking nonsense. This section asks you to separate those layers and verify each one in isolation before wiring them together in series. By the end of §2.2 you'll have, in this order, a working Python environment with a CUDA-matched PyTorch, a verified OpenVLA-7B checkpoint on disk, and a LIBERO smoke test that rolls out a scripted episode without the VLA touching anything. Only once all three lights are green does §2.3 wire them together.

Hardware floor

OpenVLA-7B (Kim et al., 2024, arXiv:2406.09246) has 7.3 billion parameters. In bfloat16 the weights alone run about 14 GB on disk and just under 15 GB on the GPU; add activations, KV cache, and the dual vision encoders, and a single inference step needs roughly 18 GB of memory in steady state. The practical hardware story shakes out like this.

A 24 GB-class consumer card (RTX 3090, 4090, A5000) is the comfortable floor for bfloat16 inference. You'll see no out-of-memory errors during ordinary use, and you can keep a few extra images in flight. A 16 GB card (RTX 4080, A4000, T4) works with 8-bit quantization via `bitsandbytes`, but quantization changes the numerics enough that some of the failure modes you'll study in §2.4 look different from the canonical ones. Useful to know, frustrating to debug the first time. A 12 GB card won't run OpenVLA-7B end to end without aggressive offloading, so this chapter assumes at least 16 GB and prefers 24.

CUDA sits on a separate axis. `nvidia-smi` reports the highest CUDA toolkit your installed driver supports; go above that and PyTorch will either refuse to import or, worse, import fine and then raise a cryptic `libcudart` error on the

first kernel launch. Match the PyTorch wheel index to your driver, not to whatever pip happens to suggest. The example below pins CUDA 12.4; adjust it to your own `nvidia-smi` output before running anything.

Disk and network are mundane but easy to botch. The OpenVLA weights run about 14 GB, and LIBERO's task assets, scene meshes, and demonstration data add a few more on top. Stage everything on the same physical disk as the project working tree, so the simulator doesn't silently fall back to a slow NFS path mid-rollout. The first weight download is the largest single network event in this chapter. On a metered link, mirror the checkpoint once with a pinned revision and set `HF_HUB_OFFLINE=1` on every subsequent invocation.

Python environment

Use a virtual environment. Not because conda is wrong (conda is fine), but because a fresh `venv` is the smallest reproducible unit, and small reproducible units are the only kind of dependency story that survives six months. The illustrative commands below aren't pinned versions, just the shape of the install. Your companion repository (`code/chapter_02/`) holds the pinned `requirements.txt` and the exact revision hashes used to produce this chapter's figures.

```
# CUDA 12.4 example — substitute your driver's CUDA major.minor.
python3 -m venv .venv
source .venv/bin/activate
pip install -U pip wheel

# Torch first: the wheel index must match nvidia-smi.
pip install torch torchvision \
    --index-url https://download.pytorch.org/whl/cu124

# Hugging Face stack for OpenVLA inference.
pip install "transformers>=4.40" accelerate sentencepiece \
    timm einops huggingface_hub
```

Verify three things before installing anything else. `python -c "import torch; print(torch.cuda.is_available(), torch.version.cuda)"` should print `True` and a CUDA version less than or equal to whatever `nvidia-smi` reports. `python -c "import torch; x = torch.randn(1024, 1024, device='cuda'); print((x @ x).sum().item())"` should run a CUDA kernel without raising. If either step fails, stop and fix the driver/wheel mismatch first. Nothing downstream will work, and the errors only get less helpful the further you push ahead.

Downloading the OpenVLA weights

The canonical home for OpenVLA-7B is the Hugging Face Hub repository `openvla/openvla-7b`. The library resolves this name automatically the first time you instantiate the model, but doing so leaves you at the mercy of whatever revision sits at `main` on download day. Pin a revision instead. The `snapshot_download` call below pulls the entire repo to a local cache, verifies SHA-256 hashes, and writes a file you can inspect.

```
from huggingface_hub import snapshot_download

local_dir = snapshot_download(
    repo_id="openvla/openvla-7b",
    # replace "main" with the commit hash for reproducibility
    revision="main",
    local_dir="weights/openvla-7b",
    local_dir_use_symlinks=False,
)
print("Weights at:", local_dir)
```

After the download finishes, a healthy local copy contains the model shards (`model-00001-of-00003.safetensors` through `model-00003-of-00003.safetensors`), a `config.json`, a `processor_config.json`, and a `dataset_statistics.json` file holding the per-embodiment normalization statistics. That last file is the most often overlooked and the most

consequential one; it's the table the detokenizer reads to convert the seven discrete action tokens back into a continuous 7-vector. Open it once and look at the keys. You'll see one entry per embodiment in Open X-Embodiment (O'Neill et al., 2023, arXiv:2310.08864). The `unnorm_key` parameter in §2.3's `predict_action` call selects which entry to use, and getting it wrong is one of the failure modes covered in §2.4.

A quick sanity check before LIBERO gets installed at all:

```
import torch
from transformers import AutoProcessor, AutoModelForVision2Seq

processor = AutoProcessor.from_pretrained(
    "weights/openvla-7b", trust_remote_code=True)
model = AutoModelForVision2Seq.from_pretrained(
    "weights/openvla-7b", torch_dtype=torch.bfloat16,
    trust_remote_code=True).to("cuda").eval()

print("Params:", sum(p.numel() for p in model.parameters()) / 1e9, "B")
```

This should print roughly `7.34 B` and use about 15 GB of GPU memory. If it hangs on the first call, the cause is almost always a partial download; delete the local directory and re-run `snapshot_download` with `resume_download=True`. If it loads but produces NaNs on a dummy forward pass, the same fix applies. Checksums aren't magic, and a process killed mid-download can leave a file whose shape is correct but whose bytes aren't.

Installing LIBERO

LIBERO (Liu et al., 2023, arXiv:2306.03310) is a benchmark suite for language-conditioned manipulation, built on RoboSuite and MuJoCo, with a collection of curated task families: `libero_object` (pick a named object out of distractors), `libero_spatial` (place at a named spatial relation), `libero_goal` (achieve a goal state described in language), and `libero_10` plus `libero_90` (longer-horizon compositions). The suite ships around 6,500 teleoperated demonstrations. This chapter uses none of them, but the same task definitions drive the evaluation in §2.4.

```
git clone https://github.com/Lifelong-Robot-Learning/LIBERO
pip install -e LIBERO
python -m libero.libero.benchmark.download_assets   # one-time, ~3 GB
```

MuJoCo wants three system libraries on a Linux machine, and they aren't always present on a fresh cloud image: `libGL`, `libEGL`, and `libosmesa6`. On Ubuntu, `sudo apt-get install libgl1 libegl1 libosmesa6 patchelf` covers the common cases. On a headless server, export `MUJOCO_GL=egl` before any LIBERO import; with a display attached, `MUJOCO_GL=glfw` works fine. The symptom of getting this wrong isn't an import error, it's an all-black observation array. The simulator runs, returns zeros for pixels, and the VLA's first action prediction looks plausible right up until you realize the image it was conditioned on was blank the whole time.

A smoke test that does not touch the VLA

Before any OpenVLA code talks to LIBERO, run a scripted episode with a random action. The point is to verify that the simulator produces images, that those images aren't black, and that the action interface matches what the VLA will eventually emit. The script below checks all three in about forty lines.

```
import os
os.environ.setdefault("MUJOCO_GL", "egl")   # headless rendering
import numpy as np
from libero.libero import benchmark
from libero.libero.envs import OffScreenRenderEnv

suite = benchmark.get_benchmark("libero_object")()
task = suite.get_task(0)
env = OffScreenRenderEnv(
```

```
    bddl_file_name=task.bddl_file,
    camera_heights=256, camera_widths=256,
)
obs = env.reset()
print("image shape:", obs["agentview_image"].shape,
      "non-zero pixels:", int(np.count_nonzero(obs["agentview_image"])))

# 7-dimensional action: [dx, dy, dz, droll, dpitch, dyaw, gripper]
rng = np.random.default_rng(0)
for step in range(40):
    action = rng.uniform(-0.02, 0.02, size=7)
    action[-1] = 1.0
    obs, reward, done, info = env.step(action)
env.close()
print("episode finished, reward at last step:", reward)
```

Three things need to be true at the end of that script. The image shape is `(256, 256, 3)`. The non-zero pixel count is large, tens of thousands, not zero. The episode runs for 40 steps without raising. If any of these fails, you don't yet have a working LIBERO install, and stacking the VLA on top will only make the next traceback harder to read.

Three setup failures that account for almost all of them

After watching dozens of readers stand this stack up, three failure modes dominate.

The first is the CUDA wheel mismatch. PyTorch imports successfully, `torch.cuda.is_available()` even returns `True`, and then the first real kernel call raises something like `CUDA error: invalid argument` or `libcudart.so.12: cannot open shared object file`. The fix is always to reinstall the PyTorch wheel from the index matching your driver's maximum CUDA version. There's no other fix; the error message stays unhelpful on purpose, because the failure sits below PyTorch's abstraction layer.

The second is the headless rendering trap. LIBERO imports, the simulator runs, the rollout finishes, and every image comes out black. This is the symptom of a missing `libEGL` or an unset `MUJOCO_GL`. The smoke test above catches it, since the non-zero pixel count comes out as zero. Without that check you find out hours later, after wondering why a VLA that behaved correctly on a development laptop is hallucinating actions on a cloud instance.

The third is the half-downloaded weight file. `model.from_pretrained` loads without complaint, the parameter count checks out, and the first forward pass returns NaNs anyway. The cause is almost always a network drop during `snapshot_download` that didn't raise, leaving a single corrupted shard inside an otherwise-valid directory tree. The fix is to delete the local cache and re-download with `resume_download=True`. Running the hash check on every shard is overkill for an interactive workflow, but it's the right move for a CI job.

None of these failures is unique to OpenVLA or LIBERO. They're the generic failure modes of GPU-plus-simulator stacks in 2026, and they're the cost of admission for running any large model on real hardware. Skipping the verification steps doesn't save time. It just relocates the time to a worse spot, after you've spent an hour suspecting the model instead of the plumbing.

A short reproducibility note

Save the exact versions you used: run `pip freeze > pinned.txt` after the install succeeds, and record the OpenVLA revision hash printed by `snapshot_download`. When you come back to this chapter in six months, or when a colleague reports a result you can't reproduce, those two files are what separate a debugging session from an archaeological dig. The book's companion repository keeps a `chapter_02/pinned.txt` checked in, and the rest of the book assumes you can do the same.

With a verified GPU, a pinned OpenVLA-7B checkpoint, and a LIBERO smoke test producing non-black images, the stack is ready. §2.3 opens the inference loop and walks it one line at a time.

2.3 Walking through the inference loop one line at a time

By the end of §2.2 you have a verified GPU, pinned OpenVLA-7B weights, and a LIBERO environment producing non-black images under a scripted policy. This section glues the two together. The result is a closed loop: image and language in, seven discrete tokens out, a 7-vector commanded to MuJoCo, a new image one timestep later. We'll walk that loop line by line, name each intermediate object, and stop at exactly the places where things can go quietly wrong without raising an exception. The model in question is OpenVLA-7B (Kim et al., 2024, arXiv:2406.09246), running against LIBERO's `libero_object` suite (Liu et al., 2023, arXiv:2306.03310).

The whole loop, in one listing

Read the following script before the prose below it. It's the entire chapter in twenty-five lines; everything after is annotation.

```
import os
os.environ.setdefault("MUJOCO_GL", "egl")
import numpy as np, torch
from PIL import Image
from transformers import AutoProcessor, AutoModelForVision2Seq
from libero.libero import benchmark
from libero.libero.envs import OffScreenRenderEnv

processor = AutoProcessor.from_pretrained("weights/openvla-7b", trust_remote_code=True)
model = AutoModelForVision2Seq.from_pretrained(
    "weights/openvla-7b", torch_dtype=torch.bfloat16,
    trust_remote_code=True).to("cuda").eval()

suite = benchmark.get_benchmark("libero_object")()
task = suite.get_task(0)
env = OffScreenRenderEnv(bddl_file_name=task.bddl_file,
                         camera_heights=256, camera_widths=256)
obs = env.reset()
instruction = task.language                          # "pick up the alphabet soup and place it in the basket"

for step in range(400):
    image = Image.fromarray(obs["agentview_image"][::-1])          # flip; LIBERO returns upside-down
    prompt = f"In: What action should the robot take to {instruction.lower()}?\nOut:"
    inputs = processor(prompt, image).to("cuda", dtype=torch.bfloat16)
    action = model.predict_action(**inputs, unnorm_key="libero_object", do_sample=False)
    obs, reward, done, info = env.step(action.tolist())
    if done: break
env.close()
```

That's the loop: a model instantiation, an environment, and a `for` step where an image becomes a prompt, the prompt becomes seven action tokens, the tokens become a 7-vector, and the simulator advances. Each of those transitions earns its own paragraph below.

Loading the processor and the model

`AutoProcessor.from_pretrained` returns a bundle: a SigLIP image processor, a DINOv2 image processor, and a Llama-2 tokenizer, all driven by a `__call__` that takes `(prompt, image)` and returns the dict of tensors the model expects. `trust_remote_code=True` is required because OpenVLA's processor class lives in the model repository rather than in the upstream `transformers` release; the flag tells `transformers` to import that class from the downloaded snapshot. It isn't optional, and skipping it produces a `KeyError` on the model's architecture string instead of a helpful message.

`AutoModelForVision2Seq.from_pretrained(..., torch_dtype=torch.bfloat16)` allocates the 7.3-billion-parameter model in bfloat16 directly, skipping the fp32-to-bf16 cast that would otherwise spike memory at load time. `.to("cuda")` moves the weights to the GPU; `.eval()` disables dropout and switches LayerNorm and GroupNorm into inference mode. After this line completes, `nvidia-smi` should report roughly 15 GB of allocated memory. See 28 GB instead, and the dtype got silently coerced to float32, usually because of an older `transformers` version that doesn't honor `torch_dtype` for vision-language models. Pin `transformers>=4.40` and the issue disappears.

Reading an observation out of LIBERO

`env.reset()` returns a dict with several arrays. The one that matters here is `obs["agentview_image"]`, a `(256, 256, 3)` uint8 array from the third-person "agent view" camera. Two non-obvious things about it. First, LIBERO returns images vertically flipped relative to what a human, or OpenVLA's vision encoder, expects. This is a MuJoCo rendering convention, not a bug, and `obs["agentview_image"][::-1]` corrects it. Skip the flip and the model sees the world upside down; the loop keeps running, the actions stay syntactically valid, and the success rate collapses without raising anything. It's the cleanest example of a silent failure in the whole loop, and the one §2.4 returns to first.

Second, `obs` also contains `obs["robot0_eye_in_hand_image"]`, a wrist camera view. OpenVLA-7B was trained primarily on third-person observations, so passing the wrist view as `image` produces coherent but task-irrelevant actions. Different VLAs expect different camera conventions: RT-2 (Brohan et al., 2023, arXiv:2307.15818) takes a single base-mounted view, π0 takes multiple, RDT-1B (arXiv:2410.07864) takes wrist plus base. Chapter 12 catalogs these choices. For OpenVLA in `libero_object`, `agentview_image` is the correct one.

`task.language` is the natural-language instruction baked into the LIBERO task definition. For `libero_object` task 0 it reads `"pick up the alphabet soup and place it in the basket"`. The string stays fixed per task; LIBERO doesn't paraphrase, by design, so the benchmark measures execution rather than language robustness. Robustness to paraphrasing gets studied separately in Chapter 15.

Formatting the prompt

OpenVLA was fine-tuned with a specific instruction template, and inference has to use it verbatim:

```
In: What action should the robot take to {instruction}?
Out:
```

Mismatch the template and the model still produces seven action tokens (the output layer can't do otherwise), but the tokens come from a distribution the model was never trained on. The symptom is action sequences that hover near zero or saturate at the gripper extremes regardless of the scene. This is the second of the three silent failures in §2.4. The fix is mechanical: copy the template from the OpenVLA model card and don't edit it. Casing matters less than spacing; the colon and newline matter most.

The processor call

`processor(prompt, image)` does three things. It tokenizes the prompt with the Llama-2 tokenizer into a `(1, T)` long tensor of token IDs, with `T` typically 20 to 25. It runs the image through both SigLIP and DINOv2 image processors (resize to 224×224, normalize with encoder-specific means and standard deviations, stack into a `(1, 2, 3, 224, 224)` tensor of pixel values). And it packages everything into a dict whose keys match the model's `forward` signature: `input_ids`, `attention_mask`, `pixel_values`. The `.to("cuda", dtype=torch.bfloat16)` call moves the tensors to the GPU and casts the floating-point ones; long tensors (token IDs) get silently left alone, which is exactly what you want.

A useful debugging habit at this point: print `inputs.pixel_values.mean()` and `inputs.input_ids.shape` once per episode. If the mean comes out exactly zero, the image was black (the headless-rendering failure from §2.2). If the input shape is unexpectedly short, the instruction string was empty, which happens when `task.language` gets read before `env.reset()` on some LIBERO versions.

The forward pass: `predict_action`

`model.predict_action(**inputs, unnorm_key="libero_object", do_sample=False)` is a thin wrapper around `model.generate` followed by an action detokenizer. Inside, three things happen in sequence.

The model generates exactly seven new tokens by greedy decoding. With `do_sample=False`, each token is the argmax of the next-token logits over the full Llama-2 vocabulary of 32,064 entries. Training pushes those argmaxes into the bottom 256 entries, the action bins, but nothing in the architecture forces that choice. If training has drifted, the model can emit a non-action token, and `predict_action` will raise. On the pinned checkpoint, in practice, this doesn't happen.

Those seven token IDs get mapped to bin indices in `[0, 255]` by subtracting the vocabulary base. Bin indices then map to continuous values using the per-embodiment statistics in `dataset_statistics.json`, keyed by `unnorm_key`. The statistics for `"libero_object"` give a `q01` (1st percentile) and `q99` (99th percentile) per axis from the training-data distribution; bin `k` maps to `q01 + (k / 255) * (q99 - q01)`. The output is a `(7,)` numpy array of floats: `[dx, dy, dz, droll, dpitch, dyaw, gripper]`.

The gripper bit is the seventh axis and is conceptually binary, but the detokenizer returns a continuous value, since the same code path serves embodiments with proportional grippers. LIBERO's Panda accepts the continuous value and thresholds internally: values above roughly 0.5 close the gripper, values below open it. Quantize at the boundary or leave it continuous, both work fine, though rounding to 0.5 makes debugging logs more readable.

The third silent failure lives here, in `unnorm_key`. Pass `"libero_object"` and your actions get scaled to the LIBERO Panda's training-time motion envelope. Pass `"bridge_orig"` instead (a Bridge V2 embodiment, present in Open X-Embodiment, O'Neill et al., 2023, arXiv:2310.08864), and your actions get scaled to a WidowX 250, typically 2 to 5 times too large for LIBERO's Panda controller. The model is doing the right thing in token space; the rescaling is wrong, and the robot flails. Always print `unnorm_key` once at the top of the loop, and again into the wandb run name if you're logging.

Stepping the simulator

`env.step(action.tolist())` advances LIBERO by one control cycle. The default control rate is 20 Hz, but RoboSuite frame-skips internally, so a single `env.step` actually advances MuJoCo by several physics substeps (typically five at 100 Hz physics). The returned `obs` is the next agent-view image. `reward` is sparse, typically zero until task success, when it becomes 1.0. `done` is `True` when the task succeeds or the episode exceeds the suite's horizon (300 steps for `libero_object`, 600 for `libero_10`). `info` carries task-specific diagnostics.

One subtlety: LIBERO's reward function is a completion check, not a shaped signal. The agent gets no partial credit for being close. This matters for evaluation in §2.4 and for training in Chapter 16, where the standard metric is success rate over many seeds, not mean reward. Sparse rewards also mean the only way to know an episode is going wrong before timeout is to watch the rendered video or instrument the loop with object-pose probes.

Throughput and the latency budget

On a 4090, the loop above runs at roughly 6 to 8 Hz: about 120 ms per `predict_action` call, plus about 30 ms for the LIBERO step. That's fast enough for tabletop manipulation in simulation, but slower than a teleoperator's natural rate, often 20 to 30 Hz. The dominant cost is the 7-billion-parameter transformer; the simulator is essentially free by comparison. Chapter 13 returns to this. $\pi 0$ (Black et al., 2024, arXiv:2410.24164) and other flow-matching VLAs achieve higher effective control rates by predicting action chunks rather than single steps, and by running smaller policy networks behind a larger language backbone. For OpenVLA on a consumer GPU, the rate above is simply the rate.

A 5-line debug print to add right now

Before §2.4 makes you watch the loop fail, instrument it. The lines below cost nothing and make every failure mode in §2.4 obvious:

```
print(f"step={step:03d} "
      f"img_mean={obs['agentview_image'].mean():.1f} "
      f"action={np.round(action, 4).tolist()} "
      f"reward={reward:.1f}")
```

If `img_mean` is ever near zero, the simulator broke. If `action` saturates at the bin edges every step, the prompt template is wrong. If `action` looks plausible but the robot drifts away from the scene, `unnorm_key` is wrong. And if everything looks plausible while `reward` is still zero at step 300, the model is doing exactly what it was trained to do, it's just that the training data never included this task.

With the inference loop running and instrumented, the next step is to make it fail on purpose, three times, and read the failures as diagnostic signal. §2.4 shows you how.

2.4 When it works and when it does not

The loop in §2.3 runs. That isn't the same as working. A loop that runs is a loop that completes 400 steps without raising; a loop that works is a loop that places the alphabet soup in the basket. Almost everything interesting about deploying a VLA happens in the gap between those two, and it's worth spending a section inside that gap before leaving Chapter 2. We'll do this in four passes: first the success case, what a clean run looks like on a task the model was demonstrably trained on; second the three silent failures §2.3 trailed, upside-down images, wrong prompt template, wrong `unnorm_key`; third the broader failure taxonomy you hit the moment you stray from `libero_object` task 0; fourth a brief look at why a single rollout tells you almost nothing, and what to measure instead.

What success looks like

Run the §2.3 script unchanged on `libero_object`, task index 0, with a fixed seed. On a single GeForce 4090 the episode takes about 50 to 70 seconds of wall clock. The instruction is "pick up the alphabet soup and place it in the basket." The Panda arm moves down and slightly forward, the fingers close on the soup can after roughly 60 to 90 control steps, the arm lifts, translates above the basket, and releases. `reward` jumps from `0.0` to `1.0` around step 180 to 220, `done` becomes `True`, the loop breaks. This sequence happens on roughly nine out of ten seeds; Kim et al. (2024, arXiv:2406.09246) report 88 to 90% success on `libero_object` for the released OpenVLA-7B checkpoint after their LIBERO-specific fine-tune, and that number reproduces within a couple of percentage points on a clean install.

Two things about that success run look unremarkable until they stop happening. First, the gripper closes once. It doesn't chatter, doesn't open and close every other step. A jittery gripper means the seventh-axis bin sits right at the 0.5 threshold and noise in the policy keeps flipping it, a sign of distribution mismatch rather than a working policy. Second, the arm's motion stays smooth across timesteps even though each step gets decoded independently. OpenVLA doesn't predict an action chunk; it predicts one 7-vector per inference call. The visual continuity of the resulting trajectory is an emergent consequence of the model having seen smooth demonstration trajectories during pretraining, not of any explicit smoothness loss. When a VLA's trajectory looks jagged, that's the second distribution-mismatch tell.

The three silent failures

A failure is silent when the loop keeps running, the tensors keep having the right shapes, and no exception gets raised, but the robot has quietly stopped doing the task. The script in §2.3 has exactly three of these built in, and each one is a one-character or one-keyword change away.

The image flip. Remove the `[::-1]` from `obs["agentview_image"][::-1]`. MuJoCo's OpenGL convention puts pixel row 0 at the bottom of the image; OpenVLA's vision encoders (SigLIP and DINOv2) were pretrained on web images where row 0 sits at the top. Without the flip, the model sees a world where the table hangs from the ceiling. The actions it produces stay syntactically valid 7-vectors, and the Panda still executes them, but the arm drifts away from the can and either swipes through it or stops at a height that has nothing to do with the actual geometry. Success rate on `libero_object` task 0 drops from around 90% to single digits. If you instrumented the loop with the five-line print from §2.3, the diagnostic is that `img_mean` looks fine (rendering works), but the trajectory itself makes no sense. Dump the agent-view PNG once per episode and the flip becomes obvious to the eye.

The prompt template. Change the prompt from `"In: What action should the robot take to {instruction.lower()}?\nOut:"` to anything else, even a seemingly equivalent rephrasing like `"What action should the robot take? {instruction}"`. The model still emits seven action tokens, since the output layer is unconditional on prompt format. But OpenVLA was fine-tuned with that exact `In:/Out:` scaffold (see the OpenVLA repo's `prompt_builder` and the discussion in arXiv:2406.09246 §3), and the language tower's attention pattern over the visual tokens depends on it. Without the scaffold, the seven generated tokens drift toward the action-bin centers (128 is the bin mapping to roughly the middle of every per-axis range), and the commanded actions hover near zero. The arm twitches and never reaches the object. The diagnostic: `np.round(action, 2)` returns values clustered near zero, or clustered at the per-axis extremes, every step, regardless of the scene.

The `unnorm_key`. Change `unnorm_key="libero_object"` to `unnorm_key="bridge_orig"`. Both keys exist in OpenVLA's `dataset_statistics.json`; `bridge_orig` is the Bridge V2 embodiment from Open X-Embodiment (O'Neill et al., 2023, arXiv:2310.08864), which used a WidowX 250 arm with a much larger workspace and faster commanded velocities than LIBERO's Panda. The model is doing the right thing in token space, still picking bins that correspond to "move down and forward," but the bin-to-meters mapping is wrong by a factor of 3 to 5 on translation axes. The Panda's controller saturates, the arm slams toward joint limits, and LIBERO either terminates the episode with an error flag in `info` or the simulation goes unstable. The diagnostic: magnitudes in your action print are obviously too large, |dx| exceeding 0.1 m per step for an embodiment that should be commanding 2 to 5 mm steps. This failure also makes a useful object lesson: action detokenization is embodiment-specific even when the policy backbone is shared, and the choice of `unnorm_key` is the cleanest example of a non-policy parameter that determines whether a VLA works at all. Chapter 11 returns to action tokenization in more depth; for now, treat `unnorm_key` as a required argument with no safe default.

These three failures aren't exhaustive, but they're the ones you'll hit first, and each corresponds to one of the three transformation boundaries inside the loop: pixels into encoder, instruction into tokens, tokens into meters. A useful rule for debugging any VLA wrapper is to check each boundary in turn before suspecting the policy itself.

When it does not work even with the script right

Now make the script correct again and change the task instead. Switch from `libero_object` to `libero_10`, the long-horizon suite. Pick task 2, which involves picking a plate from one fixture, moving it to a stove, then returning to retrieve a separate item. Run the same loop. On the released OpenVLA-7B, success drops from around 90% to roughly 50 to 55% (Kim et al., 2024). Nothing is wrong with the loop; it's identical to before. The policy is simply worse at sequencing two subtasks than at executing one. You'll watch episodes where the plate gets grasped correctly, placed correctly, and then the arm never returns for the second item, instead re-approaching the now-empty plate location and stalling. This is a behavioral failure of the model itself, not an integration bug, and there's nothing to fix on the integration side. The fix lives in Chapter 14, where dual-system architectures (Helix, GR00T N1; arXiv:2503.14734) introduce a slower high-level planner alongside the fast policy.

Distribution shift is the second category. LIBERO randomizes object positions on each reset but doesn't randomize lighting, camera angle, or distractor objects. The moment you deviate from LIBERO's defaults, adding a second can of soup to the scene, dimming the lighting, rotating the camera by 15 degrees, success on that same `libero_object` task 0 falls sharply. OpenVLA's pretraining mix (Open X-Embodiment plus LIBERO fine-tuning data) covers a great deal of visual variation, but the specific combination of LIBERO's renderer, camera, and lighting is exactly what the LIBERO checkpoint was tuned on. Visual robustness, and how to measure it, is a Chapter 15 problem.

Language brittleness is the third category. `task.language` for `libero_object` task 0 reads exactly "pick up the alphabet soup and place it in the basket." Rewrite the instruction in the prompt to "put the soup can in the bin," semantically identical, lexically different, and success drops by 10 to 30 percentage points depending on the seed. This isn't a bug in the model. It's a measurement of how much of OpenVLA's language conditioning is verbatim memorization of training prompts versus genuine semantic understanding. The numbers are public (Kim et al., 2024, Table 4), and the gap is large. RT-2 (Brohan et al., 2023, arXiv:2307.15818) and $\pi 0$ (Black et al., 2024, arXiv:2410.24164) close it partially by relying on larger language backbones. Neither closes it fully.

The fourth category has no satisfying fix at this layer of the stack: the model can fail at a task inside its training distribution simply because the per-episode physical configuration is unfavorable. The soup can spawns leaning against the basket lip, the gripper-can contact angle is bad, or the friction parameters MuJoCo sampled for this

rollout cause the can to slip mid-lift. These failures are stochastic at the level of the simulator. They contribute to the roughly 10% of episodes that fail even on the easiest LIBERO suite, and they aren't a sign anything is actually wrong.

Why one rollout does not tell you anything

The corollary of stochastic failure is that a single rollout carries almost no information. Run task 0 once and it succeeds, you still can't distinguish "the model is at 90% success rate" from "the model is at 10% success rate and you got lucky." The standard evaluation protocol for LIBERO, and for VLA work generally, runs 50 episodes per task across all 10 tasks in a suite, with seeded initial conditions, reporting mean success rate plus standard error. The OpenVLA paper uses 500 episodes per suite; reproducing within ±2% needs at least 100. On a single 4090, 500 episodes at roughly 60 seconds each comes out to about 8 hours of wall clock per suite, a useful planning number for the rest of the book and for §2.5, which covers what to chase and what to skip.

The instrumentation from §2.3, `step`, `img_mean`, `action`, `reward`, becomes especially useful during a 50-episode sweep. Dump it to a `.jsonl` file per episode; the file stays small (a few hundred KB per episode at 20 Hz), human-readable, and analyzable after the fact. A useful descriptive statistic is the step at first nonzero gripper command: across successful episodes it's tightly distributed (gripper closes between step 60 and 100), while across failed episodes it's bimodal (either never closes, or closes at step 5). That single statistic catches roughly 80% of the failure modes above without watching a single video.

With OpenVLA working on `libero_object`, three named failure modes debugged, and an honest read on where the model breaks, the remaining question is what the rest of the book does with this baseline. §2.5 lays that out.

2.5 What is left for the rest of the book

By this point you have a working OpenVLA running on LIBERO, three named failure modes you can reproduce, and an honest read on where the model breaks. That's more than most newcomers to VLAs see in their first month. It's also less than 5% of what's actually going on inside the system you just ran. The remaining 95% is the rest of the book.

This section is a map, not new content. It takes each component of the loop from §2.3, the image encoder, the tokenizer, the transformer, the action detokenizer, the simulator, the training data behind all of it, and points to the chapter that explains it in depth. Treat the map as a reading guide. You don't have to read the book in order. You do have to know where each idea lives.

The loop, decomposed

The §2.3 loop has six things in it that we've so far treated as black boxes: a vision encoder, a language tokenizer plus a language model trunk fused with the vision tokens, an action head producing discrete tokens, a detokenizer mapping tokens to a 7-vector, a simulator stepping physics forward, and, invisible at inference but determinative of everything, the training dataset and objective that made the weights what they are. Six boxes, twelve chapters of unpacking.

The vision encoder you ran, a fused SigLIP-and-DINOv2 stack from Kim et al. (2024, arXiv:2406.09246), is one specific instance of an idea two decades older than VLAs. Chapter 11 traces the path from CLIP through BC-Z through RT-1 (Brohan et al., 2022, arXiv:2212.06817), the moment language-conditioned robotic policies became possible at scale. Chapter 12 covers what changed when the vision-language backbone grew from a few hundred million parameters to several billion, the RT-2 (Brohan et al., 2023, arXiv:2307.15818) and OpenVLA story. If you want to understand why the SigLIP-and-DINOv2 fusion won out over a single encoder, those two chapters together make the argument. Short version for now: SigLIP gives semantic alignment with text, DINOv2 gives geometric features that survive lighting and viewpoint change, and the fusion buys you both at the cost of slightly more compute per frame.

The language tokenizer is the most boring component in the loop and the hardest one to actually think about. Boring because it's just the standard Llama-2 byte-pair tokenizer. Hard because re-using the bottom 256 token IDs as action bins is the trick that lets a language model do control without any architectural changes. Chapter 11 explains action

tokenization for the first time; Chapter 13 returns to it with FAST (Pertsch et al., 2025, arXiv:2501.09747), a more compressed encoding delivering smoother control on the same backbone. The "256 bins per axis" choice in OpenVLA is one point on a Pareto frontier between resolution, sequence length, and decoding latency, and Chapters 11 and 13 are where that frontier gets drawn out.

The transformer trunk is, mathematically, the same object you'll meet in Chapter 8. The two pages on attention there stay deliberately sparse: just enough to read the rest of the book without misconceptions. Chapter 8 also covers Decision Transformer and Trajectory Transformer, the lineage that established "control as sequence modeling" before VLAs took the idea to foundation-model scale. If your background is NLP or CV and you already know transformers, Chapter 8's transformer pages are skippable. The Decision Transformer pages aren't, since they provide the framing under which a next-token-prediction model can plausibly act as a controller. If your background is classical robotics and the word "transformer" is opaque, read Chapter 3 first, then Chapter 8.

The action head, the part emitting the seven discrete tokens, gets covered twice, from two different directions. Chapter 10 covers generative heads in general (diffusion, flow matching, masked prediction), with Diffusion Policy and ACT as the canonical imitation-learning representatives. Chapter 13 covers $\pi 0$ (Black et al., 2024, arXiv:2410.24164) specifically, which replaces the discrete head you ran today with a flow-matching head emitting continuous action chunks. The trade-off, previewed here: discrete heads are simpler to train and debug but cap your control resolution and produce jagged trajectories at high frequency, while continuous heads run smoother and faster in chunks but are harder to train and more sensitive to data quality. The field hasn't settled this yet, and the book gives both sides their fair share.

The detokenizer, the unassuming function mapping token 31882 to `dx=-0.012`, is the smallest piece of code in the loop and the source of more silent bugs than the rest of the model combined. You met one of those bugs in §2.4 already (the wrong `unnorm_key`). Chapter 11 covers detokenization formally; Chapter 12 covers how Open X-Embodiment (O'Neill et al., 2023, arXiv:2310.08864) calibrates per-embodiment statistics; Chapter 15 explains why "21 different embodiments" is a statement that requires 21 different detokenizers, and what that means for generalization claims. This is one of those components you don't appreciate until it eats a week of your time.

The simulator is the one box in the loop this book deliberately treats as a tool rather than a subject. LIBERO (Liu et al., 2023, arXiv:2306.03310) is one of several robotics simulators a modern VLA gets evaluated on, alongside CALVIN, RoboCasa, and SimplerEnv. Chapter 15 gives them all a tour and tells you which to reach for in which situation, and Appendix D is the practical setup guide. We won't derive contact dynamics or rigid-body integration in this book; that ground is covered adequately elsewhere, and the page budget is finite. What we do cover is how to evaluate a policy in simulation in a way that correlates with real-robot success, which turns out to be the harder problem of the two.

The training data and objective make up the largest topic in the book by page count, and the most easily underestimated by effect. OpenVLA's weights are a function of three ingredients: the Llama-2 and SigLIP-and-DINOv2 backbones (pretrained on internet-scale text and image-text data), the 970k-trajectory Open X-Embodiment dataset, and LIBERO-specific fine-tuning data. The first is covered as prerequisite material in Chapter 3 and as architectural context in Chapter 11. The second is the centerpiece of Chapter 12 and §15.2. The third is the kind of thing that gets one paragraph in a paper and an entire Chapter 16 in a book trying to teach you to fine-tune your own model. The objective itself, cross-entropy over action tokens, is the simplest possible loss for a sequence model and the right place to start. Chapters 6 and 10 cover the imitation-learning alternatives that add structure (DAgger, diffusion, flow matching), and Chapters 5 and 7 cover the reinforcement-learning alternatives that change the loss entirely.

A map by chapter

Here's the same content rotated 90 degrees: every chapter, in order, with the part of today's loop it explains.

Part 1, Foundations. Chapter 3 is the math-and-ML prerequisite kit, calibrated specifically for action models: enough linear algebra to keep up with end-effector kinematics, enough probability to read RL pseudocode, enough PyTorch to follow a 50-line training loop. It doesn't stand on its own as a general math reference; it's tuned to exactly the prerequisites used elsewhere in the book.

Part 2, The lineage. Chapters 4 through 7 cover the four families of action models that came before VLAs, in the order they were invented. Chapter 4 is classical: STRIPS, PDDL, inverse kinematics, computed torque. Chapter 5

is reinforcement learning from first principles: MDPs, value iteration, Q-learning. Chapter 6 is imitation learning, the loss family OpenVLA actually uses. Chapter 7 is deep RL: DQN, PPO, SAC. None of these methods are obsolete; they're alive in every modern robot stack, often inside the same robot as a VLA. The book isn't claiming VLAs replaced them. The claim is that VLAs are best understood as a particular synthesis of imitation learning and large-scale pretraining, and you understand that synthesis better once you know what went into it.

Part 3, Modern building blocks. Chapter 8 covers transformers and sequence models for control. Chapter 9 covers world models, the model-based alternative to a feed-forward VLA, with RSSM and Dreamer as the imitation-free representatives and Genie and V-JEPA as the video-pretrained ones. Chapter 10 covers diffusion and flow matching for action generation. Together these three chapters supply the architectural vocabulary the foundation-model chapters use without re-deriving.

Part 4, Foundation action models in depth, is where the book spends its longest sustained argument. Chapter 11 retraces the CLIP-to-BC-Z-to-RT-1 path. Chapter 12 covers the scaling moment: RT-2, OpenVLA, Octo (arXiv:2405.12213). Chapter 13 covers $\pi 0$ and the move to flow-matching action heads. Chapter 14 covers dual-system architectures like Helix and GR00T N1 (arXiv:2503.14734), which add a slower high-level planner on top of a fast policy. Chapter 15 covers datasets and evaluation in detail. By the end of Part 4, you should be able to pick up a new VLA paper on arXiv and place it on the map within about fifteen minutes.

Part 5, Building with action models. Chapter 16 covers fine-tuning a VLA for your own robot, the practical question most readers of this book will eventually want answered. Chapter 17 covers evaluation, safety, and deployment for VLAs that have to do something other than look good in a demo. Chapter 18 is the open-problems chapter: embodiment transfer, long-horizon tasks, video pretraining, reasoning-plus-action. It has the shortest shelf life of any chapter, since the field will move underneath it, but it earns its keep by telling you what to read after the book ends.

How to read the rest of the book

Read straight through and you'll spend roughly one to two evenings per chapter on the technical chapters, a weekend each on the foundation-model chapters. That's honest pacing for an upper-undergrad or first-year grad student. If you're short on time and just want to get back to standing up your own VLA, the minimum path runs Chapter 3 (only the parts you don't already know), Chapter 6, Chapter 8, Chapter 11, Chapter 12, Chapter 16. Six chapters, covering the imitation loss, the architecture, the lineage, the current state of the art, and the practical fine-tuning recipe. The rest is depth and context.

If you came to this book wanting to understand one specific model, $\pi 0$, GR00T N1, RT-2, the table in Appendix F (the VLA model zoo) cross-references each model to the chapters explaining its components, and the reading list in Appendix E.2 gives the canonical paper for each. Reading in that order works too; plenty of practitioners do it that way.

The next section closes Chapter 2.

2.6 Summary

Chapter 2 was the running-code chapter. You stood up an OpenVLA-7B checkpoint (Kim et al., 2024, arXiv:2406.09246), pointed it at a LIBERO tabletop scene (Liu et al., 2023, arXiv:2306.03310), watched it succeed on the task it was trained for, and watched it fail in three specific ways once you removed something it quietly depended on. That's the artifact the rest of the book keeps pointing back to. This summary collects the load-bearing ideas in one place. It's shorter than §1.6 because the chapter was shorter, but structured the same way so you can use it as a checklist before turning the page.

The four ideas worth carrying forward

A VLA is six boxes, not one black box. §2.1 named the four commitments (how images become tokens, how the head emits actions, what dataset taught the model physics, what evaluation discipline gets enforced). §2.3 exposed the loop those commitments live inside: a visual encoder, a language tokenizer, a transformer trunk, an action head

emitting discrete tokens, a detokenizer mapping tokens to a 7-vector, and a simulator closing the loop with a fresh image. Six boxes, each one a component you can swap, debug, or replace independently. The temptation when reading a new VLA paper is to treat the model as an undifferentiated blob of parameters. The discipline this chapter pushed on you instead is reading the paper as a set of choices for each of those six boxes. When you read that π0 (Black et al., 2024, arXiv:2410.24164) "uses flow matching for action," you should now be able to locate that sentence at box four, the action head, and ask what it implies for box five (the detokenizer becomes a flow-matching sampler) and box three (the training loss changes from cross-entropy to a flow-matching objective). That re-reading is the whole point of having stood up the loop yourself.

Action tokenization is the architectural trick that made the rest possible. §2.1 and §2.3 circled back to this idea three times because it keeps mattering. The seven integers OpenVLA emits per step aren't text tokens reused at inference; they're integer IDs in the bottom 256 entries of the Llama-2 vocabulary, repurposed during training to mean "discretized action bin 0 through 255." Cross-entropy loss on those tokens is the entire training objective. The architectural simplicity that follows, same transformer, same optimizer, same training infrastructure as a language model, is the single biggest reason a 7-billion-parameter VLA was a tractable engineering project at all. The cost is discreteness, and it shows up in §2.4 as the third silent failure: a wrong `unnorm_key` produces seven perfectly plausible integers that decode to a wildly off-scale 7-vector. Chapters 10 and 13 take the discrete-versus-continuous trade-off apart in detail. What you should carry forward from Chapter 2 is simply that the trade-off exists, that OpenVLA sits on one side of it, and that the model running on your machine right now is a discrete-action VLA whose smoothness ceiling is set by bin width.

Running and working are not the same loop. §2.4 is the chapter's clearest payoff. A loop completing four hundred steps without raising an exception isn't a loop that solves the task, and the gap between the two is exactly where silent failures live. The three you saw, a flipped image (LIBERO returns the agent-view buffer upside down), a wrong prompt template (OpenVLA's training-time format isn't the obvious one), a wrong `unnorm_key` (per-embodiment normalization statistics are how the detokenizer recovers physical units), aren't a complete list. They're a kind of bug. Each one has the same shape: a piece of metadata that was implicit in the training data and that you have to make explicit at inference time, with nothing raised if you get it wrong. The general lesson, one §17.4 returns to, is that the most expensive bugs in a deployed VLA are the ones that don't crash. Chapter 2 gave you three reproducible examples on a laptop; Chapter 17 generalizes the diagnosis discipline to hardware. The chapters in between are largely about the kinds of metadata, embodiment, camera pose, action normalization, prompt template, that have to travel with a model from training to inference.

Evaluation must outlast the cherry-picked clip. §2.4 closed with the point that a single rollout tells you almost nothing. Twenty rollouts on twenty logged seeds, across three initial conditions, with one camera variation, is the minimum unit of evidence that lets you compare two versions of the same model honestly. The version of that discipline you ran in §2.4 was small; the version Chapter 15 develops scales to real-robot evaluation, where variance runs higher and each trial costs minutes rather than seconds. The skill worth internalizing here is logging the seed and the initial condition before you watch the rollout, not after. The temptation to publish the good clip is a permanent feature of this field. The only protection against fooling yourself is the rigor of the logs you actually keep.

What you should be able to do now

Four concrete things, in increasing order of how much they matter for the rest of the book.

You should be able to re-run a single LIBERO rollout from a logged seed and reproduce the trajectory step for step. This is the smallest unit of the discipline §2.4 introduced. Can't reproduce your own rollout? Then you can't debug it, and if you can't debug it, you can't improve the policy. The recipe is the one from §2.3: pin the simulator seed, pin the initial condition file, pin the model dtype, and log the action vector at every step. When you flip to Chapter 16 to fine-tune on your own data, this same reproducibility hygiene becomes the difference between a fine-tune that converges and one you simply can't diagnose.

You should be able to read a new VLA inference script and place each line in one of the six boxes from §2.3. Concretely: hit a fresh policy on Hugging Face, say an OpenVLA fine-tune for a new embodiment or a community port of Octo (arXiv:2405.12213), and you should be able to skim its `predict_action` (or equivalent) call and identify which line tokenizes the image, which tokenizes the prompt, which runs the transformer trunk, which samples the action, and

which maps that sample back to physical units. This six-box reading skill is what the rest of Part 4 assumes you already have. Chapter 11 formalizes it; Chapter 2 gave you the practice reps.

You should be able to predict which of the three silent failures from §2.4 a given bug is, from its symptom alone. Robot moves in slow random scribbles: wrong prompt template, the model isn't in action-decoding mode. Robot moves decisively in the wrong direction: flipped image, the policy is acting on a world that doesn't exist. Robot moves at the wrong scale, way too small or way too large: wrong `unnorm_key`, the detokenizer is applying the wrong embodiment's statistics. None of these symptoms throws an exception. The mapping from symptom to cause is the muscle separating someone who's deployed a VLA before from someone who hasn't, and Chapter 2 gave you the three reps needed to start building it.

You should be able to use the §2.5 map to plan your own reading path. Short on time? The minimum path runs Chapter 3 (only the parts you don't already know), Chapter 6 (imitation learning), Chapter 8 (transformers), Chapter 11 (CLIP to BC-Z to RT-1, arXiv:2212.06817), Chapter 12 (the scaling moment), and Chapter 16 (fine-tuning). Project-driven and want the canonical reference for one specific model instead? Appendix F (the VLA model zoo) and Appendix E.2 (the chapter-by-chapter reading list) cross-reference everything. The book is designed to be read non-linearly after Chapter 2, and the map is what lets you do that without losing your place.

Where the chapter has set up the rest of the book

Three forward references are worth naming again, since they structure what comes next. The six-box decomposition from §2.3 is the spine of every model walkthrough in Part 4 (Chapters 11 through 14): each chapter re-enters the loop and replaces one or two boxes. The three silent failures from §2.4 get revisited in §17.2 (runtime monitoring as the deployment-time response to the same class of bug) and §15.4 (real-robot evaluation, where higher variance makes silent failures hardest to catch). The four commitments from §2.1, image encoding, action head, training data, evaluation discipline, are the four axes Chapter 12 uses to compare RT-2 (arXiv:2307.15818), OpenVLA, and Octo head-to-head, and the same four axes Chapter 13 uses to argue for $\pi 0$'s design.

The chapter hasn't covered three things you might have expected. It hasn't trained anything; you ran a pretrained checkpoint and modified no weights, and Chapter 16 is the training chapter. It hasn't derived any of the math behind the transformer or the cross-entropy loss; Chapters 3 and 8 handle that, in that order. And it hasn't touched a real robot at all, since everything you ran was in simulation. Chapter 17 covers the gap from simulation to deployment, and Chapter 15 covers the evaluation methodology that has to bridge it. These omissions are deliberate. The point of Chapter 2 was keeping the first VLA you ran small enough to hold in your head, with the implicit promise that the rest of the book widens the aperture without throwing away the picture you've just formed.

§2.x closes the chapter with one hands-on exercise, a deliberate variation on the §2.3 script that reproduces one of the three silent failures on demand, and the full reading list for the chapter.

2.x Hands-on exercise + chapter references

The Chapter 1 exercise was reading and classification; you sat with four papers and a table. The Chapter 2 exercise is a coding exercise instead. You already have a working OpenVLA-on-LIBERO loop on your machine, so you already have everything it needs. What you don't yet have is the muscle that turns a working loop into a reliable diagnostic instrument, and the only way to build that is to break the loop on purpose and read the symptoms cold. Budget about two hours. The deliverable is a short text file, not a model, not a notebook, not a chart, that captures three reproduced failures, the diagnostic signature of each, and the one-line fix.

Exercise 2.x.1 — Reproduce the three silent failures on demand

Take the §2.3 inference loop and copy it to a new file called `failure_drills.py`. Wrap the inner loop in a function that accepts three toggles, defaulting to the correct values:

```
def run_episode(flip_image=True,
                prompt_template="default",
                unnorm_key="libero_object",
```

```
                seed=0,
                max_steps=400):
    ...
```

Inside the function, branch on each toggle. `flip_image=False` removes the `[::-1]` slice. `prompt_template="rephrased"` substitutes the string `"What action should the robot take? {instruction}"` for the canonical `In:/Out:` scaffold. `unnorm_key="bridge_orig"` substitutes the Bridge V2 embodiment statistics from Open X-Embodiment (O'Neill et al., 2023, arXiv:2310.08864) for the LIBERO ones. Seed both the LIBERO suite and the Python `random` and `numpy` generators from the `seed` argument, so the initial condition stays fixed across runs.

Run the function four times: once with all defaults (the baseline), then three more times with exactly one toggle flipped per run. Log to a `.jsonl` file, one line per step, with the five fields from §2.3: step index, `img_mean`, the seven-element rounded action vector, `reward`, and a single string tag (`"baseline"`, `"no_flip"`, `"bad_prompt"`, `"bad_unnorm"`) marking which run produced the line. The file should land between 1600 and 2400 lines for four runs of roughly 400 to 600 steps each, small enough to grep comfortably.

The diagnostic step is the one that matters. Open the `.jsonl` in whatever column-oriented tool you prefer (`jq`, pandas, a spreadsheet) and write three predicates, one per failure mode, that take only the action vector and `img_mean` columns as input and classify a row as belonging to one of the four runs. Concretely:

1. *Bad prompt:* mean over time of `np.linalg.norm(action[:6])` is below roughly 0.05, every step, regardless of `img_mean`. The arm twitches near zero; the language tower never enters action-decoding mode.
2. *Bad `unnorm_key`:* `np.max(np.abs(action[:3]))` exceeds 0.1 m on a non-trivial fraction of steps. The translation magnitudes are off by the LIBERO-vs-Bridge ratio (roughly 3 to 5×).
3. *No flip:* harder. `img_mean` is unchanged from baseline, and the action magnitudes are reasonable. The signature is in the *trajectory shape*, not in any per-step statistic. The clearest tell is that the episode never raises `reward > 0` and `done` never goes True, but `action` magnitudes look healthy and gripper commands cycle at roughly the same rate as the baseline. The diagnosis is by elimination — the model behaves like a working model on a world that does not exist.

Write the three predicates as functions, run them on the four logs, and verify each tagged run matches exactly one predicate. If the predicates disagree with the tags, say the `no_flip` log fires the bad-prompt predicate, your toggle is wrong, not the predicate, and it's worth re-reading §2.4 before moving on.

Save the four `.jsonl` files and the predicate file to a directory called `drills/`. You'll return to it in Chapter 17, where the same predicates become the kernel of a runtime monitor running on a real robot. The drills you write today are a first sketch of the diagnostic vocabulary §17.2 formalizes later.

Exercise 2.x.2 — The 50-episode sweep

Take the corrected loop (all defaults, no toggles flipped) and sweep it across fifty seeds on `libero_object` task 0. Each episode takes roughly 60 seconds on a 4090, so the sweep runs about 50 minutes of wall clock; do this overnight or alongside something else. Log the same five fields per step, plus the per-episode final `reward`, the step at first nonzero gripper command (the seventh action element crossing 0.5), and wall-clock time.

The deliverable is a single number, the success rate over fifty episodes with standard error, plus a one-sentence comparison against the published number from Kim et al. (2024, arXiv:2406.09246), who report roughly 88 to 90% on `libero_object` for the released checkpoint. Land within ±3 percentage points and your install is healthy. Fall more than five points below the paper, and treat the gap as a bug, not a result; work backwards through §2.2 (environment) and §2.3 (loop) until you find the discrepancy. The most common cause is mixed-precision drift from a non-bfloat16 dtype. The second most common is a stale `dataset_statistics.json` from a pre-release OpenVLA checkpoint.

Then produce one descriptive plot. The plot isn't a metric, it's a discipline. On the x-axis, the step at first nonzero gripper command; on the y-axis, a histogram of episodes colored by final outcome (success versus failure). You should see what §2.4 promised: successful episodes cluster between step 60 and 100, while failed episodes go bimodal at "never closes" and "closes immediately." That single plot is the most informative one-pager you can produce about a discrete-action VLA, and the construction generalizes; Chapter 15 builds the same plot for fine-tuned policies and real-robot rollouts.

Exercise 2.x.3 — A one-page paper read

Open the OpenVLA paper (Kim et al., 2024, arXiv:2406.09246) and read Section 3 only, the architecture section. Skip the experimental results for now, skip the related work too. For each of the six boxes in §2.3, vision encoder, language tokenizer, transformer trunk, action head, detokenizer, simulator/embodiment, find the sentence in Section 3 naming that design choice and write it down. The vision encoder is the easy one (a fused SigLIP and DINOv2); the detokenizer is trickiest, tucked into the brief discussion of per-embodiment action normalization.

The exercise is practice reading a VLA paper by box. When you reach Chapter 12, you'll do the same thing for RT-2 (Brohan et al., 2023, arXiv:2307.15818) and Octo (Octo Model Team, 2024, arXiv:2405.12213). In Chapter 13, for $\pi 0$ (Black et al., 2024, arXiv:2410.24164). Same structure every time. Doing it once now, on a model you've already run, makes the next three passes much faster.

Exercise 2.x.4 — One sentence per failure

Close the laptop. On paper or in a text file, write three sentences, each following the form "If the robot is doing X, the most likely cause is Y, and the diagnostic is Z." The three X's are the symptom descriptions from §2.4: moves in slow random scribbles near the start pose, moves decisively in the wrong direction, slams the arm toward joint limits. Write all three from memory without checking §2.4, and you've internalized the silent-failure taxonomy. Can't? Re-read §2.4 once and try again the next day. The three-sentence form stays deliberately compact; it's the version you'll want in your head debugging a real fine-tune at 11pm with no time to grep.

Chapter 2 reading list

The works below are the ones cited across §2.1 through §2.6, grouped by purpose. The full bibliography for the whole book lives in Appendix E.2; this list is just the chapter-local subset.

The model and its lineage

- Kim, M. J., Pertsch, K., Karamcheti, S., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246. The model you ran end to end. Read Section 3 for the architecture and Section 5 for the LIBERO numbers.
- Brohan, A., Brown, N., Carbajal, J., et al. (2022). "RT-1: Robotics Transformer for Real-World Control at Scale." arXiv:2212.06817. The paper that introduced the action-tokenization-into-a-transformer recipe OpenVLA inherits.
- Brohan, A., Brown, N., Carbajal, J., et al. (2023). "RT-2: Vision- Language-Action Models Transfer Web Knowledge to Robotic Control." arXiv:2307.15818. The proof that a VLM backbone could carry actions; the immediate predecessor of OpenVLA's design.
- Octo Model Team (2024). "Octo: An Open-Source Generalist Robot Policy." arXiv:2405.12213. The continuous-action sibling of OpenVLA, with a diffusion head rather than discrete tokens. Useful as a contrast.
- Black, K., Brown, N., Driess, D., et al. (2024). "$\pi 0$: A Vision- Language-Action Flow Model for General Robot Control." arXiv:2410.24164. The flow-matching successor; revisited in Chapter 13.
- Pertsch, K., et al. (2025). "FAST: Efficient Action Tokenization for Vision-Language-Action Models." arXiv:2501.09747. The follow-up tokenization scheme that addresses the discreteness ceiling raised in §2.4.
- NVIDIA (2025). "GR00T N1: An Open Foundation Model for Generalist Humanoid Robots." arXiv:2503.14734. The dual-system humanoid VLA that Chapter 14 returns to.

The data and the simulator

- O'Neill, A., Rehman, A., Maddukuri, A., et al. (2023). "Open X-Embodiment: Robotic Learning Datasets and RT-X Models." arXiv:2310.08864. The dataset that underwrites OpenVLA pretraining and the source of the `bridge_orig` `unnorm_key` from §2.4. Chapter 12 covers it in depth.
- Liu, B., Zhu, Y., Gokmen, C., et al. (2023). "LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning." arXiv:2306.03310. The simulator you ran. Read the task-suite breakdown (`libero_object`, `libero_spatial`, `libero_goal`, `libero_10`) to understand what Chapter 15 will measure.

Field surveys (for breadth)

- Sapkota, R., et al. (2025). "Vision-Language-Action Models: Concepts, Progress, Applications and Challenges." arXiv:2505.04769. The survey you were asked to mark up in Exercise 1.x.3. By now several more of its sentences should make sense.
- "A Survey on Pure Vision-Language-Action Models." arXiv:2509.19012.
- "Efficient Vision-Language-Action Models: A Survey." arXiv:2510.24795.
- "Anatomy of a VLA: Modules, Milestones, Challenges." arXiv:2512.11362.

Background you may want nearby

- Touvron, H., et al. (2023). "Llama 2: Open Foundation and Fine-Tuned Chat Models." The LLM whose vocabulary OpenVLA repurposes for action tokens; the bottom-256-entries trick from §2.1 lives or dies on this.
- Zhai, X., et al. (2023). "Sigmoid Loss for Language Image Pre-Training." (SigLIP.) One half of OpenVLA's vision tower.
- Oquab, M., et al. (2023). "DINOv2: Learning Robust Visual Features without Supervision." arXiv:2304.07193. The other half.
- Todorov, E., Erez, T., & Tassa, Y. (2012). "MuJoCo: A Physics Engine for Model-Based Control." IROS 2012. The simulator underneath LIBERO.

Chapter summary

Chapter 2 stood up a complete VLA on a laptop and ran it. You watched it succeed, watched it fail in three named ways, and now have three reproduced drills, a fifty-episode sweep, and a one-page architectural read of the OpenVLA paper sitting in a directory on your disk. With that artifact in hand, you can locate the six boxes inside a `predict_action` call you've never seen before, predict the cause of a silent failure from its symptom, estimate the success-rate variance of a VLA on a finite-episode evaluation, and place any new VLA paper into the design-choice grid the rest of Part 4 uses. Chapter 3 returns to the math the rest of the book assumes, linear algebra, probability, a fifty-line PyTorch training loop, before Chapter 4 begins the lineage that produced the model you just ran.

Chapter 3 — Math and ML prerequisites in 30 minutes

3.1 Vectors, matrices, gradients, and why the chain rule rules robotics

Chapter 2 ended with a 7-vector arriving at a simulator every 50 ms. That 7-vector is where the entire stack, a 7-billion-parameter transformer, a vision encoder pretrained on internet images, a tokenizer trained on a hundred thousand teleoperation episodes, finally touches the world. Everything upstream of it is linear algebra and the chain rule, and this chapter's job is to make sure that sentence doesn't feel like a slogan. We won't re-teach linear algebra from scratch; Appendix A is there for a full refresher. What this section does instead is pick out four objects you'll see on almost every page of the rest of the book, vectors, matrices, gradients, and the chain rule, and show them doing the specific work they do inside a robot policy.

If you can already write the Jacobian of a 2-link arm without looking it up, skim this section and move to §3.2. If the phrase "gradient of the loss with respect to the weights" still feels like an incantation, sit with this one.

Vectors and matrices: the right datatype for a robot

A robot's instantaneous state is a vector. Not metaphorically, literally a list of numbers your program holds in memory. For a 7-DoF Franka arm, the joint configuration $q \in \mathbb{R}^7$ stacks the seven joint angles. The joint velocity $\dot{q} \in \mathbb{R}^7$ is another 7-vector. The end-effector pose, depending on representation, is a 6-vector (3 for position, 3 for axis-angle orientation) or a 7-vector (position plus a unit quaternion). The action OpenVLA emitted in Chapter 2 was itself a 7-vector: three for end-effector translation deltas, three for rotation deltas, one for the gripper. Calling these "states" or "poses" or "actions" hides their common type signature. They're all points in $\mathbb{R}^n$, with n small.

Matrices show up the moment one vector has to become another. Forward kinematics is a function $f : \mathbb{R}^7 \to \mathbb{R}^6$ taking joint angles to end-effector pose. Differentiate it and you get the manipulator Jacobian $J(q) \in \mathbb{R}^{6\times 7}$, the matrix mapping joint velocities to end-effector velocities:

$$\dot{x} = J(q)\,\dot{q}.$$

Inverse-kinematics solvers, impedance controllers, and damped-least-squares trackers all live inside that one equation. If you've ever wondered why classical roboticists obsess over the singular values of $J(q)$, the answer is that the matrix sometimes goes near-singular, and inverting it then produces wildly large joint commands. We'll return to Jacobians in Chapter 4, where they're the load-bearing object in classical control, and again in Chapter 13, where $\pi 0$'s flow-matching head produces continuous actions that an outer loop still has to map through a Jacobian to reach the joints (Black et al., 2024, arXiv:2410.24164).

Images are matrices too, or more precisely third-order tensors of shape $H \times W \times 3$. The OpenVLA visual encoder from §2.3 took a $224 \times 224 \times 3$ tensor, ran it through a stack of convolutions and attention layers, and emitted a sequence of 256 512-dimensional vectors. Every "embedding," "feature map," and "representation" you'll read about for the next fifteen chapters is some tensor, and every learned layer is a function turning one tensor into another. Only three operations account for the bulk of the work: matrix-vector multiplication, elementwise nonlinearity, and reduction

(sum, mean, max). A transformer block is a particular arrangement of those three. A convolution is a structured matrix multiplication with weight sharing. A normalization layer is a reduction followed by an elementwise rescale. Once you see the underlying datatype, architecture diagrams stop looking like a set of incompatible boxes.

One pragmatic warning: robotics conventions are a minefield. The same 6-vector can mean position-then-rotation or rotation-then-position. The same rotation can be a quaternion (with w first, last, or absent), an axis-angle vector, or three Euler angles in any of twelve orderings. RT-1 (Brohan et al., 2022, arXiv:2212.06817), OpenVLA (Kim et al., 2024, arXiv:2406.09246), and $\pi 0$ each pick a convention, document it somewhere in the codebase, and silently break if you feed them the wrong layout. When a learned policy outputs garbage, "the rotation convention is flipped" sits in the top three culprits more often than it has any right to. Read the data-loader, not the paper.

Gradients are linear approximations

A function $L : \mathbb{R}^n \to \mathbb{R}$, say the loss of a policy network as a function of its weights, has a gradient $\nabla L \in \mathbb{R}^n$. The gradient isn't "the slope" in any single direction. It's the vector that, dotted with any small step Δw, predicts the change in L:

$$L(w + \Delta w) \approx L(w) + \nabla L(w)^\top \Delta w.$$

That's the whole idea. Every line about "moving downhill," every diagram with a loss landscape, every optimizer trick is a corollary of that one linear approximation. The gradient is the best linear approximation in the sense that it's the only vector for which the equation above holds correct to first order, and it points toward steepest ascent because a dot product with a unit vector is largest when the two are aligned. Gradient descent, taking $w \leftarrow w - \eta \nabla L(w)$, is the obvious move once you have it.

The catch is that policies carry a lot of weights. OpenVLA-7B has, approximately, seven billion of them. The gradient is a 7-billion-entry vector, the same shape as the parameter vector itself. You won't compute it by naively perturbing each weight in turn; that would take roughly seven billion forward passes per training step. Deep learning is practical at all because there's an algorithm that computes the full gradient in about the time of two forward passes. That algorithm is backpropagation, and backpropagation is just the chain rule applied carefully.

A worked toy example helps here. Suppose the policy is a single linear layer followed by a mean-squared-error loss: $L(W) = \frac{1}{2}\|Wx - a^\star\|^2$, where $x \in \mathbb{R}^d$ is an observation, $a^\star \in \mathbb{R}^k$ is the target action, and $W \in \mathbb{R}^{k \times d}$ is the weight. Differentiating gives $\nabla_W L = (Wx - a^\star)x^\top$, a $k \times d$ matrix, exactly the shape of W, which is the first sanity check worth running on every gradient derivation you write. The outer-product structure is informative too: the gradient with respect to row i of W is the i-th component of the prediction error scaled by the input x. Wrong dimension of the output? Adjust the row that produced it, in proportion to the input that fed it. That intuition holds for a linear regressor, and it's the same template that keeps working for transformer attention heads. They just stack many such updates on top of each other.

The chain rule, and why it rules robotics

Now stack the layers. Let $a = g(h)$ and $h = f(x; W)$, with the same L as before. To update W, you need $\partial L / \partial W$. The chain rule says:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial a} \cdot \frac{\partial a}{\partial h} \cdot \frac{\partial h}{\partial W}.$$

Each factor is a Jacobian. The product is, in general, a tensor, but the practical magic is that you never materialize the full Jacobians. You materialize the vector-Jacobian product at each layer, walking backwards from L to the inputs. That's what `loss.backward()` does under the hood in PyTorch, and it's what keeps the asymptotic cost of a gradient at about twice the cost of a forward pass instead of seven billion times that cost. We'll write that loop out by hand in §3.3.

The phrase "chain rule rules robotics" isn't a pun. It's a claim about which equation does the most work in modern robot learning. Pick any modern action model and trace the loss back to the parameters it updates. In RT-1

(arXiv:2212.06817), the discrete-action cross-entropy loss flows back through a transformer trunk, a token embedding, a FiLM-conditioned CNN, and into the EfficientNet visual encoder; every arrow in that data-flow diagram is one factor of one chain-rule product. In π0, the flow-matching loss on a continuous 7-vector flows back through an action-expert transformer, a language-conditioned VLM trunk, and a tokenized image encoder. Same structure, different blocks. Even classical methods we'll meet in Chapter 4, iterative inverse-kinematics solvers, computed-torque controllers, trajectory optimizers, are walking the chain rule through the kinematic and dynamic equations of the robot. The objects differ (an explicit Jacobian instead of an autodiff graph), but the operation stays the same: compose local linearizations and read off the global one.

The chain rule has one further consequence worth flagging now, since it comes back to haunt later chapters. If any factor in the product is very small, the whole product shrinks toward zero, which is the vanishing gradient problem, why ReLU replaced sigmoid as the default activation, and why transformer trunks add residual connections letting the gradient bypass deep stacks. If any factor is very large, the whole product blows up, the exploding gradient problem, why gradient clipping exists, and why training a 7B-parameter VLA without mixed-precision care often produces NaNs by step 200. The fixes are mostly engineering. The underlying diagnosis is always the chain rule multiplying things it shouldn't.

You now have the vocabulary: a vector is a state or an action, a matrix maps one to another, a gradient is the linear approximation that lets optimization make progress, and the chain rule composes those linear approximations across the entire stack from pixels to motor torques. That's the substrate. The next section adds the second pillar, randomness, that turns this substrate into a probabilistic model of a robot interacting with an uncertain world.

3.2 Random variables, expectations, KL divergence

The previous section described a robot policy as a function turning inputs into outputs, a vector in, a vector out. That picture is incomplete. The action OpenVLA emitted in Chapter 2 wasn't a deterministic response to the image; it was the mode of a distribution over 256 discrete bins. The policy learned not "given this image, move the gripper 4.2 mm right" but "given this image, the probability distribution over move-right distances peaks near bin 143." That shift, from functions to distributions, is what this section covers. If §3.1 introduced the spatial language of robot learning, this one introduces its probabilistic language.

Random variables and distributions

A random variable X is a quantity whose value isn't fixed in advance; you can only describe how likely different values are. For a discrete random variable, one whose outcomes come from a finite or countable set $\mathcal{X}$, the distribution is a probability mass function $p(x) = P(X = x)$, satisfying $p(x) \geq 0$ for all x and $\sum_{x \in \mathcal{X}} p(x) = 1$.

For a continuous random variable, one that can take any value in an interval or in $\mathbb{R}^n$, the distribution is a probability density function $p(x)$ satisfying $p(x) \geq 0$ and $\int p(x)\, dx = 1$. A density doesn't assign probability to a single point. Instead, $\int_A p(x)\, dx$ gives the probability that X lands somewhere in the set A.

The distinction is load-bearing here. RT-1 (Brohan et al., 2022, arXiv:2212.06817) and OpenVLA (Kim et al., 2024, arXiv:2406.09246) discretize each action dimension into 256 bins, so the predicted action is a product of 7 discrete distributions, one PMF per joint, and the loss becomes a sum of 7 cross-entropies, each comparing a PMF to a one-hot target. π0 (Black et al., 2024, arXiv:2410.24164), by contrast, treats the action as a continuous random variable in $\mathbb{R}^7$ and learns a flow from Gaussian noise to the target action distribution; its loss never touches a PMF at all. Discrete versus continuous isn't an aesthetic choice. It determines which math you write and which pathologies you end up fighting, a tension we return to in Chapter 10.

A Gaussian (normal) distribution over $\mathbb{R}^n$ with mean μ and covariance Σ is:

$$\mathcal{N}(x;\, \mu, \Sigma) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\!\left(-\tfrac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right).$$

The exponent runs negative and large when x sits far from μ in the metric defined by Σ^{-1}, so high probability mass clusters near the mean. Gaussians are the default prior for continuous action spaces because they're closed under

affine transformations, have closed-form entropy, and because minimizing the negative log-likelihood of a Gaussian target with fixed diagonal covariance turns out to be exactly minimizing mean-squared error, the loss you'd write on intuition alone for a regression problem. When Octo (Ghosh et al., 2024, arXiv:2405.12213) conditions a diffusion head on task tokens, the diffusion starts from $\mathcal{N}(0, I)$ and learns to transport that noise toward the target action region; every step of that transport is a perturbation of a Gaussian. Know the Gaussian, and you understand the starting point of most continuous generative action models.

Expectation

The expectation of a function f under a distribution p is the probability-weighted average of f's values:

$$\mathbb{E}_{x\sim p}[f(x)] = \begin{cases} \displaystyle\sum_{x\in\mathcal{X}} p(x)\, f(x) & \text{discrete} \\ \displaystyle\int p(x)\, f(x)\, dx & \text{continuous.} \end{cases}$$

In the discrete case, if p is the distribution over action bins OpenVLA predicts and $f(x)$ is the cost of taking action x, then $\mathbb{E}[f(X)]$ is the expected cost of sampling from the policy's output. In the continuous case, if p is the stationary distribution induced by some policy interacting with the environment and $f(x) = r(x)$ is a reward function, then $\mathbb{E}_{x\sim p}[r(x)]$ is the average reward, exactly the quantity reinforcement learning maximizes (Chapter 5 formalizes this with Markov decision processes).

Three properties of expectation are worth memorizing, since derivations lean on them constantly.

Linearity: $\mathbb{E}[af(X) + bg(X)] = a\,\mathbb{E}[f(X)] + b\,\mathbb{E}[g(X)]$. This looks obvious, but it means constant factors move in and out of expectation freely, which lets you rearrange loss expressions however you need.

Law of total expectation: $\mathbb{E}_x[f(x)] = \mathbb{E}_y[\mathbb{E}_{x|y}[f(x)]]$. Condition on some intermediate variable y, compute the inner expectation over x given y, then average out y. In robotics, y might be the latent state in a world model (Chapter 9), and this identity is how you justify marginalizing over it.

Monte Carlo estimation: $\mathbb{E}_{x\sim p}[f(x)] \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i)$ for $x_i \overset{\text{iid}}{\sim} p$. The law of large numbers guarantees convergence. In practice, $N = 1$ during training; you sample a single trajectory and use its reward as an unbiased estimator of the expected reward. The high variance of that estimate is the main reason policy-gradient training feels noisy, and the variance-reduction tricks in Chapter 7 (baselines, GAE, advantage normalization) are all interventions on this exact Monte Carlo estimator.

Entropy and cross-entropy

The entropy of a distribution p measures how spread out it is:

$$H(p) = -\mathbb{E}_{x\sim p}[\log p(x)] = -\sum_{x} p(x) \log p(x).$$

A one-hot distribution (all mass at one outcome) has $H = 0$. A uniform distribution over k outcomes has $H = \log k$. In action distributions, low entropy means the policy is confident (one action dominates); high entropy means the policy is uncertain, or deliberately exploratory.

Cross-entropy between a target distribution p and a predicted distribution q is:

$$H(p, q) = -\mathbb{E}_{x\sim p}[\log q(x)] = -\sum_{x} p(x) \log q(x).$$

When p is the one-hot distribution over the correct action token, as it is in every discrete-action VLA trained on demonstrations, $H(p, q)$ reduces to $-\log q(a^\star)$: the negative log-probability assigned to the correct action. That's

the cross-entropy loss you saw applied to each of the 7 action dimensions in §2.3. Minimizing it pushes q to assign high probability to the demonstrated action, and this loss dominates the supervised phase of nearly every language-conditioned imitation model from RT-1 (arXiv:2212.06817) through RT-2 (arXiv:2307.15818) through OpenVLA (arXiv:2406.09246).

One subtlety worth flagging: cross-entropy isn't symmetric. $H(p, q) \neq H(q, p)$ in general. The convention starts to matter once you move beyond supervised imitation, for instance when a VLA needs to cover all high-probability actions of a multimodal demonstration distribution rather than just fit one mode. Chapter 6 revisits that asymmetry when discussing why behavior cloning struggles with multimodal data.

KL divergence

The Kullback-Leibler divergence from q to p (read: how different q is from p) is:

$$D_{\mathrm{KL}}(p \,\|\, q) = \mathbb{E}_{x \sim p}\left[\log \frac{p(x)}{q(x)}\right] = H(p, q) - H(p).$$

That last equality comes from expanding the expectation: $\mathbb{E}_p[\log p - \log q] = -H(p) + H(p, q)$, rearranged. Because $D_{\mathrm{KL}} \geq 0$ always (by Jensen's inequality, with equality iff $p = q$), minimizing $H(p, q)$ over q is the same as minimizing $D_{\mathrm{KL}}(p \,\|\, q)$ whenever the target p doesn't depend on q, since $H(p)$ is then just a constant. The cross-entropy loss is the KL divergence, shifted by the entropy of the target. Knowing this is more than trivia. It tells you what you're actually doing when you train with cross-entropy: driving the predicted distribution as close as possible to the demonstrated distribution, in the specific asymmetric sense that $D_{\mathrm{KL}}(p \,\|\, q)$ penalizes placing zero probability mass where the target p has nonzero mass. If the teacher only ever demonstrates one action variant, p carries low entropy, and the penalty structure of $D_{\mathrm{KL}}(p \,\|\, q)$ will happily ignore modes of the true (unknown) distribution the teacher never exhibited. Compounding error in behavior cloning, covered in Chapter 6, follows directly from this.

The reverse KL, $D_{\mathrm{KL}}(q \,\|\, p)$, penalizes placing mass where the target p is zero, which produces mode-seeking behavior instead. Variational inference and some policy-optimization methods use the reverse KL; supervised imitation typically uses the forward KL. Which direction you pick is an architectural choice that shapes the failure modes of the resulting policy.

KL divergence in world models and latent variable models

KL divergence appears explicitly in the training objective of any model with a stochastic latent variable. The standard variational bound (ELBO) for a latent variable model with observations o, latent state z, and parameters θ is:

$$\log p_\theta(o) \geq \underbrace{\mathbb{E}_{z \sim q_\phi(z|o)}[\log p_\theta(o \mid z)]}_{\text{reconstruction}} - \underbrace{D_{\mathrm{KL}}\big(q_\phi(z \mid o) \,\|\, p_\theta(z)\big)}_{\text{regularization}}.$$

The first term rewards the model for explaining the observation using the learned latent. The second regularizes the posterior $q_\phi(z \mid o)$ toward the prior $p_\theta(z)$, keeping the latent from memorizing and ignoring structure. World models such as RSSM (used in DreamerV3, covered in Chapter 9) minimize a variant of this bound; the KL term is what keeps the learned dynamics smooth enough to plan in. Set the KL coefficient to zero and the latent space collapses into a lookup table, losing its generalization ability. Set it too high, and the posterior gets dragged too close to a unit Gaussian, degrading reconstruction. Tuning that balance, the β in β-VAE variants, is a core engineering concern in latent world models.

A concrete worked example: the OpenVLA action head

Putting this together concretely: OpenVLA (arXiv:2406.09246) outputs 7 consecutive token positions, each decoded from a 32,000-token vocabulary. Only 256 of those tokens actually get used as action bins in practice; the rest stay masked. The model's output at position t is a vector of 32,000 logits, which a softmax converts into a PMF $q(\cdot)$ over the vocabulary. The training target is the demonstration action $a_t^\star$, encoded as a bin index, which is a one-hot p. The loss for that position is:

$$\ell_t = -\log q(a_t^\star).$$

This is $H(p, q)$ for a one-hot p, which equals $D_{\mathrm{KL}}(p \,\|\, q) + H(p) = D_{\mathrm{KL}}(p \,\|\, q)$ (since $H(\text{one-hot}) = 0$). The total action loss is $\frac{1}{7}\sum_{t=1}^{7} \ell_t$, averaged over the 7 dimensions and over the minibatch.

Now consider what happens when the demonstrator is ambiguous, say the task "place the cup on the tray" could be completed by moving left first or right first. The demonstration dataset ends up with both trajectories. The one-hot targets for those trajectories land in different bins on the first step. The model, minimizing forward KL, tries to assign nonzero probability to both bins at once. For a unimodal softmax output, the result is a compromise distribution spread across two modes, the classic mode-averaging failure. This is a consequence of the KL's direction, not a bug in the architecture, and that observation motivates much of Chapter 10's focus on diffusion and flow heads, which can represent multimodal action distributions explicitly instead of averaging over them.

What you need to carry forward

Three facts do most of the work in the remaining chapters.

First, cross-entropy loss equals KL divergence when the target is fixed. Every supervised VLA trains by minimizing a KL from the demonstration distribution to the model's output distribution, whether or not the paper frames it that way.

Second, expectations appear everywhere: expected reward in RL, expected loss over a minibatch, expected reconstruction in the ELBO. The Monte Carlo estimator ($N = 1$) is almost always what gets used in practice, and its variance is almost always the bottleneck.

Third, the direction of KL divergence is a design decision with downstream consequences. Forward KL (used in supervised imitation) is mode-averaging. Reverse KL (used in some RL variants) is mode-seeking. Choosing between them means choosing between different failure modes.

The next section puts these tools to work. We'll write a complete 50-line training loop in PyTorch, step through the cross-entropy and MSE losses in code, and watch the gradient flow backward through a small policy network from loss to weight update.

3.3 A 50-line PyTorch training loop, annotated

The last two sections established the math: the chain rule computes gradients, cross-entropy is the operational form of KL divergence for discrete targets, and minimizing the expected loss over a dataset of demonstrations is what behavior cloning actually does. This section makes all of that executable. We'll build the smallest possible policy network, write its training loop, and annotate every line doing non-obvious work. The goal isn't a toy you should ship. It's a template you should be able to recognize in any VLA codebase.

The policy here is a three-layer MLP with a separate linear head per action dimension, producing 256-bin logits exactly like the OpenVLA action head described in §2.3 and §3.2. Inputs are 64-dimensional observation embeddings; outputs are 7 per-joint discrete distributions over action bins. This isn't a vision model, not a transformer, not a sequence model. It's simple enough to fit on one screen and complex enough to exercise every part of the gradient path you need to understand.

The model

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader, TensorDataset

class SmallPolicy(nn.Module):
    def __init__(self, obs_dim=64, hidden=256, n_bins=256, n_joints=7):
        super().__init__()
```

```
        self.trunk = nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.ReLU(),
            nn.Linear(hidden, hidden), nn.ReLU(),
        )
        self.heads = nn.ModuleList(
            [nn.Linear(hidden, n_bins) for _ in range(n_joints)]
        )

    def forward(self, obs):           # obs: (B, obs_dim)
        h = self.trunk(obs)           # h: (B, hidden)
        return [head(h) for head in self.heads]   # list of 7 × (B, n_bins)
```

Two architectural decisions are worth pausing on. First, the trunk is shared across all 7 joints. Every action dimension sees the same representation, which is correct, since the decision about how far to move joint 4 depends on the same observation state as the decision about joint 6. Second, the heads are separate linear layers rather than a single linear layer with 7×256 outputs. The difference is purely organizational; they're mathematically equivalent. But the `ModuleList` makes it easy later to add per-head losses, per-head regularizers, or per-head frozen weights.

The output is a list of logits, not probabilities. PyTorch's `CrossEntropyLoss` applies the softmax internally, which is numerically more stable than calling `F.softmax` first and then `F.nll_loss`. Don't apply softmax before the loss function; you'll get the right answer, just with worse numerical conditioning.

The training loop

```
def train(n_epochs=50, seed=42):
    torch.manual_seed(seed)
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    policy    = SmallPolicy().to(device)
    optimizer = torch.optim.AdamW(
        policy.parameters(), lr=1e-4, weight_decay=1e-5
    )
    loss_fn = nn.CrossEntropyLoss()

    # synthetic dataset: 4096 (obs, action-bin-vector) pairs
    obs  = torch.randn(4096, 64)
    acts = torch.randint(0, 256, (4096, 7))   # one bin index per joint
    loader = DataLoader(
        TensorDataset(obs, acts), batch_size=64, shuffle=True
    )

    for epoch in range(n_epochs):
        epoch_loss = 0.0

        for obs_b, acts_b in loader:
            obs_b  = obs_b.to(device)
            acts_b = acts_b.to(device)

            optimizer.zero_grad()                       # (1)

            logits = policy(obs_b)                      # (2)

            loss = sum(
                loss_fn(logits[j], acts_b[:, j])
                for j in range(7)
```

```
            ) / 7.0                                     # (3)

            loss.backward()                             # (4)
            torch.nn.utils.clip_grad_norm_(             # (5)
                policy.parameters(), max_norm=1.0
            )
            optimizer.step()                            # (6)

            epoch_loss += loss.item()

        if (epoch + 1) % 10 == 0:
            avg = epoch_loss / len(loader)
            print(f"epoch {epoch + 1:3d}  loss {avg:.4f}")

    return policy
```

Six lines are labelled above. Here's what each one actually does.

(1) `optimizer.zero_grad()`

PyTorch accumulates gradients by default. Every call to `loss.backward()` adds to the `.grad` attribute of each parameter rather than replacing it. Forget `zero_grad()` and you get gradient accumulation across batches, which produces the illusion of a larger effective batch size but is almost never what you actually want inside a standard training loop. Call `zero_grad()` before every forward pass. The exception, gradient accumulation as a deliberate technique for simulating large-batch training under limited memory, gets covered in Chapter 16, where we fine-tune a 7B-parameter VLA under a 40 GB VRAM ceiling. Same mechanism; different intent.

(2) `logits = policy(obs_b)`

This is the forward pass: data flows from the input tensor through the trunk, through each head, and returns a list of logit tensors. PyTorch's autograd engine watches the whole thing. Every operation touching a tensor with `requires_grad=True` (which all model parameters carry by default) gets recorded into a computation graph. After this line, PyTorch knows the complete sequence of operations that produced `logits` from `policy.parameters()`, and the backward pass will traverse that graph in reverse.

(3) Loss computation

```
loss = sum(
    loss_fn(logits[j], acts_b[:, j])
    for j in range(7)
) / 7.0
```

`loss_fn(logits[j], acts_b[:, j])` computes the cross-entropy for joint j: it softmaxes the logits into a predicted PMF q_j, then computes $-\log q_j(a_j^\star)$ for each item in the batch and averages. Summing over 7 joints and dividing by 7 gives the mean per-joint cross-entropy, numerically identical to what OpenVLA computes on its 7 action-token positions (arXiv:2406.09246), except OpenVLA uses a 32,000-token vocabulary masked to 256 action bins rather than a native 256-class softmax. The gradient structure is the same either way.

Note that the sum over 7 joints is part of the computation graph. `loss` is a scalar tensor, not a Python float. Calling `.item()` to log it detaches it from the graph, so don't call `.item()` before `.backward()`.

(4) `loss.backward()`

This is the chain rule, executed. PyTorch walks the computation graph from the scalar loss back to every parameter, computing vector-Jacobian products at each node. After this line, every parameter in `policy` has its `.grad` attribute populated with $\nabla_\theta \mathcal{L}$, a tensor the same shape as the parameter itself. This is the n-dimensional vector described in

§3.1: for `SmallPolicy`, it runs on the order of 64×256 + 256 + 256×256 + 256 + 7×(256×256 + 256) ≈ 490,000 entries. Computing all of them costs roughly twice the forward pass.

Since the loss here is a Python sum of `CrossEntropyLoss` values, backprop automatically distributes through the sum via linearity. Each head's loss contributes a gradient to `self.heads[j]` parameters and also to `trunk` parameters, because the trunk output feeds into every head. The trunk's gradient is the sum of 7 head gradients, all propagated through the shared representation. This is a structural fact worth internalizing: training multiple heads on the same trunk isn't the same as training 7 independent networks. They regularize each other through the shared parameters.

(5) Gradient clipping

```
torch.nn.utils.clip_grad_norm_(policy.parameters(), max_norm=1.0)
```

Before the optimizer updates the weights, this line rescales the gradient vector if its L2 norm exceeds 1.0. If $\|\nabla L\|_2 > 1$, the gradient gets replaced by $\nabla L / \|\nabla L\|_2$; otherwise it's left alone. This prevents the exploding-gradient failure mode from §3.1, where a single unusually large loss value produces a gradient spike that shoves weights far from a good region. Clipping at 1.0 is a commonly safe default; VLA fine-tuning recipes (Chapter 16) sometimes lower it to 0.5 for the first few thousand steps. The cost is one extra pass over all parameters to compute the norm, which is negligible.

Note the placement carefully: clipping must happen after `.backward()` (since the `.grad` attributes need to exist) and before `.step()` (otherwise you're clipping a stale gradient the optimizer has already used). This ordering is the source of a surprisingly common training-loop bug: clipping after `step()` does nothing at all.

(6) `optimizer.step()`

This applies the AdamW update rule:

$$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2,$$

$$\theta_t \leftarrow \theta_{t-1} - \eta \cdot \frac{m_t / (1-\beta_1^t)}{\sqrt{v_t/(1-\beta_2^t)} + \epsilon} - \eta \lambda \theta_{t-1}.$$

The first two equations maintain a running mean (m_t) and running variance (v_t) of the gradient. Dividing by the square root of the variance is Adam's "adaptive" part: dimensions with high-variance gradients get smaller effective learning rates, and dimensions with low-variance gradients get larger ones. The defaults $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ work for the overwhelming majority of robot learning training runs.

The last term, $-\eta\lambda\theta_{t-1}$, is the AdamW weight decay. It directly decays the parameter toward zero, independently of the gradient. This differs subtly from L2 regularization added to the loss (which couples the regularizer through the adaptive denominator instead). For fine-tuning large pretrained models like OpenVLA, AdamW's decoupled decay is the right choice; plain Adam with L2-regularized loss can under-regularize large-magnitude parameters.

Reading the diagnostics

After 50 epochs on random data, the loss should plateau around $\log(256) \approx 5.54$, the entropy of a uniform distribution over 256 bins. That's the floor for a model that has learned nothing, since random action-bin targets are uniformly distributed and the best a model can do is predict uniform logits. Loss dropping below $\log(256)$ means the model found structure in the synthetic data, almost certainly overfitting to random correlations. Loss staying well above $\log(256)$ usually means the learning rate is too small, or gradients are being clipped down to nothing.

Two diagnostic quantities beyond the loss are worth logging from the start of any training run. First, the gradient norm before clipping: sitting consistently at the clip threshold means you're cutting information every step. Second, the weight update magnitude. `optimizer.step()` changes the parameters by some amount, and the ratio of that amount to the parameter norm (sometimes called the update ratio or param SNR) should stay in the range 10^{-3} to 10^{-2} for stable training. A ratio above 10^{-2} signals the effective learning rate is too high; below 10^{-3} signals the

model isn't learning at all. Neither PyTorch nor AdamW provides these quantities automatically, so log them with a few extra lines before `optimizer.step()`, using `torch.nn.utils.clip_grad_norm_`'s return value (the unclipped norm) plus a parameter-snapshot comparison.

Three things to change for a real task

The loop above is a template, not a production recipe. Three changes get you from random synthetic data to actual robot demonstrations.

Replace the synthetic dataset. Real behavior cloning loads `(observation, action)` pairs from a teleoperation log. In practice this means a DataLoader over a dictionary of tensors, often loaded from HDF5 or RLDS format. The shapes change, observations might be $(B, 224, 224, 3)$ RGB images rather than $(B, 64)$ vectors, but the loop structure stays the same.

Replace the loss for continuous actions. Switch from discrete-bin actions to raw continuous vectors, as $\pi 0$ does (arXiv:2410.24164), and the per-joint cross-entropy becomes per-joint mean-squared error, a flow-matching loss, or a diffusion denoising loss. The backward pass and optimizer step stay identical; only `loss_fn` changes. Chapter 10 covers each of these alternatives in detail.

Add a validation loop. One loop over training data measures whether the model is fitting. A second loop over held-out episodes, without calling `optimizer.step()`, measures whether it's generalizing. Training loss dropping while validation loss doesn't means the model is memorizing demonstrations rather than learning a transferable policy. Chapter 15 discusses how to construct evaluation splits that are actually informative in the robot setting, where collecting genuinely out-of-distribution validation data is far from trivial.

With these three modifications, the 50-line loop above is functionally the core of the training pipelines used in RT-1 (arXiv:2212.06817), OpenVLA (arXiv:2406.09246), and Octo (arXiv:2405.12213). The transformer architectures differ, the datasets differ, the hardware scale differs by orders of magnitude, but `zero_grad`, forward pass, loss, `backward`, clip, `step` is the sequence running through each of them. The next section extends the vocabulary one step further: rather than a single supervised loss on demonstrated actions, we examine the three loss families, supervised, reinforcement, and self-supervised, that the rest of the book draws from, and pin down exactly when each one is the right tool.

3.4 Three loss families: supervised, RL, self-supervised

The training loop in §3.3 assumed a specific loss: cross-entropy on discretized action bins, computed against human-demonstrated actions. That assumption holds for RT-1 (arXiv:2212.06817) and OpenVLA (arXiv:2406.09246), but it isn't the only game in town. Every action model in this book trains by minimizing some scalar loss function, and a model's properties get shaped not just by its architecture but by which loss family that function belongs to.

Three families cover essentially everything in the current literature. Supervised loss: you have labeled data, you know the right answer, and you penalize prediction errors directly. Reinforcement learning loss: you don't have labeled actions, but you have a reward signal, and you update the policy to increase expected future reward. Self-supervised loss: there are no human labels at all, and the training signal gets manufactured from the structure of the data itself, typically by corrupting something and asking the model to reconstruct it. Understanding these three families, their trade-offs, and which conditions favor each one is a prerequisite for reading any modern VLA paper.

Supervised loss

A supervised loss measures the discrepancy between a model's prediction and a target provided by a human or a trusted oracle. In robot learning, that oracle is typically a teleoperator who demonstrated the task.

Discrete targets, cross-entropy. When actions get discretized into bins, as in RT-1, OpenVLA, and the SmallPolicy from §3.3, the loss is cross-entropy between the model's predicted probability distribution and the one-hot target:

$$\mathcal{L}_{\text{CE}}(\theta) = -\frac{1}{N}\sum_{i=1}^{N} \log q_\theta(a_i^* \mid o_i)$$

where a_i^* is the demonstrated action bin and q_θ is the model's predicted distribution. This is identical to the formula in §3.2; we're minimizing the average negative log-probability of the demonstrated actions under the model's distribution. The discrete-bin vocabulary turns this into a standard classification problem: one class per bin per action dimension, cross-entropy aggregated across dimensions.

Continuous targets, regression loss. Some models output raw continuous vectors rather than bin indices. The simplest loss then is mean-squared error:

$$\mathcal{L}_{\text{MSE}}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \|f_\theta(o_i) - a_i^*\|^2$$

MSE is differentiable everywhere, easy to implement, and corresponds to maximum likelihood under a Gaussian likelihood on the actions. Its limitation for robot learning is unimodality: if a demonstrated action is sometimes "grasp from the left" and sometimes "grasp from the right" depending on subtle visual cues, MSE averages the two and predicts a point in between, a prediction that may be physically invalid. Discrete-bin cross-entropy is also unimodal in effect (the gradient pushes all probability mass toward the single demonstrated bin), though the binning can at least represent high-entropy predictions by spreading mass across adjacent bins. Multimodal action distributions are a recurring theme in this book, and the diffusion and flow-matching losses in Chapter 10 are specifically designed to address the problem.

When to use supervised loss. The prerequisite is demonstrated action data, teleoperation, kinesthetic teaching, motion capture, or any other method producing (o, a) pairs where the action is known to be correct. Modern large datasets like Open X-Embodiment (arXiv:2310.08864) are overwhelmingly composed of this kind of data. With a few thousand demonstrations and no simulator, supervised loss is almost always the right choice. It's sample-efficient, numerically stable, and produces gradients that directly increase the log-probability of behaviors you actually want.

The weakness is dependency on demonstration quality. A supervised model can't exceed its training data; a teleoperator using a suboptimal strategy teaches the model that same strategy. Supervised loss also can't benefit from interactions the policy generates itself, since every gradient step uses human-provided targets rather than the model's own exploratory experience.

Reinforcement learning loss

A reinforcement learning (RL) loss doesn't require demonstrated actions. Instead, the policy generates trajectories by interacting with an environment, and a reward function evaluates each trajectory. The goal is finding parameters that maximize expected cumulative reward:

$$\mathcal{L}_{\text{RL}}(\theta) = -\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \gamma^t r_t\right]$$

The negative sign is a convention: minimizing the loss corresponds to maximizing reward. The discount factor $\gamma \in [0, 1)$ down-weights rewards occurring far in the future; setting γ close to 1 makes the policy care about long-horizon outcomes.

The challenge is computing the gradient of this expectation with respect to θ. Unlike supervised loss, where a_i^* is fixed and the gradient of $\log q_\theta(a_i^*|o_i)$ is straightforward, the distribution over trajectories itself depends on θ. The standard solution is the policy gradient theorem, which rewrites the gradient as:

$$\nabla_\theta \mathcal{L}_{\text{RL}} = -\mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \cdot G_t\right]$$

where $G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ is the return from step t. The gradient is an expectation over sampled trajectories, estimated in practice by rolling out the policy, collecting rewards, and treating each step as a weighted log-probability maximization. High-return steps get strong positive gradient signal; low-return steps get weak or negative signal.

The variance problem. G_t is a sum of random rewards, and its variance can run enormous, especially in long-horizon tasks where a single episode spans hundreds of steps. High-variance gradient estimates need many samples to converge. Practical algorithms like PPO (Chapter 7) reduce variance by subtracting a baseline, clipping the probability ratio between old and new policy, and using advantage estimates rather than raw returns. The mechanics get detailed in Chapter 7; the key point here is that RL losses run inherently noisier than supervised losses, which means RL training typically needs one to two orders of magnitude more compute and far more environment interactions.

When to use RL loss. RL requires a reward signal, which in turn requires either a real environment (expensive and slow), a simulator (faster, but subject to sim-to-real gap), or a learned reward model. In the current VLA landscape, pure RL from scratch for manipulation stays rare, since the exploration problem in high-dimensional continuous spaces is severe and most tasks don't yield a natural dense reward. RL shows up most prominently as a fine-tuning signal on top of supervised pretraining, the reinforcement-learning-from-human-feedback (RLHF) paradigm imported from language models and applied to robotics actions. Chapters 5 and 7 treat RL in depth; Chapter 18 discusses its emerging role in reasoning-augmented VLAs.

Self-supervised loss

A self-supervised loss generates training signal from the structure of the data itself, without external labels. The key move is defining an auxiliary task constructible automatically from unlabeled observations: corrupt the input somehow, then train the model to reconstruct what was removed.

Reconstruction and denoising. The classic form is denoising: add noise to a clean sample, then penalize the model's error in recovering the original. Formally, let x be a clean data point, $\tilde{x} = x + \epsilon$ a noisy version, and f_θ the model. The loss is:

$$\mathcal{L}_{\text{denoise}}(\theta) = \mathbb{E}_{x,\epsilon}\left[\|f_\theta(\tilde{x}) - x\|^2\right]$$

This looks like an MSE loss, but the target x isn't a human-provided label. It's a corrupted version of the model's own input. This is the loss that trains diffusion models: with a specific noise schedule and parameterization, the denoising objective becomes equivalent to learning the score function of the data distribution (the gradient of the log-density), which you can then use to generate new samples through iterative denoising. Octo (arXiv:2405.12213) uses a diffusion head on top of a transformer backbone, trained with a variant of this loss to generate smooth action trajectories from a noise-corrupted starting point.

Flow matching. A related but distinct self-supervised objective is flow matching. Instead of learning to denoise, the model learns a vector field transporting samples from a simple noise distribution p_0 (say, Gaussian) to the target action distribution p_1 (the distribution of demonstrated actions). Given a linearly interpolated path $x_t = (1-t)\epsilon + ta^*$ between a noise sample $\epsilon \sim \mathcal{N}(0, I)$ and a demonstrated action a^*, the target velocity is $(a^* - \epsilon)$ and the loss is:

$$\mathcal{L}_{\text{FM}}(\theta) = \mathbb{E}_{t,\epsilon,a^*}\left[\|v_\theta(x_t, t) - (a^* - \epsilon)\|^2\right]$$

The model v_θ learns to predict the velocity pointing from the noise toward the target. At inference, you start from Gaussian noise and integrate the learned velocity field, arriving at a sample from the action distribution after a fixed number of steps. This is exactly the action head in π0 (arXiv:2410.24164): the flow-matching loss lets the model represent multimodal action distributions without committing to a single mode, critical for dexterous manipulation where the same task can be accomplished in qualitatively different ways. We return to flow matching in Chapter 10.

Contrastive loss. A second major branch of self-supervised learning is contrastive: rather than predicting corrupted inputs, the model learns an embedding space where semantically similar pairs pull together and dissimilar pairs push apart. The InfoNCE loss, underlying CLIP, is:

$$\mathcal{L}_{\text{InfoNCE}} = -\mathbb{E}\left[\log \frac{\exp(f(x)^\top g(y^+)/\tau)}{\sum_k \exp(f(x)^\top g(y_k)/\tau)}\right]$$

where (x, y^+) is a positive pair (an image and its caption, for instance), $\{y_k\}$ includes the positive and negatives, and τ is a temperature parameter. CLIP (which underlies the visual encoder in OpenVLA) was trained with this loss on 400 million image-text pairs. The result is a visual encoder whose representations align with natural-language descriptions, precisely the property that makes zero-shot language conditioning possible in VLAs. Chapter 11 covers CLIP's role in the VLA recipe in detail.

When to use self-supervised loss. Self-supervised losses dominate as the pretraining signal whenever labeled data is scarce or nonexistent. The internet holds billions of images with captions, hours of robot video without action labels, and structured language data of every variety, and self-supervised losses can exploit all of it. The cost is indirection: a denoising or contrastive loss trains the model on a proxy task, and the quality of downstream behavior depends on how well that proxy predicts the real task. Getting this right requires careful choices about architecture, noise schedule, and what counts as a "positive" pair, choices that stay largely empirical and chapter-specific.

Comparing the three families

The table below summarizes the practical decision factors:

Factor	Supervised	RL	Self-supervised
Requires labeled actions	Yes	No	No
Requires reward function	No	Yes	No
Requires environment interaction	No	Yes (extensively)	No
Handles multimodal actions	Poorly (MSE) / Partially (CE)	Yes	Yes (diffusion, flow)
Sample efficiency	High	Low	Medium
Training stability	High	Low	Medium
Common use in VLAs	Fine-tuning, action heads	Post-training RLHF	Pretraining, diffusion/flow heads

In practice, the largest action models combine all three. A typical pipeline runs like this: a vision-language backbone gets pretrained on internet-scale data using contrastive or masked-prediction self-supervised losses, an action head gets added and the combined model is supervised on robot demonstrations, and the model is optionally fine-tuned with RL or RLHF using a reward model trained from human preferences. RT-2 (arXiv:2307.15818) is an example of the first stage feeding into the second. Post-training with RL remains an active research area; Chapter 18 discusses the current state of reasoning-augmented VLAs that use RL to extend beyond behaviors seen in demonstrations.

A diagnostic question

Before writing any loss function, ask where the training signal comes from. A human selected the target: supervised. The environment evaluated a trajectory: RL. The target derived from the input data itself without human selection: self-supervised. This question gets subtle sometimes. The denoising target x in a diffusion loss was originally a human-demonstrated action, but the noisy version and the loss construction happen automatically, so the loss is self-supervised in character despite its origins. What matters operationally is whether the gradient update points toward human-provided targets (supervised), toward high-reward behavior (RL), or toward internal data consistency (self-supervised).

Getting this right matters for debugging. A supervised model failing points first at data quality: are the demonstrated actions actually correct? An RL model failing points first at reward design: is the reward function specifying what you think it is? A self-supervised model failing points first at the proxy task: is denoising or contrastive alignment actually predictive of downstream performance? §3.5 gives a checklist for diagnosing each of these failure modes in a training run that won't converge.

3.5 Debugging a model that will not train

"It's not converging" is the most common thing a researcher says after spending a week setting up a training run, and it covers at least four distinct situations: the loss decreasing but too slowly, the loss stuck at a plateau, the loss oscillating or spiking, or the loss going NaN outright. Each has a different cause and a different fix, and lumping them together wastes days chasing the wrong hypothesis. This section gives a structured way to tell them apart, organized around the loss curve as the primary diagnostic instrument, with secondary checks on gradients, data, and loss-family-specific failure modes.

The SmallPolicy from §3.3 and the three loss families from §3.4 serve as running examples throughout. Where behavior differs between a 50-line MLP and a 7B-parameter VLA, that gets noted explicitly.

Read the loss curve first

Before touching hyperparameters, look at the loss curve over training steps. Log it to TensorBoard, Weights & Biases, or even a CSV, but log it every N steps, not just at epoch end. The shape of the curve is the fastest way to eliminate hypotheses.

Loss decreasing steadily but not reaching a useful value is usually a capacity problem or a data problem, not a training problem. The optimizer is doing its job; there just isn't enough signal in the data or enough model capacity to reach a useful optimum. Check whether your model is actually large enough for the task's complexity, and whether your demonstrations are consistent or whether different demonstrators do the same task in incompatible ways. Inconsistent demonstrations can hold any model below a certain loss floor, since there's no single answer for it to learn.

Loss plateauing early, then flat, means the optimizer found a local minimum and stopped moving. Common causes: learning rate too small (gradient steps are so tiny the optimization never escapes the initialization basin), weight decay too large (the regularizer overpowers the loss gradient), or gradient vanishing (covered below). Try increasing the learning rate by one order of magnitude, re-run for a few hundred steps, and watch whether the loss breaks out of the plateau. If it does, you have a learning rate problem, not a capacity problem.

Loss oscillating without converging, going down, then up, then down again in a cycle, is almost always a learning rate that's too large. The optimizer overshoots the minimum on every step. Halve the learning rate and check again. If oscillation persists at a lower rate, check whether your loss function has a discontinuity or a very narrow valley; this can happen with binned cross-entropy if the action-bin boundaries sit poorly relative to the actual action distribution.

Loss spiking intermittently, in a training curve that's otherwise well-behaved, is a gradient explosion signature. A single bad batch, one with a large action outlier or a numerical instability from a very deep network, produces a huge gradient that shifts the weights far from the current minimum and destroys progress. Add gradient clipping: `torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)` before the optimizer step. OpenVLA (arXiv:2406.09246) uses a max-norm clip of 1.0 as a default for exactly this reason. If spikes persist after clipping, look at the data: an action outlier orders of magnitude larger than the median action will produce a proportionally large MSE gradient. Outlier filtering or robust normalization (covered below) is the fix.

Loss going NaN immediately or within a few steps means something blew up before training meaningfully started. Check these in order. Is there a `log(0)` or `0/0` in the loss? Cross-entropy on a one-hot target that the model assigns exactly zero probability produces `-log(0) = ∞`. PyTorch's internal `log_softmax` should catch this, but numerical precision failures in mixed-precision (fp16 or bf16) training can produce genuine zeros in the softmax output; switch to full fp32 to diagnose, then add epsilon to probabilities if needed. Are your input features normalized? NaN on step 1 is almost always a normalization failure: the network receives inputs of magnitude 10^3 or 10^4, the first linear layer produces similarly large pre-activations, and the sigmoid or softmax saturates to exactly 0 or 1 in float16. Normalize all inputs to approximately zero mean and unit variance before the first layer.

Gradient inspection

Once you've characterized the loss shape, the next diagnostic is gradient magnitude. Add this to your training loop:

```
for name, param in policy.named_parameters():
    if param.grad is not None:
        writer.add_scalar(
            f"grad_norm/{name}",
            param.grad.norm().item(),
            global_step
        )
```

A well-behaved run shows gradient norms in the range `[1e-4, 1e0]` across most layers, roughly uniform depth-to-depth. Two pathologies look very different from that baseline.

Vanishing gradients: gradient norms near the input end of the network run orders of magnitude smaller than those near the output. The chain-rule product (§3.1) is being multiplied down to near-zero by near-zero Jacobians deep in the network. Solutions: switch from sigmoid or tanh activations to ReLU or GELU, add residual connections (which provide a direct gradient path bypassing deep multiplicative chains), and use layer normalization before each attention or MLP block. Transformer-based VLAs like RT-1 and OpenVLA resist vanishing gradients architecturally, since they have residual connections everywhere, but an MLP policy without residuals can suffer badly in networks deeper than four or five layers.

Exploding gradients: gradient norms spike to `1e3` or `1e6` in certain layers. Gradient clipping controls the symptom, but the root cause is usually one of a very large learning rate, a badly initialized weight matrix (random initialization producing singular values much larger than 1), or a loss function that isn't Lipschitz near the current weights. For flow-matching losses ($\pi 0$, arXiv:2410.24164), check that the time variable `t` is sampled from `[0, 1]` with appropriate clamping. A `t` very close to 0 causes the velocity target `(a* - ε)` to get evaluated at nearly pure noise, which can produce extreme gradient magnitudes.

Data diagnostics

The training loop, gradients, and loss function are the plumbing. The demonstrations are the signal, and getting the signal wrong can mimic every failure mode above; it's frequently the actual cause.

Check action statistics before anything else. Print or plot the mean, standard deviation, minimum, and maximum of each action dimension across the full training set. A joint range of `[-3.14, 3.14]` radians with a few teleoperator slips producing actions of `±50` means those outliers will dominate the MSE gradient and make everything else irrelevant. Two action dimensions with standard deviations of `0.01` and `2.0` means an MSE loss effectively ignores the first, since it contributes negligibly to the scalar loss. The standard fix is per-dimension normalization: subtract the per-dimension mean and divide by the per-dimension standard deviation across the training set. Store these statistics; you'll need them at inference to denormalize the output.

For cross-entropy losses, check bin coverage. If you're using discretized action bins (as in RT-1, OpenVLA), check how many of the 256 bins actually receive training examples in each action dimension. If 90% of demonstrated actions fall in bins 100 through 130, the model quickly learns to assign zero probability to bins 0 through 99 and 131 through 255 and never revisits them. This is fine for accuracy but produces overconfident predictions and poor generalization. Setting bin boundaries from the action statistics so each bin covers an equal fraction of the training distribution (computing percentile boundaries rather than linear spacing) helps substantially; this is what OpenVLA's preprocessing pipeline does.

For supervised loss, check for label noise. A common form in robot demonstrations is late labeling, where the teleoperator's hand was already moving when the timestamp got assigned, so the recorded action corresponds to a slightly later observation frame. Add a constant time-shift correction and verify action timestamps align with observation timestamps within the latency of your recording hardware. A 50 ms misalignment at 10 Hz means half your labels are wrong.

Loss-family-specific diagnostics

§3.4 argued that the loss family determines which failure mode to suspect first. Here are the failure-mode-specific questions for each.

Supervised loss not converging: is the training set large enough? Behavior cloning is sample-hungry, since it learns a direct mapping from observation to action with no mechanism to generalize. As a rule of thumb, you need at least a few hundred demonstrations to learn a single-task policy in a clean simulator, and orders of magnitude more for cross-task generalization. Fewer than 100 demonstrations, and expect a supervised model to overfit rather than generalize. Also ask whether the demonstrations are consistent: two demonstrators with different preferred grasp orientations produce a bimodal action distribution that an MSE-trained model can't represent. Adding a diffusion or flow-matching head (Chapter 10) is the correct architectural fix; reducing the demonstrator pool is the pragmatic data fix.

RL loss not converging: is the reward dense enough? A sparse reward firing only at episode completion gives the policy no gradient signal for the vast majority of its experience. Shaped rewards, intermediate rewards for partial progress such as reducing end-effector distance to the target, are almost always necessary for manipulation tasks longer than a few steps. Also check reward scale: a reward of `1.0` for success combined with an episode length of 500 steps and discount `γ = 0.99` means the discounted value at step 0 is only `0.99^500 ≈ 0.007`. The policy sees no useful gradient through the discount unless the learning rate is tuned to compensate. Chapter 5 covers RL diagnostics in depth.

Self-supervised (diffusion/flow) loss not converging: is the noise schedule appropriate for your action distribution? A diffusion model trained with a noise schedule calibrated for images (bounded in `[0, 1]`) behaves poorly on robot actions with a standard deviation of `2.0`, since the noise magnitudes are miscalibrated. Set the noise schedule relative to the action statistics, not copied from an image diffusion paper. For flow matching specifically, check that the learned velocity field produces trajectories arriving at the target distribution: sample 10 trajectories from the model at inference and plot the final-step distribution against the training action distribution. A mismatch means the velocity points in the right direction but the step size is wrong, or the ODE integrator is accumulating too much error.

A checklist

The sequence of checks, in practice:

1. Print action statistics. Verify range, mean, standard deviation per dimension. Fix outliers and normalize.
2. Verify timestamp alignment. Confirm observation-action pairs are correctly synchronized.
3. Run for 10 steps with a tiny batch of 4. Loss should strictly decrease. If it doesn't, there's a fundamental problem with the forward pass or loss computation.
4. Check gradient norms per layer. They should stay within three orders of magnitude of each other. Add clipping if norms exceed `1.0`.
5. Check for NaN. Add `assert not torch.isnan(loss)` and `assert not torch.isinf(loss)` after the loss computation.
6. Read the loss curve shape. Plateau points to learning rate. Oscillation points to a learning rate that's too high. Spike points to exploding gradients or a data outlier. Steadily decreasing but stuck at the wrong floor points to capacity or data consistency.
7. Apply the loss-family diagnostic. Supervised points to data size and label noise. RL points to reward density and scale. Self-supervised points to noise schedule calibration.

None of this is specific to robotics; it's standard deep learning practice. What makes robot learning unusual is that the feedback loop between training and evaluation runs long and expensive. Running 100 real-robot trials to evaluate whether a training change helped takes days. That cost makes it worth investing heavily in the simulator and logging infrastructure described in Chapter 2 and Chapter 15, so you can evaluate cheaply before committing hardware time.

A model that refuses to train is telling you something specific. Read the loss curve, check the gradients, inspect the data, apply the loss-family checklist in that order, and you'll usually identify the cause before reaching for a more exotic hypothesis.

With the prerequisite tools in place, the calculus, the probabilistic framing, the training loop, the loss families, and the debugging practice, we're ready to close Part 1 and summarize what the first three chapters have established.

3.6 Summary

Chapter 3 was the math chapter, and the one that closes Part 1. The promise at the top was a 30-minute refresher; the actual chapter runs longer than that, because the refresher needed to be specific to robot policies rather than to deep learning in general. The five sections introduced the four objects you'll see on almost every page of the rest of the book (vectors, matrices, gradients, the chain rule), the probabilistic frame that makes those objects behave (random variables, expectations, KL divergence), the 50-line PyTorch loop turning the math into running code, the three loss families that parameterize "training" across the whole field, and the debugging discipline keeping a run from silently producing nonsense. This summary collects the load-bearing ideas in one place and tells you which ones the next four chapters lean on hardest.

The four ideas worth carrying forward

Calculus is the API between a policy and an optimizer. §3.1 made the case that the gradient of the loss with respect to the weights is the only thing SGD ever sees. Everything else, the architecture, the tokenization, the data pipeline, exists to shape that gradient into something useful. The chain rule is the mechanism by which a scalar loss at the end of a 7-billion-parameter network produces a per-parameter update at the beginning. Two consequences follow. When a model fails to learn, the gradient is the first thing to inspect, not the architecture. And the manipulator Jacobian $J(q) \in \mathbb{R}^{6\times 7}$ and the loss Jacobian $\partial L/\partial \theta$ are the same kind of object, computed by the same chain-rule decomposition; the only difference is whether the chain runs through joint angles to end-effector pose, or through network weights to a scalar loss. That's why someone who can derive the Jacobian of a 2-link arm tends to pick up backprop quickly, and why someone who has trained a deep network on images tends to pick up inverse kinematics quickly. They're reusing the same idea.

Cross-entropy and KL divergence are the same loss in two notations. §3.2 worked through this twice, once with the math and once with the OpenVLA action-bin example, because it's the single most useful identity for reading the modern literature. RT-1 and OpenVLA describe themselves as training with "cross-entropy loss on discretized action bins"; π0 describes itself as training with a flow-matching loss equivalent to minimizing a particular KL divergence between forward and reverse processes. The descriptions sound different. They aren't. Cross-entropy $H(p, q) = -\sum_i p_i \log q_i$ and KL divergence $D_{\mathrm{KL}}(p\|q) = \sum_i p_i \log(p_i/q_i)$ differ only by the entropy of p, a constant when p is the target distribution. Minimizing one minimizes the other. Once that's internalized, the loss function of essentially every supervised and self-supervised action model in the book reduces to one of three patterns: cross-entropy on tokens (RT-1, OpenVLA), MSE on continuous actions (ACT, Diffusion Policy, arXiv:2303.04137), or a denoising-style velocity loss (π0, arXiv:2410.24164). The variety in named loss functions is mostly notation.

A training loop is six lines doing the actual work and a few hundred lines of plumbing. §3.3 made this concrete by writing the SmallPolicy loop in 50 lines and annotating which six lines carry out the optimization step. The remaining 44 lines are data loading, logging, learning-rate scheduling, and device placement, necessary but mechanical. Read the OpenVLA training code, or the Diffusion Policy training code, or the π0 training code, and the six lines are present unchanged: forward pass, loss computation, zero gradients, backward pass, optimizer step, scheduler step. What differs between codebases is the plumbing, especially the data pipeline, and this matters because the plumbing is where almost all the bugs live. A correctly written six-line update running on a corrupted dataloader still produces a model that doesn't work. The §3.5 diagnostic, printing action statistics before training a single step, exists precisely because the data pipeline is the default suspect.

The loss family determines the failure mode. §3.4 introduced the three families and §3.5 turned them into a diagnostic table. Supervised losses fail by overfitting, mode collapse on multimodal data, and label noise. Reinforcement-learning losses fail by reward hacking, sparse-reward stagnation, and reward miscalibration. Self-supervised losses fail by noise-schedule mismatch and integrator error at sampling time. Knowing which family your loss belongs to narrows the hypothesis space the moment a run starts misbehaving. The whole point of organizing the chapter this way is that "my model isn't training" isn't a question anyone can answer, while "my supervised behavior-cloning loss plateaus at 1.2 nats per token after step 5000 on a dataset with two demonstrators" is a question with two or three plausible answers and a clear next experiment.

What you should be able to do now

Four concrete capabilities, in increasing order of how much the rest of the book relies on them.

You should be able to read a model card or paper appendix and identify the loss function in one pass. The OpenVLA paper says "we train with the standard next-token cross-entropy objective over 256 discrete action bins per dimension." That sentence packs three §3.4 commitments, supervised family, cross-entropy form, tokenized output, and it lets you predict the §3.5 diagnostic suite you'll need when fine-tuning the model. The Diffusion Policy paper says "we train an ϵ-prediction denoising network conditioned on observations, with a fixed cosine noise schedule." That sentence packs four commitments: self-supervised family, MSE-on-noise form, continuous output, fixed schedule. Reading either sentence quickly saves the half-day you'd otherwise spend reverse-engineering the loss from the training script.

You should be able to write a 50-line PyTorch training loop from scratch without consulting a reference. Not a state-of-the-art one, a working one: the six update lines, plus a dataloader, plus a logging hook, plus a learning-rate scheduler. The SmallPolicy from §3.3 is the template, and copying it for a new architecture should be a 20-minute job, not a two-day job. This skill matters because every chapter from 6 onward asks you to train at least one variant of the model under discussion, and the hands-on exercises assume you can stand up a loop without ceremony. The handful of moving parts the loop needs, gradient zeroing, backward call, optimizer step, scheduler step, are universal across PyTorch projects. The §3.4 loss family is the only piece that changes from chapter to chapter.

You should be able to debug a non-converging run in under an hour by working the §3.5 checklist: print action statistics, verify timestamp alignment, run for 10 steps on a batch of 4, inspect gradient norms, add NaN assertions, read the loss curve, apply the loss-family diagnostic. This checklist exists because in practice the cause is almost always one of seven things, and identifying which of the seven saves you from the much longer tail of "it must be the architecture." A modern VLA fine-tune costs on the order of hundreds of GPU-hours; misdiagnosing gets paid back in days. The checklist is cheap by comparison.

You should be able to predict the architectural consequences of a loss choice. A paper proposing a new VLA that uses MSE on continuous actions with a single deterministic head should immediately prompt the question: how does it handle bimodal demonstrations? A paper proposing RL fine-tuning on a real robot should prompt: what's the reward function, and how dense is it? A paper proposing a flow-matching action head should prompt: how many integration steps at inference, and what's the latency budget? These aren't gotcha questions. They're the questions the §3.4 taxonomy implies, and asking them surfaces the design decisions a paper has made and the trade-offs it has accepted. By the end of Chapter 10 you'll recognize each of these trade-offs as the entry point to a specific later chapter, with the §3.4 vocabulary as the index.

Where the chapter has set up the rest of the book

Part 1 is now complete. Three forward references are worth naming again, since they structure all of Part 2 and most of Part 3. The three-loss-families framing from §3.4 is the spine of Chapter 5 (RL family in depth), Chapter 6 (imitation family in depth), Chapter 7 (deep RL), and Chapter 10 (the self-supervised family applied to action generation); each of those chapters elaborates one family into its own design space. The PyTorch loop from §3.3 is the substrate for every code listing in Chapters 6, 7, 10, 11, and 12; later chapters add larger models, bigger datasets, and richer logging, but the six-line update stays put. The debugging checklist from §3.5 is the substrate for Chapter 16's fine-tuning recipes and Chapter 17's deployment monitoring, both of which treat the checklist as a prerequisite rather than introducing their own.

The chapter hasn't set up two things you might have expected. It hasn't covered transformers, attention, or the architectural details of the models listed in Part 4; those wait for Chapter 8, where they get discussed alongside Decision Transformer and the lineage that produced RT-1. It also hasn't covered the data side, Open X-Embodiment, LIBERO, CALVIN, SimplerEnv, which is the entire subject of Chapter 15. The §3.5 admonition to print action statistics before training a single step is a preview; the systematic treatment of robot datasets, their failure modes, and their benchmark properties sits twelve chapters away.

Part 1 closes here. Chapters 1, 2, and 3 together gave you the vocabulary of action models (Chapter 1), a working VLA you trained and broke yourself (Chapter 2), and the math, code, and debugging instinct that turn the rest of the book into a series of one-section variations on recognizable themes (Chapter 3). Part 2 starts in Chapter 4 with the

classical-methods family, PDDL, IK, motion planning, computed-torque control, and walks the four-family taxonomy forward through Chapter 7. By the time you reach Chapter 11 and the VLA recipe proper, every component you encounter will be something you've already seen a smaller, earlier version of. That progression is the design of the book, and Part 1 is where the design gets set.

§3.x closes the chapter with one hands-on exercise, a debugging puzzle on an intentionally broken SmallPolicy run with the cause hidden in one of the seven places §3.5 named, and the full reading list for the chapter.

3.x Hands-on exercise + chapter references

Chapter 3 was the math chapter; the exercise is where that math shows up on a stopwatch. The four drills below take a combined two hours on a laptop CPU. No GPU required, no robot required, no internet required after the initial dependency install. The point is to leave the chapter with the SmallPolicy from §3.3 sitting in a directory on your disk, a debugging notebook beside it with the seven §3.5 checks scripted into reusable functions, and a one-page side-by-side comparison of the three §3.4 loss families on the same toy task. That artifact is what Chapters 6, 7, and 10 will assume you can stand up from scratch in under twenty minutes.

Exercise 3.x.1 — Implement SmallPolicy and reproduce the baseline curve

Open a new file called `small_policy.py` and re-implement the SmallPolicy from §3.3 based on the chapter's description. Don't copy the listing verbatim. Writing it from the description forces you to notice the six lines of actual optimization work against the surrounding plumbing, which is the central claim of §3.6. The architecture is a three-layer MLP: input is a 14-dimensional observation (7 joint angles, 7 joint velocities), output is a 7-dimensional continuous action, hidden size 256, ReLU activations, no dropout. The dataset is synthetic, a toy inverse-dynamics map where the action is a fixed linear function of the observation plus Gaussian noise. Generate 10,000 training pairs and 2,000 validation pairs once, cache them to `.npz`, and reuse the cache across all four exercises so the data side stays identical throughout.

Train with MSE loss, Adam at `lr=1e-3`, batch size 64, for 50 epochs. Log training and validation loss to a CSV every 100 steps. The deliverable is one plot, training loss and validation loss on the same y-axis, step on the x-axis. You should see a clean monotone decrease for both, validation sitting slightly above training, both flattening around step 3,000. If your curves look qualitatively different, oscillation, NaN, flat from step 0, no gap between train and validation, stop and run the §3.5 checklist before continuing. The whole point of this exercise is that the baseline curve has a known shape, and any deviation is diagnostic. Save the plot as `baseline_mse.png`.

Wall clock on a 2023-era laptop CPU: about ten minutes.

Exercise 3.x.2 — Break the loop on purpose, one fault at a time

Copy `small_policy.py` to `broken_drills.py`. Introduce, one at a time and never simultaneously, the following five faults, each in its own branch controlled by a `fault` argument. Run a 50-epoch training for each fault, and save the loss curve. Don't look at the §3.5 prediction for a fault until after you've run the drill and stared at the curve yourself.

1. `fault="big_lr"`: change `lr` from `1e-3` to `1e0`. Expected signature from §3.5: oscillation, possibly NaN.
2. `fault="tiny_lr"`: change `lr` to `1e-8`. Expected signature: training loss decreases by less than a percent in 50 epochs; flat- looking curve.
3. `fault="unnormalized_inputs"`: skip the per-dimension normalization step on the observations, so they are passed in raw with a standard deviation around 50. Expected signature: NaN within the first few steps, or extremely large initial loss that decays anomalously.
4. `fault="action_outlier"`: inject one demonstration per batch with an action value of `1e3` in dimension 3, leaving the other 99% of data intact. Expected signature: periodic loss spikes whenever a poisoned batch is drawn; gradient clipping makes them disappear.
5. `fault="bimodal_labels"`: synthesize the dataset with two demonstrators whose preferred actions differ by a fixed offset in dimension 0, so the target distribution at each observation is bimodal. Expected signature: training loss plateaus at a value roughly equal to half the squared inter-mode distance, never lower.

For each fault, write one sentence, "training loss did X, gradient norm did Y, the diagnosis from §3.5 is Z," and save all six plots (baseline plus five faults) to `drills/curves/`. The collection is the deliverable. Most students get four of the five diagnoses right on the first read of the curves; the bimodal one is hardest, since the plateau looks similar to a capacity ceiling. The Chapter 10 discussion of Diffusion Policy (arXiv:2303.04137) and ACT (arXiv:2304.13705) returns to this exact failure as the motivating example for distributional action heads.

Wall clock: about thirty minutes for all five drills combined.

Exercise 3.x.3 — The three-loss-family bake-off

This is the loss-family exercise from §3.4 made concrete. Take the same SmallPolicy architecture and the same synthetic dataset from Exercise 3.x.1, and train three versions:

- *Supervised cross-entropy*: discretize each action dimension into 64 uniform bins (taken from the per-dimension min and max of the training set). The model outputs `7 × 64` logits per observation; loss is per-dimension cross-entropy summed across dimensions. This is a smaller version of the RT-1 / OpenVLA recipe (arXiv:2212.06817, arXiv:2406.09246), bins reduced from 256 to 64 to fit a CPU budget.
- *Supervised MSE*: the baseline from 3.x.1. Continuous output, MSE loss. This is the ACT-style recipe (arXiv:2304.13705), simplified.
- *Self-supervised denoising MSE*: add Gaussian noise of variance σ^2 to the target actions (with σ sampled per-batch from a uniform schedule on `[0.01, 1.0]`), pass observation plus the noisy action plus σ to the model, and have it predict the noise. This is a minimal Diffusion-Policy-shaped objective (arXiv:2303.04137).

Train each for 50 epochs with the same Adam settings, log the same fields, and produce one comparison plot: three loss curves on the same axes, normalized by their respective theoretical minimum so the y-axes stay comparable. Then, at inference, do something interesting: for each of 20 held-out observations, sample 10 predictions from each model (deterministic argmax for cross-entropy, deterministic forward pass for MSE, 10 reverse-process samples with 20 integration steps each for the denoising model). Plot the spread.

The plot should make three things visible. The cross-entropy model is deterministic at argmax and produces zero spread; switching to a sampled prediction makes it multimodal, but only at bin resolution. The MSE model collapses to a single mean prediction at every observation regardless of bimodality in the data. The denoising model produces a smooth distribution that, on the bimodal dataset from Exercise 3.x.2 fault 5, recovers both modes. That single plot is the visual answer to "why does $\pi 0$ use flow matching" (arXiv:2410.24164) and "why does Octo use a diffusion head" (arXiv:2405.12213). Chapter 10 spends most of its length unpacking what you see here in 30 minutes of CPU time.

Wall clock: about forty-five minutes including the inference sweep.

Exercise 3.x.4 — Read one PyTorch paper appendix

Open the OpenVLA paper (Kim et al., 2024, arXiv:2406.09246) to the training-details appendix (typically Appendix A or B; the exact section heading differs between arXiv versions). Read only that appendix. Mark, with a pencil or in a text file, every hyperparameter specifically called out: optimizer, learning rate, schedule, weight decay, batch size, gradient-clip threshold, mixed-precision dtype, warmup steps, total steps. Then open `small_policy.py` and write a short comment block above your training loop recording the difference between each of those values and what you used. For example: "OpenVLA: AdamW, `lr=5e-5`, cosine schedule with 1000 warmup steps, weight decay `0.01`, grad clip `1.0`, bf16 mixed precision. Me: Adam, `lr=1e-3`, no schedule, no weight decay, no clip, fp32." There's no scoring rubric here. The point is making the OpenVLA training appendix stop feeling like a list of inscrutable magic numbers, since every one of those numbers is a knob you've now turned at least once in the wrong direction yourself.

When you reach Chapter 16 and need to fine-tune OpenVLA for your own data, you'll return to this comment block. The shape of what you have to tune away from defaults matters far more than the specific values.

Wall clock: about thirty minutes for the read plus annotation.

Chapter 3 reading list

The works below are the ones cited across §3.1 through §3.6, grouped by purpose. Full bibliographic entries for everything cited in the whole book live in Appendix E.2; this list is just the chapter-local subset.

Deep learning foundations (linear algebra, calculus, training)

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press. Chapters 2–4 (linear algebra, probability, numerical computation) cover what §3.1 and §3.2 sketched in 30 pages.
- Murphy, K. P. (2022). *Probabilistic Machine Learning: An Introduction.* MIT Press. The probability and information-theory chapters are the canonical reference for §3.2.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer. The older reference, still the cleanest exposition of the KL-vs-cross-entropy equivalence used in §3.2.
- Kingma, D. P., & Ba, J. (2015). "Adam: A Method for Stochastic Optimization." arXiv:1412.6980. The optimizer that shows up in §3.3 and in every chapter from 6 onward.
- Loshchilov, I., & Hutter, H. (2019). "Decoupled Weight Decay Regularization." (AdamW.) arXiv:1711.05101. The OpenVLA training appendix uses AdamW; this is the paper that explains why.
- He, K., Zhang, X., Ren, S., & Sun, J. (2015). "Delving Deep into Rectifiers." arXiv:1502.01852. The initialization scheme that prevents the vanishing-gradient pathology §3.5 names.

Loss families: supervised, RL, self-supervised

- Pomerleau, D. A. (1988). "ALVINN." *NeurIPS 1988*. The original supervised behavior-cloning paper; the entire §3.4 supervised-family diagnostic table grows out of the failure modes ALVINN exhibited.
- Ross, S., Gordon, G., & Bagnell, D. (2011). "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." (DAgger.) *AISTATS 2011*. Cited in Chapter 6; the canonical compounding-error response §3.4 forward-references.
- Schulman, J., Wolski, F., Dhariwal, P., et al. (2017). "Proximal Policy Optimization Algorithms." arXiv:1707.06347. The reference RL loss for the §3.4 RL-family discussion.
- Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). "Soft Actor-Critic." *ICML 2018*. The off-policy comparison point in §3.4.
- Ho, J., Jain, A., & Abbeel, P. (2020). "Denoising Diffusion Probabilistic Models." arXiv:2006.11239. The diffusion training objective Exercise 3.x.3 minimally re-implements.
- Lipman, Y., Chen, R. T. Q., Ben-Hamu, H., et al. (2023). "Flow Matching for Generative Modeling." arXiv:2210.02747. The flow-matching objective $\pi 0$ (arXiv:2410.24164) inherits, mentioned in §3.4 and §3.5.

Action-model instances used as worked examples

- Brohan, A., Brown, N., Carbajal, J., et al. (2022). "RT-1: Robotics Transformer for Real-World Control at Scale." arXiv:2212.06817. The cross-entropy-on-bins recipe from §3.2 and §3.4.
- Kim, M. J., Pertsch, K., Karamcheti, S., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246. The scaled-up version of the same recipe, and the source of the gradient-clip-1.0 default §3.5 cites.
- Chi, C., Feng, S., Du, Y., et al. (2023). "Diffusion Policy: Visuomotor Policy Learning via Action Diffusion." arXiv:2303.04137. The self-supervised-on-actions worked example in §3.4 and the motivating model behind Exercise 3.x.3.
- Zhao, T. Z., Kumar, V., Levine, S., & Finn, C. (2023). "Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware." (ACT.) arXiv:2304.13705. The MSE-on-continuous-actions instance compared in Exercise 3.x.3.
- Octo Model Team (2024). "Octo: An Open-Source Generalist Robot Policy." arXiv:2405.12213. The diffusion-head VLA referenced in §3.4.
- Black, K., Brown, N., Driess, D., et al. (2024). "$\pi 0$: A Vision-Language-Action Flow Model for General Robot Control." arXiv:2410.24164. The flow-matching example in §3.5; the chapter that unpacks it is Chapter 13.

Background you may want nearby

- Lynch, K. M., & Park, F. C. (2017). *Modern Robotics: Mechanics, Planning, and Control.* Cambridge University Press. Chapter 5 derives the manipulator Jacobian used in §3.1.
- Siciliano, B., Sciavicco, L., Villani, L., & Oriolo, G. (2010). *Robotics: Modelling, Planning and Control.* Springer. The alternative reference for the same Jacobian derivation.
- Paszke, A., Gross, S., Massa, F., et al. (2019). "PyTorch: An Imperative Style, High-Performance Deep Learning Library." arXiv:1912.01703. The PyTorch reference; useful when the §3.3 training loop has to be ported to a more involved data pipeline.

Chapter summary

Chapter 3 was the math, code, and debugging chapter, and it closes Part 1. You can now write the gradient of a scalar loss with respect to a parameter vector and follow it through a transformer. You can convert between cross-entropy and KL divergence without thinking about it. You can stand up a 50-line PyTorch training loop for any of the three loss families in under twenty minutes, and you have a seven-step debugging checklist that diagnoses the most common reasons a training run refuses to converge. With those four capabilities in hand, Part 2 begins. Chapter 4 covers the classical-actions family, STRIPS, PDDL, inverse kinematics, motion planning, computed-torque control, both the oldest layer in the action-model story and the one still running underneath every modern VLA stack you'll encounter in the rest of the book.

Chapter 4 — Classical action models: planning and inverse dynamics

4.1 Symbolic actions: STRIPS, PDDL, and action schemas

The action models you've met so far emit numbers. OpenVLA emits a 7-vector every 50 ms, the SmallPolicy from §3.3 emits a 7-vector every step, and Diffusion Policy emits a sequence of 7-vectors at once. Symbolic action models emit something else entirely: a name and a list of arguments, drawn from a finite alphabet the engineer wrote down ahead of time. `pick(block_a)` is a symbolic action. So is `move(robot, kitchen, living_room)`, or `pour(cup_1, bowl_2)`. No real number appears anywhere in the output. It reads closer to a programming-language statement than to a vector.

This is the family the field started with, and the family every other chapter of this book defines itself against in some way. Part of the reason for opening Chapter 4 here is historical: STRIPS predates every other family by at least a decade, and the rest of the field's vocabulary inherits its terminology. The more important reason is that symbolic action models still run, today, inside the top layer of many deployed robots. The hierarchical decomposition an LLM produces when SayCan (arXiv:2204.01691) asks "what should the robot do next" is, structurally, a STRIPS plan. The task scheduler inside a warehouse robot that ships a VLA for grasping is, structurally, a STRIPS planner. Understanding what a symbolic action is, and what it isn't, is the prerequisite for reading those modern systems as systems rather than as black boxes.

The STRIPS world model

STRIPS, the Stanford Research Institute Problem Solver (Fikes and Nilsson, 1971), was the planner that drove Shakey, the first mobile robot built to navigate a physical lab. The representation has three pieces, and once you have these three you can read essentially every symbolic planner written since.

The first piece is a set of predicates, Boolean facts about the world. `(on block_a block_b)` says block A sits on top of block B. `(holding block_a)` says the robot's gripper currently holds block A. `(clear block_b)` says nothing sits on top of block B. A state is a set of predicates currently true; everything not in the set is assumed false (the closed-world assumption). For a five-block tabletop, the state might be the four-element set `{(on a b), (on b table), (clear a), (handempty)}`. That's the entire world model the planner has to work with.

The second piece is a set of actions, operators that take a state to a new state. Each action has a name with typed arguments, a list of preconditions that must hold for the action to apply, and a list of effects saying which predicates become true and which become false once the action fires. The canonical pick-up action:

```
action: pick-up(?x - block)
  precondition:  (clear ?x) and (on ?x table) and (handempty)
  effect:        not (on ?x table) and not (clear ?x) and
                 not (handempty) and (holding ?x)
```

The `?x` is a parameter, making this an action schema with a free variable. To actually execute, the planner grounds the schema by binding `?x` to a specific block, producing concrete instances like `pick-up(a)`, `pick-up(b)`, and so on. Grounding turns a small set of schemas into a much larger set of actually-executable actions. For ten blocks and

four schemas you can easily end up with a few hundred ground actions, and for a realistic warehouse domain with thousands of objects, grounding alone can produce millions of ground actions and become a non-trivial step in its own right.

The third piece is the planning problem: an initial state, a set of ground actions derived from the schemas, and a goal expressed as a partial predicate set. A solution is a sequence of ground actions that, applied in order, transforms the initial state into one satisfying the goal. The planner's job is searching for such a sequence, typically through heuristic search over the state space (depth-first, breadth-first, A* with a relaxed-plan heuristic, and various more recent landmark-based heuristics).

That's it. Predicates, action schemas, planning problem. Every PDDL domain you'll ever read sits on top of those three abstractions.

PDDL: the standard syntax

STRIPS as a representation outlasted STRIPS as a piece of software. By the mid-1990s every research group had its own incompatible syntax, and the 1998 AIPS planning competition demanded a common one. The result was the Planning Domain Definition Language (McDermott et al., 1998), known universally as PDDL. PDDL has gone through several versions: PDDL 2.1 (2003) added numeric fluents and durative actions, PDDL 3 (2005) added preferences and trajectory constraints, and the various PDDL+ dialects extend the language for hybrid and probabilistic domains. Chapter 4 mostly sticks to PDDL 1, the plain STRIPS-flavored subset, since it's small enough to teach in a section and large enough to express the exercise at the end of the chapter.

A PDDL specification has two files: the domain file, which defines predicates and action schemas (and is reusable across many problems), and the problem file, which defines a specific initial state and goal. Here's a complete domain for a minimal block-stacking world:

```
(define (domain blocks)
  (:requirements :strips :typing)
  (:types block)
  (:predicates
    (on ?x - block ?y - block)
    (on-table ?x - block)
    (clear ?x - block)
    (holding ?x - block)
    (handempty))
  (:action pick-up
    :parameters (?x - block)
    :precondition (and (clear ?x) (on-table ?x) (handempty))
    :effect (and (holding ?x) (not (on-table ?x))
                 (not (clear ?x)) (not (handempty))))
  (:action put-down
    :parameters (?x - block)
    :precondition (holding ?x)
    :effect (and (on-table ?x) (clear ?x) (handempty)
                 (not (holding ?x)))))
```

And a problem instance asking the planner to swap two blocks:

```
(define (problem swap-ab)
  (:domain blocks)
  (:objects a b - block)
  (:init (on a b) (on-table b) (clear a) (handempty))
  (:goal (and (on b a) (on-table a))))
```

Saved as `blocks.pddl` and `swap.pddl`, those two files are enough input for any STRIPS-compatible planner. The exercise at the end of the chapter asks you to run Fast Downward (Helmert, 2006) on a slightly larger version of this domain; for now, the point is just reading the syntax without flinching.

Two pieces of PDDL syntax trip up beginners often enough to flag here. The `:typing` requirement is what makes `?x - block` mean "`?x` is of type block"; without `:typing`, the planner has no notion of types and treats every parameter as an untyped object. The negation form `(not (on-table ?x))` inside an `:effect` block means "remove the predicate `(on-table ?x)` from the state." PDDL uses the same `not` token for two distinct purposes, a precondition test and an effect deletion, and which one applies is purely positional.

Search, heuristics, and why classical planning is fast

Given a domain and a problem, what does a planner actually do? At the highest level, it searches over the implicit graph whose nodes are states and whose edges are ground actions, looking for a path from the initial state to a goal-satisfying state. The branching factor, the number of ground actions applicable in a state, typically runs dozens to thousands; the depth, the number of actions in an optimal plan, typically runs a few to a few hundred. A naive breadth-first search is hopeless even for medium domains.

Modern planners get their speed from heuristics. The single most influential idea is the relaxed-plan heuristic: pretend, for the sake of estimating distance to the goal, that actions have only positive effects and ignore the negative ones. Under that relaxation, the planning problem becomes polynomial, essentially a reachability calculation, and its solution length gives an admissible-ish estimate of the true distance. The Fast-Forward planner (Hoffmann and Nebel, 2001) made the relaxed-plan heuristic practical, and Fast Downward (Helmert, 2006), the current de facto baseline, uses landmark-based and merge-and-shrink heuristics building on the same idea. The headline number is that Fast Downward routinely solves problems with millions of ground actions in seconds. That kind of speed is what keeps symbolic planners load-bearing in modern stacks: even when the low-level actions come from a learned policy, a STRIPS-style task layer can sit on top and decide which subgoal to issue next without slowing the loop down noticeably.

A worked example, briefly: for the swap-ab problem above, an optimal plan runs four actions long, `unstack(a, b); put-down(a); pick-up(b); stack(b, a)`, assuming we add `stack` and `unstack` schemas alongside pick-up and put-down (omitted here for space; the full domain lives in the chapter's hands-on directory). Fast Downward solves this problem in milliseconds. The search space for a four-block instance is still tractable for a human to reason through by hand; by ten blocks it isn't, and by a hundred it's firmly planner territory.

What an action schema is, and what it is not

An action schema is not a controller. It says nothing about how the robot's joints move to accomplish `pick-up(a)`; it only asserts that, if the preconditions hold, the action can be issued, and after it issues, the effects will hold. The actual motion of the arm, the seven-vector at 50 ms that §3.1 spent so much time on, is somebody else's problem entirely.

That somebody else, in a classical stack, is the geometric layer covered in §4.2 (inverse kinematics, motion planning) and the dynamic layer covered in §4.3 (computed-torque control). In a modern stack, the somebody else is a learned policy: a VLA fine-tuned to execute pick primitives, an RL controller for legged locomotion, a Diffusion Policy for fine manipulation. The interface between the symbolic layer and the executor layer is the ground-action name with its bindings, `pick-up(block_a)`, plus whatever geometric context the executor needs (a target pose, a grasp annotation, a language string). This vertical decomposition is what enables modern task-and-motion planning (Garrett et al., 2020, ICAPS) and language-grounded planning (SayCan, arXiv:2204.01691) to combine symbolic reasoning at the top with learned actuation at the bottom.

Three failure modes of pure-symbolic action models are worth naming, since they directly motivate the rest of the book. The frame problem: writing down every effect of every action is brittle, and forgetting a single effect produces plans that look correct on paper and fail in reality. The grounding problem: the planner assumes its predicates correspond to real-world facts, but `(on a b)` isn't something a robot can directly observe; it has to be perceived first. The coverage problem: any action the engineer didn't write a schema for is invisible to the planner. Learned action models address these one by one. A Diffusion Policy doesn't need a frame-perfect effect model, since it operates on observations directly. A VLA doesn't need predicate grounding, since it consumes raw pixels. A foundation-scale imitation model doesn't need hand-engineered schemas, since it acquires its action repertoire from demonstration.

What it gives up in exchange are the guarantees a symbolic planner provides: that the plan is provably correct under its world model, that sub-optimality bounds are explicit, and that adding a new goal is a one-line change to a problem file rather than a fine-tune. This trade is fundamental, and we'll revisit it explicitly in §4.4 and again in §17.5 when discussing what we can't certify.

The next section moves from the symbolic layer down to the geometric one and asks how a ground action like `pick-up(block_a)` actually becomes a collision-free trajectory of joint angles.

4.2 Geometric actions: inverse kinematics and motion planning

A STRIPS planner hands the executor a string: `pick-up(block_a)`. The executor's first job is turning that string into something a motor controller can consume, a sequence of joint-angle vectors, sampled at maybe 100 Hz, that moves the gripper from where it is now to a pose letting it close around block A without colliding with the table, with block B, or with itself. That translation is this section's topic. It's a layer almost every modern robot still runs, sitting either under a symbolic planner (the classical task-and-motion stack) or under a learned high-level policy (a VLM proposing subgoals to a geometric controller).

The translation splits into two distinct subproblems. The first is inverse kinematics (IK): given a desired pose for the end effector, what joint angles produce it? The second is motion planning: given a starting configuration and a goal configuration, what continuous, collision-free path connects them? IK asks about one instant in time; motion planning asks about the curve through configuration space. Confusingly, both get called "geometric" because both ignore the forces and torques actually driving the joints, that's §4.3's problem. Here, the robot is treated as a rigid mechanism that can be commanded to any kinematically reachable configuration, and the question is which configuration, and how to get there.

Forward kinematics, then inverse

A serial manipulator with `n` joints is parameterized by a vector `q ∈ R^n` of joint angles (or, for prismatic joints, displacements). Forward kinematics is the map from `q` to the pose of the end effector, written `T(q) ∈ SE(3)`, a 4×4 homogeneous transform encoding both position and orientation. The map is built by chaining one transform per link, using Denavit–Hartenberg parameters or the more modern product-of-exponentials formulation (Lynch and Park, 2017, Ch. 4). It's closed-form, smooth, and cheap: for a 7-DOF arm, evaluating `T(q)` takes roughly two dozen multiplications. There's no algorithmic difficulty in forward kinematics, only bookkeeping.

Inverse kinematics runs the opposite direction: given a target pose `T_desired`, find a `q` with `T(q) = T_desired`. The difficulty here is real. For a generic 6-DOF arm, IK is a system of six nonlinear equations in six unknowns. For arms with three intersecting wrist axes, Puma, Kuka KR series, most industrial designs, Pieper's solution from 1968 gives a closed-form answer: position decouples from orientation, and each piece solves with trigonometric identities. For arms without that geometric niceness, and for any robot with more than six joints, which includes most modern collaborative arms (Franka Panda, KUKA iiwa, Universal Robots), there's no closed form, and the problem becomes either numerical or underdetermined. A 7-DOF arm reaching a 6-DOF target has a one-dimensional manifold of solutions, a continuous family of elbow positions, the "elbow swivel" parameter. The IK solver has to pick one, and the choice matters; picking poorly puts the elbow into a doorframe.

How an IK solver actually works

In practice almost no one writes their own IK solver anymore. Everyone calls into TRAC-IK (Beeson and Ames, 2015), KDL, or IKFast (the analytic generator distributed with OpenRAVE). It's still worth understanding the two underlying recipes, since their failure modes show up right at the interfaces.

The closed-form recipe is what IKFast does. It takes the URDF of the robot, symbolically derives the trigonometric equations from the forward-kinematics chain, and emits a C++ file that, given a target pose, returns the up-to-sixteen analytic solutions in microseconds. It's exact, has no hyperparameters, and is the default choice when the arm has the right geometry. It fails when the arm doesn't, which covers most 7-DOF arms, because the symbolic derivation simply never terminates.

The numerical recipe handles everything else. It starts from a current guess `q_0` and iterates a small update `q_{k+1} = q_k + Δq` until the error `|T(q_k) - T_desired|` falls below a tolerance. The update uses the kinematic Jacobian `J(q) = ∂T/∂q` (a 6×n matrix mapping joint velocities to end-effector twists), inverted in some sense to map the pose error back to a joint update. The simplest version is `Δq = J^+ · e`, where `J^+` is the Moore-Penrose pseudoinverse and `e` is the 6-vector pose error. In practice everyone uses damped least squares instead, `Δq = J^T (JJ^T + λ²I)^{-1} e` (Nakamura and Hanafusa, 1986), because it stays well-conditioned near singularities, configurations where `J` loses rank and the arm momentarily can't move in some Cartesian direction. TRAC-IK runs damped least squares and sequential quadratic programming in parallel and returns whichever converges first.

The failure modes are worth a paragraph, since they're what an executor above the IK layer has to handle. No solution exists: the target pose sits outside the workspace, or inside a self-collision. A solution exists but the solver missed it: numerical IK is a local method, and a bad initial guess can sit in a basin that never reaches the global optimum. Multiple solutions exist: which one comes back depends on the seed, and naively re-seeding between calls produces a robot that flips its elbow between consecutive waypoints. Every production system wraps IK to handle these cases, seeding with the previous solution, filtering by collision, choosing among solutions by a "distance from current pose" heuristic.

Configuration space and the motion-planning problem

Suppose IK has succeeded and returned a goal configuration `q_goal`. The robot currently sits at `q_start`. Why not just interpolate the joint angles linearly between them? Because the straight line in joint space typically passes through configurations where the arm sits inside the table, inside itself, or inside the block it's trying to grasp.

The cleanest way to think about this is configuration space, or C-space: the space of all joint vectors `q ∈ R^n` (with appropriate periodicity for revolute joints). Each pose of the robot is one point in C-space. Each obstacle in the workspace, the table, the second block, the gripper itself, projects to a forbidden region `C_obs ⊂ R^n`, and the free region `C_free = R^n \ C_obs` is where the robot is allowed to be. The motion-planning problem becomes finding a continuous curve from `q_start` to `q_goal` lying entirely inside `C_free`.

`C_free` is the catch. It's implicitly defined; there's no closed-form description of it, only a black-box collision-check taking a `q` and returning true or false. For a 7-DOF arm in a typical tabletop scene, one collision check costs a few hundred microseconds (FCL, the standard library, broadphases AABB intersections then runs GJK on candidate pairs). Motion planners get judged, in large part, by how few of these checks they need.

Sampling-based motion planning

The dominant approach for robotic arms is sampling-based motion planning (LaValle, 2006). The idea is simple almost to the point of embarrassment: rather than explicitly modeling `C_free`, sample points uniformly at random from the joint-space box, discard the ones that collide, and build a graph out of the survivors. Two algorithms dominate.

Probabilistic Roadmap (PRM), introduced by Kavraki et al. (1996), samples thousands of configurations once, connects each to its k nearest neighbors with straight-line edges (checking each edge for collisions along its midpoint and recursively), and stores the resulting roadmap. At query time, `q_start` and `q_goal` get connected to the roadmap and Dijkstra finds a path. PRM is multi-query: you amortize the roadmap build across many planning queries in the same scene.

Rapidly-exploring Random Tree (RRT), introduced by LaValle (1998), builds a tree rooted at `q_start`. Each iteration samples a random `q`, finds the nearest node in the tree, and extends the tree by stepping a small distance toward the sample, adding the new node if the edge is collision-free. Bias the sampling toward `q_goal` with probability 0.05, and the tree finds the goal quickly. RRT-Connect (Kuffner and LaValle, 2000) grows two trees, one from start and one from goal, and connects them, empirically the fastest single-query planner for high-DOF arms and the default in most modern stacks.

Neither RRT nor PRM produces optimal paths. The RRT* and PRM* variants from Karaman and Frazzoli (2011) recover asymptotic optimality by rewiring edges as new nodes arrive; in the limit of infinite samples, the returned path converges to the shortest one. In practice, the post-processing step that matters most is shortcutting: take the returned waypoint sequence, pick random pairs of waypoints, and replace the segment between them with a

straight line if that line is collision-free. A 200-waypoint RRT output typically shortens to about 20 waypoints after shortcutting.

The reference implementation everyone uses is OMPL (Şucan, Moll, and Kavraki, 2012), wrapped in MoveIt for ROS. A typical IK-plus-RRT-Connect-plus-shortcut pipeline for a 7-DOF arm in a known scene plans a pick motion in 100 to 500 ms.

Optimization-based motion planning

The other family is optimization-based. CHOMP (Ratliff et al., 2009) formulates planning as gradient descent on a trajectory cost, a sum of smoothness (joint acceleration squared) and obstacle cost (the distance field of the scene, smoothed). It starts from a straight-line initialization in joint space and iteratively pushes the trajectory away from obstacles. TrajOpt (Schulman et al., 2013) does the same thing with sequential convex optimization and explicit collision constraints rather than soft penalties. Both are local methods: they need a reasonable initial trajectory and return a local optimum, not the global one.

The trade-off against sampling-based planners is fairly clean. Sampling-based methods are probabilistically complete, meaning given infinite time they find a solution if one exists, but produce jagged paths and ignore costs other than path length. Optimization-based methods produce smooth, low-cost paths but get stuck in local minima around obstacles. Production stacks typically run RRT-Connect first to find a path, then hand the result to TrajOpt as a warm start for polishing.

What the geometric layer hands to §4.3

The output of this whole pipeline, IK to get `q_goal`, motion planning to get a path, shortcutting and smoothing, is a discrete sequence of joint configurations `(q_0, q_1, …, q_K)` with associated timestamps. This is a kinematic trajectory: it describes positions over time but says nothing about whether the robot's motors can actually generate those positions under gravity and inertia. Sending the trajectory directly to position-controlled joints often works fine for slow motions and stiff industrial arms; sending it to a torque-controlled compliant arm moving fast will produce trajectory tracking error of centimeters, more than enough to miss the block entirely.

Closing that gap, turning a kinematic trajectory into joint torques, is the inverse-dynamics problem of §4.3.

4.3 Inverse dynamics and computed-torque control

§4.2 ended with a kinematic trajectory: a sequence of joint configurations `(q_0, q_1, …, q_K)` with timestamps, smoothed and collision-free. That trajectory describes where the joints should be at each instant. It says nothing about what force the motors have to apply to put them there. Closing that gap is the inverse-dynamics problem, and the controller using its solution is called computed-torque control.

The gap exists for mechanical reasons. A robot arm isn't a free particle. Gravity pulls on every link. Lifting a payload changes the inertia the shoulder has to fight. Moving the elbow fast generates Coriolis forces on the wrist. Friction in the harmonic drive of joint three eats torque proportional to angular velocity. A position-commanded joint that ignores all of this works fine for a slow industrial arm with high-ratio gearboxes, since the gearbox absorbs the discrepancy. It doesn't work for a fast, low-inertia, torque-controlled arm like a Franka Panda or KUKA iiwa, precisely the class of robots most modern manipulation research runs on. For those, the controller has to ask what torque produces the desired acceleration, then send that torque.

The manipulator equation

Every textbook on robot dynamics arrives at the same equation, often called the manipulator equation or the equation of motion in joint space:

$$M(q)\,\ddot{q} + C(q,\dot{q})\,\dot{q} + g(q) + \tau_f(\dot{q}) = \tau$$

Each term is a vector in $\mathbb{R}^n$, where n is the number of joints. Reading left to right: $q, \dot{q}, \ddot{q} \in \mathbb{R}^n$ are the joint positions, velocities, and accelerations. $M(q) \in \mathbb{R}^{n \times n}$ is the mass matrix, also called the joint-space inertia matrix.

It's symmetric positive-definite, and its entries depend on the current configuration, since moving the elbow changes how much inertia the shoulder feels. `C(q, q̇) q̇` packs the Coriolis and centripetal terms, forces appearing because joints are moving with respect to each other. `g(q)` is the gravity vector, the torque each joint has to produce just to hold the arm static against gravity in configuration `q`. `τ_f(q̇)` collects friction and other velocity-dependent losses; it's notoriously hard to model precisely, and viscous and Coulomb terms are the usual cartoon version. `τ ∈ R^n` is the vector of joint torques the controller commands.

The whole equation is Newton's second law lifted to a chain of rigid bodies. The derivation, via Lagrangian or Newton-Euler mechanics, is a standard chapter in Spong et al. (2006), Craig (2005), and Lynch and Park (2017). What matters here is the use of the equation, not its derivation.

Inverse dynamics asks: given `q, q̇, q̈`, compute `τ`. That's a direct evaluation of the right-hand side, substitute the numbers, get the torque. It's the easy direction. Forward dynamics runs the opposite way: given `q, q̇, τ`, compute `q̈`. That requires inverting `M(q)` and is what physics simulators (MuJoCo, Bullet, Drake) do every step. Manipulation control is mostly inverse dynamics; physics simulation is mostly forward dynamics.

Computing inverse dynamics: RNEA in one paragraph

A naive implementation of `M(q) q̈ + C(q, q̇) q̇ + g(q)` constructs each matrix and multiplies, costing `O(n^3)` per call. The actual implementation everyone uses is the Recursive Newton-Euler Algorithm (RNEA), due to Luh, Walker, and Paul (1980) and developed in full generality by Featherstone (2008). RNEA computes `τ` directly without ever forming `M` or `C`, in two sweeps along the kinematic chain. The forward sweep propagates velocities and accelerations outward from the base to the end-effector, computing the linear and angular acceleration of each link's center of mass. The backward sweep propagates forces and moments inward, summing up the torque each joint must apply to produce that link's acceleration plus the loads transmitted by its children. The total cost is `O(n)`. For a 7-DOF arm, one RNEA call takes a few microseconds in C++, fast enough to run inside a 1 kHz control loop with thousands of cycles to spare. RNEA is what `pinocchio`, `RBDL`, and the Drake dynamics library compute under the hood; you almost never write the recursion yourself.

The computed-torque law

Suppose the motion planner has handed us a reference trajectory: `q_d(t), q̇_d(t), q̈_d(t)`, desired position, velocity, and acceleration at time `t`. The controller measures the actual `q(t), q̇(t)` and defines the tracking error:

```
e = q_d - q
ė = q̇_d - q̇
```

The computed-torque control law, sometimes called the inverse-dynamics control law, is:

```
τ = M(q) [ q̈_d + K_d ė + K_p e ] + C(q, q̇) q̇ + g(q) + τ_f(q̇)
```

`K_p` and `K_d` are diagonal positive-definite gain matrices. Read it piece by piece. The bracketed term is the desired acceleration plus a PD correction pulling the error toward zero. Multiplying by `M(q)` turns that desired acceleration into the torque required to produce it. Adding `C(q, q̇) q̇ + g(q) + τ_f(q̇)` cancels the gravitational, Coriolis, and frictional torques the arm already feels. The net effect is striking: substituting this `τ` into the manipulator equation and simplifying gives `ë + K_d ė + K_p e = 0`, a linear, decoupled, second-order error dynamics on every joint. The robot, which was a coupled nonlinear mechanical system, has been transformed into `n` independent linear springs. This trick is called feedback linearization, and it's one of the load-bearing ideas in classical robot control.

In practice the controller doesn't invert `M(q)` symbolically; it calls RNEA. Once with `q̈_d + K_d ė + K_p e` as the acceleration argument to get the feedforward-plus-linearizing torque, since RNEA returns the full right-hand side in one pass. A minimal pseudocode sketch:

```
def computed_torque(q, qd, q_des, qd_des, qdd_des, Kp, Kd):
    e   = q_des  - q
    ed  = qd_des - qd
    qdd_cmd = qdd_des + Kd * ed + Kp * e
```

```
    tau = rnea(q, qd, qdd_cmd)   # one O(n) call
    return tau
```

One RNEA call per timestep at 1 kHz is unproblematic on a modern CPU.

Why the model is never quite right, and what to do about it

Computed-torque control assumes you know `M, C, g, τ_f` exactly. You don't. The arm's link masses are known to maybe 1%, the inertia tensors to worse; friction runs wildly nonlinear and changes with temperature; the moment a payload gets attached, every parameter in `M(q)` shifts. The literature has three standard responses.

The first is not inverting the full model, keeping only the easy parts. PD-plus-gravity control uses

```
τ = K_p e + K_d ė + g(q)
```

dropping the inertial and Coriolis cancellation entirely. The arm tracks worse, especially at high speed, but the law stays robust: as long as the gravity model is approximately right and the gains are stable, the arm holds its trajectory acceptably. Most teaching robots and many ROS controllers ship this configuration by default.

The second is robust or adaptive control. Slotine and Li (1987) showed you can adapt the parameter estimates online: define a parameter vector `θ̂` containing the estimated masses, inertias, and friction coefficients, update `θ̂` according to a Lyapunov-derived rule each step, and use the current estimate inside the computed-torque law. Under reasonable conditions, tracking error and parameter error both converge. Adaptive control is elegant and was, for a decade, the right answer for high-performance arm control. It's now overshadowed in practice by learning the residual instead: estimate the model from data and let a small network correct what the parameterized model gets wrong.

The third response is impedance control (Hogan, 1985). Instead of forcing the arm to track a position trajectory exactly, command it to behave like a virtual spring-damper attached to the desired pose:

```
τ = J(q)^T [ K_imp (x_d - x) + D_imp (ẋ_d - ẋ) ] + g(q) + (model terms)
```

where `x = T(q)` is the end-effector pose and `J(q)` is the Jacobian. Now if the gripper hits something unexpectedly, it deflects rather than crashing. This is the control mode the Franka Panda exposes by default, and it's also what makes safe physical human-robot interaction possible. We revisit impedance control in Chapter 17 when discussing safety as a layer.

Operational-space control, briefly

Joint-space computed-torque control places gains on each joint independently, the wrong unit of analysis when the task is specified in Cartesian space, "move the gripper 5 cm forward, then press down with 5 N." The operational-space formulation of Khatib (1987) rewrites the same equations in end-effector coordinates:

```
F = Λ(q) ẍ_d + μ(q, q̇) + p(q)
τ = J(q)^T F + (null-space term)
```

where `Λ` is the task-space inertia matrix and `μ, p` are the Coriolis and gravity contributions projected into task space. The joint torques then get computed by multiplying the task-space force `F` by the Jacobian transpose. The redundant nullspace (extra DOF beyond the six needed for the task) gets filled by a secondary objective, joint-limit avoidance, manipulability maximization, or simply staying away from a configuration the engineer dislikes. Operational-space control is the standard formulation for humanoid whole-body control, and it underpins the dual-system architectures we'll see in Chapter 14, where a high-level VLM names a task pose and a low-level operational-space controller realizes it.

Where this connects to the rest of the book

A learned policy can take responsibility for as much or as little of the manipulator equation as the engineer assigns it. The spectrum is wide, and worth seeing as a spectrum.

At one extreme, OpenVLA and Diffusion Policy emit joint positions or end-effector poses and rely on a downstream position controller, typically PD-plus-gravity, sometimes full computed-torque, to actually produce the torques. The

learned model never sees τ. This is the default in nearly all VLA literature, and it works because the underlying classical controller is doing the dynamics-aware work.

A step further in, RL controllers for legged locomotion (ANYmal, Cassie, Atlas) typically emit joint torques directly. They learn an end-to-end policy whose output is τ, and the manipulator equation appears only inside the simulator they trained in. There's no inverse-dynamics layer in the deployed stack; the policy has internalized whatever inertial and gravitational compensation the task requires.

Further still, residual and hybrid approaches treat classical computed-torque as a strong prior and learn only the correction. A common pattern in fine manipulation and contact-rich assembly: classical impedance control provides 80% of the right torque, and a small learned network adds a residual to handle hard-to-model contact. This is one of the more practical recipes for getting a learned policy into a production robot quickly.

§4.3 earns its own section in a book mostly about VLAs because even when the controller is learned, the language the rest of the robotics stack uses to describe its job is still the language of the manipulator equation. "We had a tracking error of 3 cm at the wrist when the payload doubled" isn't a sentence you can debug without `M(q)`. A foundation policy that produces beautiful joint targets is still shipping torques through a classical controller most of the time, and when that controller misbehaves, the only useful description of the misbehavior stays in inverse-dynamics terms.

The next section, §4.4, takes the three layers we've now built, symbolic, geometric, dynamic, and asks where each is still load-bearing in robots actually shipping in 2026, even as learned components take over more of the stack.

4.4 Where classical methods are still load-bearing in modern robots

The previous three sections built a three-layer cake. At the top, a symbolic planner produces a sequence of named actions. In the middle, a geometric layer turns each action into a joint-space trajectory. At the bottom, an inverse-dynamics controller turns the trajectory into torques. A reader who's been following along might reasonably ask the next question: in 2026, with VLAs trained on a million teleoperated episodes and $\pi 0$ outputting smooth continuous actions at 50 Hz, how much of that cake is still in the kitchen?

The honest answer is most of it. Learned components have grown in, mostly at the top and the middle, but they've rarely replaced a classical layer outright. They sit on top of it, beside it, or inside it as a residual term. This section catalogues where each of the three classical layers is still doing real work in deployed systems, and where it's being squeezed out. The taxonomy matters because the rest of the book keeps returning to it. When Chapter 14 discusses dual-system architectures, the "low-level" system is almost always one of the classical controllers from §4.3. When Chapter 16 discusses fine-tuning, the cheapest gains often come from leaving the geometric and dynamic layers alone and training only the perception-to-pose head. A learned action model is rarely a learned stack; it's a learned part of a stack that still has classical bones underneath.

The symbolic layer: alive and well at the top

The cleanest survivor is the top layer. Almost no commercial robot executes a long-horizon task as a single forward pass through a learned policy. Warehouse picking, restaurant assembly, surgical assistance, and household tidying all share a structure: a high-level component decides what to do next, and a low-level component decides how. That high-level component is, structurally, a symbolic planner. The implementation has changed, increasingly it's an LLM rather than Fast Downward, but the contract stays the same. The output is a discrete action drawn from a finite alphabet of skills, optionally with arguments, and the downstream system then has to ground each call in geometry.

SayCan (arXiv:2204.01691) is the cleanest published example. The high level is a language model that, given a goal like "bring me the sponge," produces a candidate sequence of skill calls drawn from a predefined library, `find(sponge)`, `pick(sponge)`, `bring_to(user)`. The model also scores each candidate using a value function estimating how feasible the skill is from the current state. The output is a STRIPS-flavored plan, just with the planner replaced by a much better engine. Code as Policies (arXiv:2209.07753) makes the same substitution more literal: the LLM emits Python calling a library of parameterized skills, and the Python is the plan. In both cases the shape of the top layer is exactly what §4.1 described.

The serious modern engineering happens in task and motion planning (TAMP), where the symbolic and geometric layers get interleaved rather than stacked. PDDLStream (Garrett, Lozano-Pérez, and Kaelbling, 2020) lets a symbolic planner call out to motion-planning samplers mid-search: the planner can't commit to `pick(block_a)` until a geometric subroutine has confirmed there's a collision-free grasp, and once the grasp is found, its result feeds back into the symbolic state. This kind of architecture is the standard top-of-stack for household robots that need to actually open a drawer and find the spoon inside it. The symbolic layer is doing logical bookkeeping that no end-to-end network has yet been shown to handle reliably for tasks longer than a handful of steps.

The squeeze on this layer comes from below, in two ways. VLAs are getting better at short-horizon multi-step tasks; $\pi 0$ (arXiv:2410.24164) executes "fold this shirt" or "set the table" in one shot, where five years ago each would have needed three or four symbolic steps. And LLM-based planners are eroding the strict distinction between "symbolic plan" and "natural-language script." A robot's task list is increasingly a list of English sentences fed to a VLA, with the LLM acting more like a teacher than a logician. But even with those changes, some discrete planner decides which sentences come in what order, and that planner still inherits its vocabulary from STRIPS.

The geometric layer: hidden but indispensable

The middle layer is the one most newcomers underestimate. VLA papers tend to elide it. A model "outputs end-effector poses" and the reader imagines the model is doing the geometry. It isn't. Almost universally, the VLA outputs a target pose at 5 to 50 Hz, and a classical IK solver, a workspace bounding box, and a short-horizon trajectory generator sit underneath, turning that target into joint commands while keeping the motion smooth and collision-free.

OpenVLA (arXiv:2406.09246) is a representative case. The model emits seven numbers, six for the delta end-effector pose, one for the gripper. Those numbers feed into a server-side controller that does the IK, clips the result against the robot's joint limits, blends it with the previous command for smoothness, and forwards a joint-space target to the arm's low-level controller. The model doesn't know what "unreachable" means. If the user puts the cup on the floor and asks the robot to grab it, the IK solver, usually TRAC-IK or whatever ships with the manufacturer's SDK, is the component that returns "no solution," and the VLA never sees the failure as a learning signal during deployment.

The same is true of motion planning. A learned policy emitting poses at 5 Hz is producing waypoints, not trajectories. Real motion planning, collision-free path, smooth interpolation, joint-limit projection, self-collision avoidance, is happening somewhere, and unless the research paper explicitly says "we replaced motion planning with the network," that somewhere is a classical planner. RRT-Connect or OMPL is still running. The most ambitious systems that try to learn motion planning end to end, some legged-locomotion controllers among them, do so because the geometry is simple (a few-DOF foot trajectory) even though the contact dynamics aren't. For manipulation in cluttered environments, classical motion planning remains the default. A clear way to see this is inspecting the GR00T N1 (arXiv:2503.14734) reference stack: the foundation model produces high-level intent and short-horizon action chunks, but the deployment-time controller still includes a collision-aware geometric layer that vetoes commands which would put a link through a table.

Two places where the geometric layer is genuinely shrinking are free-space motions and learned contact-rich manipulation. For free-space reaching, modern VLAs produce trajectories smooth enough on their own that downstream geometry reduces to safety checking. For contact-rich tasks, insertion, wiping, peg-in-hole, the model learns the contact phase of the motion that classical geometric planners struggle with anyway, while the approach phase is still a planned trajectory. Even here the cake stays layered, just with a thinner geometric slice in the middle.

The dynamic layer: torques are still classical

The bottom of the stack is the most strongly classical of all, and it's likely to stay that way for the foreseeable future on manipulators. Once a joint target comes down from the upper layers, something has to turn it into motor current, and that something is, in every commercially deployed manipulator the authors are aware of in 2026, a classical controller: PD-plus-gravity, computed-torque, or impedance control, in the language of §4.3.

The Franka Panda is the canonical example. The arm ships with several control interfaces, joint position, joint velocity, joint impedance, Cartesian impedance, torque, and every one of them, torque included, gets consumed by the manufacturer's onboard controller, which applies its own dynamics-aware compensation before driving the

motors. A research user who selects the torque interface still has the manufacturer's gravity compensation, friction model, and joint-limit shielding active underneath. They haven't replaced the dynamic layer; they've set the high-level term in the manipulator equation and let the low-level term handle the rest. This is also why papers report "torque-level control" of a Franka with a learned policy without ever actually estimating `M(q)` in the learned network. The manufacturer's controller is doing it already.

The strong exception is legged robotics. RL-trained controllers for ANYmal, Cassie, and Atlas typically emit joint torques (or position targets that drive a high-bandwidth, low-impedance underlying joint PD), and the policy has internalized the inertial and gravitational terms during training in simulation. This works for legs and not for manipulator end-effectors because legged-locomotion controllers run in closed loop with proprioception at 200 to 1000 Hz on a relatively low-dimensional output, with a clear and easily simulated reward, and crucially with no contact specification: the policy chooses where to step, but the contact itself is a constraint the physics imposes rather than a target the controller has to enforce. Manipulators have to push on objects with specified forces; legs only have to push on the ground.

Where the dynamic layer is being augmented but not replaced is the residual pattern mentioned at the end of §4.3: classical impedance control provides 80% of the right torque, and a small network learns the residual that handles deformable objects, cloth, contact transitions, or fast in-hand reorientation. Production deployments of such residuals at companies like Covariant, Figure, and 1X aren't heavily documented in the literature, but the architectural pattern stays consistent across the few systems whose papers do describe it.

A summary table you can keep in your head

If you have to argue with someone about whether classical robotics is "dead," the right shape of the argument runs layer by layer.

The symbolic layer stays mostly classical, increasingly LLM-driven, with the interface (named skills with arguments) unchanged since STRIPS. It gets replaced wholesale by a learned policy only on tasks short enough to fit in a single VLA call. The geometric layer stays almost universally classical inside deployed manipulators in 2026: learned policies emit poses, and IK and motion planning sit underneath. It's genuinely shrinking only for short-horizon contact-rich motions and some legged controllers. The dynamic layer stays classical, full stop, for manipulators. RL has taken the dynamic layer on legged platforms because the structure of the task is friendly to torque-level policies, and residual learning is the active research frontier on top of classical control rather than a replacement for it.

The reason this matters for the rest of the book is that, when we discuss what a foundation action model outputs in Chapters 11 through 14, we're almost always discussing what enters the geometric layer from above. The picture of "VLA to motor" you may have built up while reading the OpenVLA paper is missing two layers in the middle. They're still there, still being maintained by the same kind of engineer who maintained them in 2005, and they're still where most production failures live.

§4.5 closes Chapter 4 with a summary that consolidates the layered view and previews how each subsequent part of the book attacks the layers from below.

4.5 Summary

Chapter 4 traced the lineage. The four sections walked down the classical stack, symbolic plans at the top (§4.1), geometric trajectories in the middle (§4.2), inverse-dynamics torques at the bottom (§4.3), then a frank accounting of which of those layers is still doing real work inside a deployed 2026 robot (§4.4). The chapter sits at a particular hinge in the book: it's the first chapter of Part 2 and the one where the four-family taxonomy from §1.4 starts getting unpacked rather than just asserted. This summary collects the load-bearing ideas in one place and flags which ones Chapters 5 through 14 lean on hardest.

The four ideas worth carrying forward

A classical action model is a function from a structured state to a structured action. §4.1 made the case for symbolic planning by reducing STRIPS to three pieces, predicates, action schemas, planning problems, and showing that every

PDDL domain sits on top of those three. What newcomers miss is that the interface of a symbolic action model survives even when the planner inside gets replaced. SayCan (arXiv:2204.01691) and Code as Policies (arXiv:2209.07753) both swap a heuristic-search planner for a language model, but the output is still "a named action with arguments, drawn from a finite alphabet." When Chapter 14 dissects dual-system architectures, the top system emits that same kind of object, only the implementation has moved from Fast Downward to a 70-billion-parameter transformer. The shape of the interface is what matters; the choice of engine behind it is increasingly an engineering preference rather than a research commitment.

Geometry is the layer newcomers most often forget exists. §4.2 spent its pages on inverse kinematics, RRT-Connect, and the difference between sampling-based and optimization-based motion planners, because the failure mode of reading a learned-policy paper in isolation is imagining the model itself is doing the geometry. It isn't. The seven numbers OpenVLA (arXiv:2406.09246) emits at 5 Hz are a target pose delta; the IK solver, the joint-limit clipper, the workspace bounding box, and the short-horizon smoother sitting underneath the model are the geometric layer, and they're doing the same kind of work TRAC-IK and OMPL were doing in 2015. The chapter's most useful diagnostic, if a VLA paper doesn't name its IK solver, assume TRAC-IK; if it doesn't name its motion planner, assume the manufacturer's default, is what keeps you from being surprised when a fine-tune that "should have worked" fails because the target pose sat 1.5 cm outside the dexterous workspace.

Inverse dynamics is the layer that has barely moved in forty years and probably won't move much in five. §4.3 worked through the manipulator equation $M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau$ and the computed-torque, PD-plus-gravity, and impedance-control patterns sitting on top of it. This layer is so durable because the structure of the problem is favorable to derivation: the dynamics model is known up to parameters, those parameters can be identified, and the resulting controller has stability properties that matter when something heavy is moving fast. Legged locomotion is the visible exception, since RL-trained controllers on ANYmal, Cassie, and Atlas do emit torques and have internalized the inertial terms during training, but for manipulators in 2026, the bottom of every commercial stack the authors know of is still a classical controller. Residual learning (Hogan 1985's impedance plus a learned correction) is the research frontier on top of classical control, not its replacement.

The right argument about classical methods runs layer by layer, not yes or no. §4.4 was the chapter's punch line, worth repeating here because it's the single most common rhetorical trap in the field. "Is classical robotics dead" is a malformed question. The question that admits an answer is: in this layer of this stack, what fraction of the work is a classical method doing versus a learned one, and which way is the line moving? The §4.4 breakdown, symbolic mostly classical with LLM-flavored variants in the top slot, geometric overwhelmingly classical with thinning slices for short-horizon contact-rich motion, dynamic essentially fully classical on manipulators and learned on legs, is the form of the answer the rest of the book keeps returning to. Whenever a paper claims to have "replaced the classical pipeline," the right reflex is asking which layer it actually replaced and whether the other two are still there. They almost always are.

What you should be able to do now

Four concrete capabilities, in roughly increasing order of how much the rest of the book relies on them.

You should be able to write a small PDDL domain and run a planner on it. The chapter's hands-on directory has a five-block tabletop with `pick-up`, `put-down`, `stack`, and `unstack` schemas; you should be able to read that domain without reaching for documentation and extend it with a fifth schema, `push`, say, or `swap`, without breaking the rest. The skill is small but load-bearing: every language-model planner in Part 4 (SayCan, Code as Policies, the top half of GR00T N1, arXiv:2503.14734) produces something readable as a PDDL plan with the predicates implicit, and being able to write the explicit version makes the implicit one intelligible.

You should be able to derive a 6-DOF inverse-kinematics solution analytically for a simple geometry, or pick a numerical IK solver off the shelf for a harder one, and know which case you're in. §4.2 named the deciding factor: closed-form IK exists when the wrist is spherical and the geometry is simple enough for the Pieper criterion to apply, and that's when you reach for it. Otherwise you call damped least squares or TRAC-IK and accept the solver may return nothing on some queries. This matters because Chapter 16's fine-tuning recipes assume you can stand up an IK layer underneath the VLA without ceremony, and Chapter 17's deployment chapter assumes you can debug "the model outputs a perfectly reasonable pose, but the arm does not move" by checking the IK return code rather than the network.

You should be able to read a torque-control paper and identify which terms of the manipulator equation the controller computes explicitly, which it's learning, and which it's letting the manufacturer's onboard firmware handle. The Franka Panda case from §4.4 is the canonical illustration: a research user who sends a "torque-level" command is, in fact, setting the high-level term while the manufacturer's gravity compensator, friction model, and joint limits operate underneath. A paper reporting "we trained an RL policy to output joint torques on a Franka" isn't training a controller that produces motor current; it's training a controller that produces the top of the manipulator equation, with everything below it already classical. Naming what's learned versus what's provided is the difference between reading the paper at face value and reading it correctly.

You should be able to draw the three-layer cake for any deployed robot in the literature and label, for each layer, classical versus learned. Pick any system, OpenVLA on a WidowX, π0 on a bimanual manipulator, GR00T N1 on a humanoid, Aloha on a teleop rig, and the cake-labeling exercise tells you what the system actually is. OpenVLA: classical top (task is implicit in the language prompt), learned middle-top (network emits poses), classical middle-bottom (IK and trajectory generation), classical bottom (joint-level control). π0: similar, but with a flow-matching middle-top emitting action chunks rather than single poses. GR00T N1: LLM-flavored top, learned middle, classical bottom. The exercise is mechanical once you can do it for one system, and it's the skill the rest of Part 4 quietly assumes you have.

Where the chapter has set up the rest of the book

Chapter 4 hands off to the next three chapters of Part 2 along a specific path. Chapter 5 picks up the reward slot from §1.2, the training-signal slot that classical action models leave empty, and develops MDPs and reinforcement learning as the family that fills it. Chapter 6 does the same for demonstrations and behavior cloning, the family filling that same slot with imitation data instead of reward. Chapter 7 is the deep-RL chapter, where the torque-level controllers from §4.3 meet the function-approximation ideas from §3.1. By the time you finish Part 2, you'll have seen the three other ways of producing an action, reward-driven, imitation-driven, and the deep-RL hybrid, and the classical methods from this chapter will have moved from being "the way robots are programmed" to being "the layer underneath the way modern robots are programmed."

Two specific forward references from this chapter are worth naming again, since they recur. The PDDL interface from §4.1, named skills with arguments drawn from a finite alphabet, is the interface between Chapter 14's high-level system and its low-level system in a dual-system architecture. The residual-learning pattern from §4.3 and §4.4, classical controller plus a small learned correction, is the architectural pattern Chapter 16 reaches for first when the question is how to add a small amount of learning to an existing classical stack without breaking it. Both ideas appear later under different names, and the §4 versions are the small, readable prototypes.

The chapter hasn't set up two things you might have expected. It hasn't covered task-and-motion planning in depth; PDDLStream (Garrett, Lozano-Pérez, and Kaelbling, 2020) appeared in §4.1 as a named reference rather than a worked example, and the kind of joint symbolic-geometric search TAMP performs is its own subfield with its own dedicated literature, one we point to rather than reproduce. It also hasn't covered the operational-space formulation (Khatib, 1987) in detail; it appeared in §4.3 as one of the named patterns, but the full treatment of force control belongs in a manipulation textbook, not this one. Both omissions are deliberate. The aim of Chapter 4 was giving you the classical vocabulary the rest of the book reuses, not turning you into a planning-systems engineer or a force-control engineer.

Part 2 starts here, and the four-family taxonomy from §1.4 is now half-explained: the classical family in Chapter 4, the RL family next, the imitation family after that, the foundation/VLA family in Part 4. Each subsequent chapter adds a single family to the working vocabulary, and Chapter 4's contribution is the family that defines the interfaces, the named skills, the joint-space trajectories, the joint torques, that every later family ultimately has to produce something compatible with. Whatever a VLA outputs in 2030, an IK solver and an inverse-dynamics controller are still likely to be the last two functions called before a motor sees current.

§4.x closes Chapter 4 with one hands-on exercise, running Fast Downward on a five-block PDDL domain, then wiring its symbolic plan to a TRAC-IK-driven pick-and-place executor in PyBullet, and the full reading list for the chapter.

4.x Hands-on exercise + chapter references

Chapter 4 was the classical-lineage chapter; the exercise is where you build a small version of the three-layer cake on your laptop and stack one block on top of another with no neural network anywhere in the loop. The four drills below take a combined three hours on a laptop CPU. The point is leaving Chapter 4 with a working symbolic-to-geometric-to-dynamic pipeline on disk, so that when Chapters 11 through 14 start replacing the top and middle of the cake with learned components, you have something to replace into.

Exercise 4.x.1 — A five-block PDDL domain and a planner call

Create a directory `drills_ch4/` with two files: `blocksworld.pddl` (the domain) and `problem_5.pddl` (the instance). The domain encodes the standard Blocksworld formulation from §4.1, predicates `(on ?x ?y)`, `(on-table ?x)`, `(clear ?x)`, `(holding ?x)`, `(arm-empty)`, and four action schemas: `pick-up`, `put-down`, `stack`, `unstack`. Write the domain from memory and check it against §4.1 afterwards. If you can't reproduce the `stack` schema's add and delete lists without peeking, you haven't yet internalized what an action schema is.

The problem file describes five blocks `a` through `e` arranged as two towers, `(a on b)`, `(b on c)`, `(c on-table)`, `(d on e)`, `(e on-table)`, plus the matching `clear` and `arm-empty` predicates, and a goal that stacks all five into a single column with `a` on top and `e` on the bottom: `(on a b) (on b c) (on c d) (on d e) (on-table e)`. Fast Downward should return one of several plans of length 12.

Install Fast Downward (Helmert, 2006), from source, via the dockerized variant, or via the `pyperplan` wrapper for a simpler alternative, and run it on the two files. Save the returned plan to `plan.txt`. The deliverable is the plan plus a one-line note recording which heuristic the planner used (`seq-opt-lmcut` and `lama-first` are the two defaults worth knowing). Runtime should be well under a second. If the planner reports `unsolvable`, your goal or initial state is inconsistent, most often a missing `clear` predicate, and §4.1's discussion of the closed-world assumption is the place to re-read.

Wall clock: about thirty minutes including the Fast Downward install.

Exercise 4.x.2 — Wire the plan to a PyBullet pick-and-place executor

Copy the plan from Exercise 4.x.1 into a new Python file `execute_plan.py`. Its job is taking the plan and producing joint-space motion on a simulated Franka Panda in PyBullet. The architecture is the §4.5 three-layer cake in code form:

1. Symbolic layer (already done): the plan from 4.x.1, read in as a list of `(action_name, arguments)` tuples.
2. Geometric layer (this exercise): a function `solve_action(action, world_state)` that, for each symbolic action, computes a sequence of Cartesian waypoints (pre-grasp pose, grasp pose, lift pose, transport pose, place pose, retract pose), passes each waypoint through TRAC-IK to obtain joint targets, and concatenates the joint targets into a trajectory. Use the TRAC-IK Python bindings (Beeson and Ames, 2015) if installed; otherwise fall back to the damped-least-squares IK from §4.2 with a damping factor of 0.05 and a maximum of 200 iterations. RRT-Connect (Kuffner and LaValle, 2000) is overkill for tabletop blocksworld, since straight-line motion in joint space between IK solutions is sufficient, but drop OMPL in if you have it handy.
3. Dynamic layer (PyBullet provides it): set joint targets with `setJointMotorControlArray` in position-control mode. PyBullet's internal PD-plus-gravity controller does the §4.3 work for you; the point of this exercise is not implementing computed-torque from scratch, but noticing that the simulator is silently providing the bottom layer of the cake.

The world is a flat table at `z=0`, five colored 4 cm cubes whose initial positions match `problem_5.pddl`, and a Franka mounted at the table edge. PyBullet ships the URDF at `pybullet_data/franka_panda/panda.urdf`. A skeleton for the execution loop, illustrative rather than exhaustive:

```
plan = read_plan("plan.txt")
world = init_pybullet_world(blocks=["a", "b", "c", "d", "e"])
arm = load_panda(world)
ik = TracIKSolver(arm.urdf, "panda_link0", "panda_hand")
```

```
for action_name, args in plan:
    waypoints = waypoints_for(action_name, args, world)
    for pose in waypoints:
        q_target = ik.solve(pose, q_seed=arm.current_q())
        if q_target is None:
            raise RuntimeError(f"IK failed for {action_name} at {pose}")
        arm.move_to(q_target, duration=1.0)
    world.apply_symbolic_effects(action_name, args)
```

Run the script and watch the arm execute the twelve-action plan in about thirty seconds of wall clock simulation time. Save a screenshot of the final state, a single tower with `a` on top, to `final_state.png`. Save the joint trajectory to `trajectory.npz`; you'll reuse it in Exercise 4.x.3.

The most common failure is IK returning `None` for the pre-grasp pose of `c`, because the dexterous workspace of the Panda doesn't quite reach a block at table height directly below the base. The §4.2 fix is lowering the table by 5 cm or mounting the arm 10 cm higher. If the IK succeeds but the gripper passes through an adjacent block on approach, you've rediscovered the problem PDDLStream (Garrett, Lozano-Pérez, and Kaelbling, 2020) was written to solve: the symbolic planner doesn't know about geometric infeasibility, and a true TAMP system would interleave the two searches. We don't implement that here; leaving the bug visible is the lesson.

Wall clock: about ninety minutes including TRAC-IK installation.

Exercise 4.x.3 — Replace the bottom layer with explicit computed-torque

Copy `execute_plan.py` to `execute_plan_torque.py` and change one thing: instead of using PyBullet's position controller, drive the arm in pure torque mode (`setJointMotorControlArray(... , controlMode=p.TORQUE_CONTROL, forces=tau)`) and compute `tau` explicitly from the §4.3 computed-torque law

$$\tau = M(q)(\ddot{q}_d + K_d(\dot{q}_d - \dot{q}) + K_p(q_d - q)) + C(q, \dot{q})\dot{q} + g(q).$$

PyBullet exposes the inertia matrix and the bias terms through `calculateMassMatrix` and `calculateInverseDynamics`, so you don't have to derive the Panda's dynamics from URDF by hand. Choose `K_p = 100` and `K_d = 20` on every joint as a starting point. The reference trajectory `q_d(t)` is the trajectory you saved in 4.x.2; numerically differentiate it once for `q̇_d` and twice for `q̈_d` with a simple central difference, or fit a cubic spline first if the differentiated signal comes out too noisy.

The arm executes the plan again, with modest overshoot at each waypoint that decays in under half a second. Plot `q(t)`, `q_d(t)`, and `tau(t)` for joint 4 (the elbow, which carries the largest gravity load) and save as `joint4_torque.png`. The plot should make three things visible: `tau(t)` isn't zero when the arm is stationary (the static gravity term), peaks in `tau(t)` line up with the lift-and-transport phases (dynamic torques scale with acceleration), and tracking error grows when `K_d` is reduced; try `K_d = 5` and watch the arm wobble. That wobble is what §4.3 cited as the motivation for matching damping coefficients to the inertia matrix, and the residual-learning pattern from §4.4 is what a research group reaches for when the wobble persists even with well-tuned gains.

This is the exercise where the three-layer cake stops being a metaphor. You've written the geometric and dynamic layers explicitly; the symbolic layer is the text file from 4.x.1; the learned layer is the empty slot the rest of the book fills in.

Wall clock: about forty-five minutes.

Exercise 4.x.4 — A PDDL diff against a SayCan transcript

Open the SayCan paper (Ahn et al., 2022, arXiv:2204.01691) to one of the appendices listing a full kitchen-task transcript; the "bring me a snack" examples typically run several pages. Pick one transcript and, on paper or in a

text file, transcribe the language-model output into PDDL. Each numbered step the model produces is one ground action; figure out which predicates would have to hold in the precondition and which would be added or deleted.

You'll discover three things. SayCan plans are short, four to eight steps, well within Fast Downward's reach. The predicate set runs much larger than blocksworld's (`in-hand`, `at-location`, `is-open`, `contains`), since a real kitchen has more state than five blocks. And several SayCan steps will be hard to express as STRIPS schemas at all without admitting the language model is doing implicit hierarchical decomposition that the symbolic formalism would express with HTN-style methods rather than pure STRIPS.

The deliverable is the transcribed PDDL plus a half-page note on which steps mapped cleanly, which didn't, and what would have been needed to make them clean. The §4.5 claim that the interface of a symbolic action model survives, with only the engine having moved, holds true in a limited sense, and you should have first-hand experience with where that limit sits. Chapter 14 returns to this when it dissects the high-level system of a dual-system architecture.

Wall clock: about thirty minutes.

Chapter 4 reading list

The works below are the ones cited across §4.1 through §4.5, grouped by purpose. Full bibliographic entries for everything cited in the whole book live in Appendix E.2; this list is just the chapter-local subset.

Symbolic planning (§4.1)

- Fikes, R. E., & Nilsson, N. J. (1971). "STRIPS — A New Approach to the Application of Theorem Proving to Problem Solving." *Artificial Intelligence* 2(3–4). The paper §4.1 starts from.
- McDermott, D., et al. (1998). "PDDL — The Planning Domain Definition Language." AIPS-98 Planning Competition. The lingua franca §4.1 recommends learning first.
- Helmert, M. (2006). "The Fast Downward Planning System." *JAIR* 26. The reference planner used in Exercise 4.x.1.
- Hoffmann, J., & Nebel, B. (2001). "The FF Planning System." *JAIR* 14. The other heuristic-search planner worth knowing, and the lineage behind Fast Downward's `lama-first` configuration.
- Garrett, C. R., Lozano-Pérez, T., & Kaelbling, L. P. (2020). "PDDLStream — Integrating Symbolic Planners and Blackbox Samplers via Optimistic Adaptive Planning." ICAPS. The TAMP reference §4.1 and Exercise 4.x.2 both point to.
- Ahn, M., et al. (2022). "Do As I Can, Not As I Say — Grounding Language in Robotic Affordances." (SayCan.) arXiv:2204.01691. The LLM-as-symbolic-planner system §4.1 and Exercise 4.x.4 use.
- Liang, J., et al. (2022). "Code as Policies — Language Model Programs for Embodied Control." arXiv:2209.07753. The companion LLM-planner paper §4.4 cites.

Geometric actions: IK and motion planning (§4.2)

- Craig, J. J. (2005). *Introduction to Robotics — Mechanics and Control*, 3rd ed. Pearson. The standard manipulator-kinematics textbook.
- Lynch, K. M., & Park, F. C. (2017). *Modern Robotics — Mechanics, Planning, and Control.* Cambridge University Press. The modern geometric-algebra-flavored alternative to Craig.
- LaValle, S. M. (2006). *Planning Algorithms.* Cambridge University Press. The reference textbook for motion planning.
- Kavraki, L. E., Švestka, P., Latombe, J.-C., & Overmars, M. H. (1996). "Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces." *IEEE T-RA* 12(4). The PRM paper.
- Kuffner, J. J., & LaValle, S. M. (2000). "RRT-Connect — An Efficient Approach to Single-Query Path Planning." ICRA. The single-query planner §4.2 recommends as the default.
- Karaman, S., & Frazzoli, E. (2011). "Sampling-based Algorithms for Optimal Motion Planning." *IJRR* 30(7). RRT* and PRM*.
- Ratliff, N., Zucker, M., Bagnell, J. A., & Srinivasa, S. (2009). "CHOMP." ICRA. The optimization-based planner §4.2 contrasts with sampling.

- Beeson, P., & Ames, B. (2015). "TRAC-IK — An Improved Inverse Kinematics Solver." Humanoids. The numerical IK solver used in Exercise 4.x.2.
- Şucan, I. A., Moll, M., & Kavraki, L. E. (2012). "The Open Motion Planning Library." *IEEE RAM* 19(4). The OMPL reference; the planner every real ROS stack reaches for.

Inverse dynamics and control (§4.3)

- Spong, M. W., Hutchinson, S., & Vidyasagar, M. (2006). *Robot Modeling and Control.* Wiley. The textbook §4.3 follows for the manipulator equation derivation.
- Featherstone, R. (2008). *Rigid Body Dynamics Algorithms.* Springer. The reference for recursive algorithms, RNEA, CRBA, that any real inverse-dynamics implementation uses.
- Luh, J. Y. S., Walker, M. W., & Paul, R. P. C. (1980). "On-Line Computational Scheme for Mechanical Manipulators." *ASME J. Dynamic Systems* 102(2). The original RNEA paper.
- Khatib, O. (1987). "A Unified Approach for Motion and Force Control of Robot Manipulators — The Operational Space Formulation." *IEEE J. Robotics and Automation* 3(1). The operational-space reference.
- Hogan, N. (1985). "Impedance Control — An Approach to Manipulation." *ASME J. Dynamic Systems* 107(1). The impedance-control paper §4.3 and §4.4 cite as the residual-learning antecedent.
- Slotine, J.-J. E., & Li, W. (1987). "On the Adaptive Control of Robot Manipulators." *IJRR* 6(3). The adaptive-control reference for when the dynamic parameters aren't perfectly known.

Where classical methods still live in modern stacks (§4.4)

- Kim, M. J., et al. (2024). "OpenVLA — An Open-Source Vision-Language-Action Model." arXiv:2406.09246. The §4.4 worked example for what the seven numbers per step actually feed into.
- Black, K., et al. (2024). "π0 — A Vision-Language-Action Flow Model for General Robot Control." arXiv:2410.24164. The continuous-action VLA §4.4 contrasts with OpenVLA's discrete tokens. Chapter 13 is the full treatment.
- NVIDIA (2025). "GR00T N1 — An Open Foundation Model for Generalist Humanoid Robots." arXiv:2503.14734. The humanoid dual-system example §4.4 cites; Chapter 14 unpacks it.
- Padalkar, A., et al. (2023). "Open X-Embodiment — Robotic Learning Datasets and RT-X Models." arXiv:2310.08864. The dataset every §4.4 modern-stack reference was trained on. Chapter 12 is the full treatment.

Chapter summary

Chapter 4 was the classical-action chapter, sitting at the hinge between Part 1 (the foundations) and Part 2 (the lineage that produced VLAs). You can now write a small PDDL domain and run Fast Downward on it. You can stand up a tabletop IK-plus-motion-planning layer that takes a symbolic plan and turns it into joint-space motion on a simulated Franka. You can write the manipulator equation and the computed-torque law without looking them up, and explain which terms PyBullet, the manufacturer's onboard firmware, and the application code are each responsible for. And you can read any modern VLA paper and draw its three-layer cake, labeling each layer classical or learned. Chapter 5 begins the next chapter of Part 2, reinforcement learning, the family that fills the training-signal slot the classical models leave empty, and the controllers from §4.3 return there as the policies an MDP optimizes over.

Chapter 5 — Learning from rewards: MDPs and reinforcement learning

5.1 States, actions, rewards, and policies

Chapter 4 ended with a complaint and a promise. The complaint was that classical action models leave the training-signal slot of the §1.2 anatomy empty: a STRIPS planner knows what its actions do, but only because somebody wrote the effects by hand, and it has no way to learn that one sequence of actions was better or worse than another. The promise was that the next three chapters of Part 2 would each fill that empty slot with a different signal. Chapter 5's signal is reward.

The mathematical object that lets you talk about reward properly is the Markov decision process, or MDP. Almost every reinforcement-learning paper of the last forty years opens by writing down an MDP tuple in its second sentence, and this section spends ten pages on what those five letters mean, because the rest of the chapter, most of Chapter 7, and the offline-RL background you'll need in Chapter 12 stay unreadable until you can read the tuple without translating in your head. The section also makes the case that the MDP is a modeling choice, not a fact about the world, and that the gap between a clean MDP on paper and a messy robot in a lab is the source of more wasted GPU-hours than any other modeling decision in the field.

The five-tuple

An MDP is a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$. Five letters, each of which has been the subject of its own subfield. Take them one at a time.

$\mathcal{S}$ is the state space, the set of all configurations the world can be in. For the canonical 4×4 gridworld appearing in every RL textbook from Bellman (1957) onward, $\mathcal{S}$ is the set of sixteen cells, with one cell marked as the goal and one or two marked as obstacles. For a tabletop manipulation task with a 7-DOF Franka arm and one block, the state might be the 14-vector of joint positions and velocities concatenated with the block's 6-DOF pose, so $\mathcal{S} \subseteq \mathbb{R}^{20}$. For an end-to-end pixels-in policy like OpenVLA (arXiv:2406.09246), the "state" the policy sees is a 224×224×3 image, and whether that image is really a Markov state is a question we return to in §5.5.

$\mathcal{A}$ is the action space. In a discrete-action MDP the action space is a finite set: `{up, down, left, right}` for the gridworld, `{open-gripper, close-gripper, move-to-pose}` for a high-level skill MDP. In a continuous-action MDP it's a subset of $\mathbb{R}^n$: the 7-vector OpenVLA emits, the 8-DOF (joint torques plus gripper) command an SAC controller produces, the chunk of 16 poses Diffusion Policy outputs at once. The same physical robot can be described as either a discrete-action or a continuous-action MDP depending on what the engineer chooses to expose as $\mathcal{A}$, and that choice is a major architectural commitment. Chapter 7 lives mostly in continuous action spaces; this chapter, for clarity, stays discrete.

P is the transition function. Given that the agent is in state s and takes action a, $P(s' \mid s, a)$ is the probability of landing in state s'. For a deterministic gridworld, P collapses onto a single s' per (s, a) pair. For a noisy gridworld, one with "slippery" cells where `up` moves the agent up with probability 0.8 and sideways with probability 0.1 each, P becomes a non-trivial distribution. For a real robot, P encodes contact dynamics, motor backlash, sensor noise,

and the friction of the table the block sits on. It's unknown in almost every interesting case, and the central problem statement of reinforcement learning is "learn a good policy without ever writing P down."

R is the reward function, a scalar signal $R(s, a, s')$ the environment hands the agent on each transition. By convention R is real-valued, often dense in a sim benchmark (one unit of reward per small move toward the goal) and almost always sparse on a real robot (one unit at task completion, zero otherwise). §5.4 spends its entire budget on the consequences of that single design choice.

$\gamma \in [0, 1)$ is the discount factor, the multiplicative weight applied to future rewards: a reward r_t received at time t contributes $\gamma^t r_t$ to the agent's total. $\gamma = 0$ makes the agent myopic, caring only about the immediate reward, while γ near 1 makes it far-sighted; common values run 0.95 to 0.99 for episodic tasks, and the precise value affects both the solution and the convergence rate of the algorithms in §5.2. The mathematical reason $\gamma < 1$ gets enforced, rather than left at 1, is that an infinite sum of bounded rewards otherwise diverges. The practical reason is that a finite γ acts as a kind of horizon truncation keeping the value function bounded.

Five letters. The full Sutton & Barto (2018) treatment elaborates each one for a chapter; this section's goal is working fluency, not completeness.

The Markov property

The "Markov" in "Markov decision process" is doing real work, and it's the single assumption the rest of the formalism rests on. A process is Markov if the next state depends only on the current state and action, not on the history that produced the current state:

$$P(s_{t+1} \mid s_t, a_t, s_{t-1}, a_{t-1}, ...) = P(s_{t+1} \mid s_t, a_t).$$

For the gridworld, the property holds trivially: the cell you're in plus the action you take determines (stochastically) the cell you land in, and how you got to the current cell is irrelevant. For the Franka arm with a full joint-position-and-velocity state vector, the property holds approximately, since given q and $\dot{q}$, the dynamics equation $M(q)\ddot{q} + C(q, \dot{q})\dot{q} + g(q) = \tau$ from §4.3 says the next state is determined by the current one plus the torque command, with no explicit dependence on the past.

For OpenVLA staring at a single 224×224 image with no temporal context, the Markov assumption is false. A still image of a block on a table doesn't tell you whether the gripper is moving toward the block or away from it; a stack of two or three frames is a strictly larger Markov state than one frame. RT-1 (arXiv:2212.06817) and most modern VLAs include a short history of frames precisely because a single frame is sub-Markov for most manipulation tasks. The general treatment of "the state the agent observes is not the true Markov state" is the partially observable MDP (POMDP) of Kaelbling, Littman, and Cassandra (1998). We won't develop the POMDP machinery in this book, but the name is worth knowing, because every time a VLA paper talks about "history conditioning," it's effectively reaching for a finite-history approximation of the underlying POMDP.

Policies, returns, and value

A policy π is a mapping from states to actions. A deterministic policy $\pi : \mathcal{S} \to \mathcal{A}$ commits to a single action in each state. A stochastic policy $\pi(a \mid s)$ gives a probability distribution over actions in each state, and this is the form almost every learned policy in this book takes. RT-1 and OpenVLA both output a distribution over discretized action tokens; Diffusion Policy samples from a learned conditional distribution; π0 (arXiv:2410.24164) samples from a flow-matching distribution. The notation $a \sim \pi(\cdot \mid s)$, which you'll see in almost every algorithm box from here on, means "sample an action from the policy's distribution conditioned on the current state."

A trajectory under policy π starting from state s_0 is the sequence $s_0, a_0, r_0, s_1, a_1, r_1, s_2, ...$ generated by sampling $a_t \sim \pi(\cdot \mid s_t)$ and $s_{t+1} \sim P(\cdot \mid s_t, a_t)$ at each step, with $r_t = R(s_t, a_t, s_{t+1})$. The return from time t is the discounted sum

$$G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}.$$

The value of a state under policy π is the expected return starting from that state:

$$V^\pi(s) = \mathbb{E}_\pi[\, G_t \mid s_t = s \,].$$

And the Q-value (or action-value) is the expected return starting from state s, taking action a, and following π thereafter:

$$Q^\pi(s, a) = \mathbb{E}_\pi[\, G_t \mid s_t = s, a_t = a \,].$$

Those three objects, return, value, action-value, are the only ones you need to keep in your head for the next two sections. Value iteration and policy iteration in §5.2 are algorithms for computing V^π and Q^π when P and R are known; Q-learning in §5.3 is the algorithm for estimating Q^π when they aren't. Almost everything in modern deep RL is, structurally, a way to fit a parametric approximation of V or Q (the critic) and use it to update a parametric approximation of π (the actor).

A worked gridworld

To make the tuple concrete, here's a small MDP written out explicitly. The state space $\mathcal{S}$ is the sixteen cells of a 4×4 grid, indexed (i, j) for $i, j \in \{0, 1, 2, 3\}$. The goal cell is $(3, 3)$ and is absorbing. The action space is $\mathcal{A} = \{\text{N}, \text{S}, \text{E}, \text{W}\}$. The transition P is deterministic for now: action N from (i, j) produces $(i-1, j)$ if that cell exists, and (i, j) otherwise (a no-op against the wall). The reward $R(s, a, s') = -1$ for every step until s' is the goal, at which point the episode ends. The discount is $\gamma = 0.95$.

Three things are worth noticing about this five-line specification. It fits on a napkin, and so does every gridworld variant you'll see for the next two chapters; don't let the formal notation suggest the underlying object is complicated. The reward is sparse-ish: negative-one-per-step is the convention that turns "reach the goal" into "reach the goal as fast as possible," a convention that exists because a purely sparse reward (zero everywhere, one at the goal) is harder to learn from in tabular settings. And the optimal policy is obvious to a human, move toward the goal, but the algorithm that discovers it without being told is the substance of the chapter, and the rest of §5.2 through §5.4 covers how that algorithm works and why it sometimes doesn't.

What the MDP buys you, and what it costs

The MDP gives you three things the symbolic models of §4.1 don't. A training signal: the reward R is precisely what the symbolic-action engineer was missing when she had to write down all effects by hand. Uncertainty in the dynamics: P is a distribution, and the agent can be optimal in expectation even when the world is non-deterministic. And a compositional notion of optimality: the value function is recursive in the structure of the state space, which is what makes the Bellman equation in §5.2 the workhorse of the field's next thirty years.

What it costs is the assumption that the world fits the tuple. The state space has to be specifiable, the transition has to be Markov in that state space, the reward has to be expressible as a scalar, and the discount has to capture the actual horizon of the task. None of these are automatic on a real robot, and §5.5 is dedicated to the ways they fail. For now, the working agreement is that the MDP is the right vocabulary for talking about reward-driven action models, even when the specific MDP you write down for your robot is wrong in the third decimal place.

Section 5.2 picks up the value function and shows how it can be computed, via value iteration and policy iteration, whenever P and R are known.

5.2 Value iteration and policy iteration

§5.1 introduced the value function V^π and the action-value Q^π but didn't say how to compute them. This section closes that gap for the case the rest of the chapter doesn't assume: the case where the transition P and the reward R are known. That assumption sounds restrictive, and from a robotics standpoint it is; you almost never have a clean tabular P for a real arm. But the two algorithms that fall out of it, value iteration and policy iteration, are

the algorithmic foundation every later method in this book inherits its structure from, from Q-learning in §5.3 to the PPO update of §7.3 to the critic inside SAC. Read this section as the place where the shape of all subsequent RL algorithms gets set, with the inconvenient assumption stripped away later.

Both algorithms rest on a single equation. We spend the first half of the section understanding that equation, then turn it into code.

The Bellman equations

Fix a policy π and a state s. The value $V^\pi(s)$ is the expected return from s under π. The defining recursive identity is the Bellman expectation equation:

$$V^\pi(s) \;=\; \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a)\,\big[\, R(s, a, s') + \gamma V^\pi(s') \,\big].$$

In words: the value of s under π is the average, over the actions π would pick and the transitions those actions would induce, of the immediate reward plus the discounted value of where you land. The equation is a statement about consistency, not optimality. It's true for any policy. The corresponding identity for Q^π is

$$Q^\pi(s, a) \;=\; \sum_{s'} P(s' \mid s, a)\,\big[\, R(s, a, s') + \gamma \sum_{a'} \pi(a' \mid s') Q^\pi(s', a') \,\big],$$

which differs only in committing to action a at the current step before falling back on π.

The Bellman optimality equation is the same identity, written for the optimal value $V^\star(s) = \max_\pi V^\pi(s)$:

$$V^\star(s) \;=\; \max_a \sum_{s'} P(s' \mid s, a)\,\big[\, R(s, a, s') + \gamma V^\star(s') \,\big].$$

The $\sum_a \pi(a \mid s)$ has become a $\max_a$. The optimal value of a state is the best you can do in one step plus the optimal value of where that step lands you. Bellman (1957) proved that this equation has a unique fixed point in V for $\gamma < 1$ and bounded rewards. Both algorithms in this section are ways to find that fixed point.

Value iteration

Define the Bellman optimality operator $T^\star$ acting on a value function V:

$$(T^\star V)(s) \;=\; \max_a \sum_{s'} P(s' \mid s, a)\,\big[\, R(s, a, s') + \gamma V(s') \,\big].$$

A value function is optimal exactly when $V = T^\star V$. The algorithm called value iteration picks an initial V_0, usually the all-zeros vector indexed by $\mathcal{S}$, and applies $T^\star$ repeatedly:

$$V_{k+1}(s) \;=\; (T^\star V_k)(s).$$

The key analytical fact is that $T^\star$ is a γ-contraction in the ℓ_∞ norm: for any two value functions V and V', $\|T^\star V - T^\star V'\|_\infty \le \gamma \|V - V'\|_\infty$. Banach's fixed-point theorem then guarantees $V_k \to V^\star$ at a geometric rate, and after k steps the worst-case error is at most $\gamma^k \|V_0 - V^\star\|_\infty$. With $\gamma = 0.95$, an error tolerance of 10^{-3} on rewards bounded by 1 takes roughly 150 sweeps, small enough to run on a laptop for tabular problems and small enough to fit in a footnote in every RL textbook.

Once $V^\star$ is in hand, the optimal policy is the greedy policy with respect to it:

$$\pi^\star(s) \;=\; \arg\max_a \sum_{s'} P(s' \mid s, a)\,\big[\, R(s, a, s') + \gamma V^\star(s') \,\big].$$

Here's value iteration as a 12-line Python loop on the §5.1 gridworld:

```
import numpy as np

n_states, n_actions = 16, 4   # 4x4 grid; N, S, E, W
P = build_transition_tensor()  # shape (n_states, n_actions, n_states)
R = build_reward_tensor()      # shape (n_states, n_actions, n_states)
gamma, tol = 0.95, 1e-6

V = np.zeros(n_states)
while True:
    Q = (P * (R + gamma * V[None, None, :])).sum(axis=2)  # (S, A)
    V_new = Q.max(axis=1)
    if np.max(np.abs(V_new - V)) < tol:
        break
    V = V_new
pi_star = Q.argmax(axis=1)
```

Two lines do all the work. `Q = (P * (R + gamma * V[None, None, :])).sum(axis=2)` is the Bellman operator written as a tensor contraction; `V_new = Q.max(axis=1)` is the $\max_a$. On the deterministic gridworld of §5.1, this loop converges in well under a hundred sweeps and recovers the optimal policy of "move toward the goal," with $V^\star$ taking values close to $-1/(1-\gamma) \approx -20$ in the corner cells and 0 at the absorbing goal.

Policy iteration

Howard (1960) proposed an alternative scheme that splits the fixed-point problem into two simpler ones. Start with an arbitrary policy π_0, then repeat two steps.

Policy evaluation computes V^{π_k}, the value of the current policy, by solving the Bellman expectation equation. Because π_k is fixed, this is a linear system in $|\mathcal{S}|$ unknowns:

$$V^{\pi_k}(s) \;=\; \sum_{s'} P_{\pi_k}(s' \mid s)\,\big[\,R_{\pi_k}(s,s') + \gamma V^{\pi_k}(s')\,\big],$$

where P_{π_k} and R_{π_k} marginalize over the policy. You can either solve this linear system directly with a matrix inverse, fine for a 16-state gridworld, painful for a 1000-state model, or run a small inner loop of Bellman expectation updates to convergence.

Policy improvement replaces π_k with the greedy policy with respect to V^{π_k}:

$$\pi_{k+1}(s) \;=\; \arg\max_a \sum_{s'} P(s' \mid s,a)\,\big[\,R(s,a,s') + \gamma V^{\pi_k}(s')\,\big].$$

The policy improvement theorem (Howard 1960; Sutton & Barto 2018 §4.2) guarantees π_{k+1} is at least as good as π_k everywhere, with strict improvement in at least one state unless π_k is already optimal. Since there are only finitely many deterministic policies, the iteration terminates in finitely many steps, usually a handful, even for moderately sized MDPs.

A compact Python sketch:

```
pi = np.zeros(n_states, dtype=int)    # start with "always N"
while True:
    # 1. Evaluation: solve linear system for V^pi
    P_pi = P[np.arange(n_states), pi, :]            # (S, S)
    R_pi = (P[np.arange(n_states), pi, :]
            * R[np.arange(n_states), pi, :]).sum(axis=1)  # (S,)
    V = np.linalg.solve(np.eye(n_states) - gamma * P_pi, R_pi)
```

```
    # 2. Improvement: greedy w.r.t. V
    Q = (P * (R + gamma * V[None, None, :])).sum(axis=2)
    pi_new = Q.argmax(axis=1)
    if np.array_equal(pi_new, pi):
        break
    pi = pi_new
```

On the gridworld of §5.1, policy iteration converges in three or four outer iterations: an initial "always go north" policy improves to something sensible in the first sweep, refines to the optimal policy in the second, and the third sweep confirms there's nothing left to improve. Each outer step solves a small linear system and does much more work per step than a value-iteration sweep, but there are far fewer of them.

Two algorithms, one family

Value iteration and policy iteration look different but interpolate into each other. Modified policy iteration replaces the exact linear solve in policy evaluation with m Bellman-expectation sweeps; at $m = 1$ this is approximately value iteration, at $m = \infty$ it's exact policy iteration. Asynchronous variants (Bertsekas 2017, §1.3) update individual states rather than the whole vector at once, and converge under the same contraction guarantee as long as every state is visited infinitely often. Sutton & Barto (2018) §4.5 introduces these under the name generalized policy iteration and makes the point this section will end on: every learning algorithm in the rest of the book is some form of GPI. There's an evaluation step that pushes a value estimate toward consistency with the current policy, and an improvement step that pushes the policy toward greediness with respect to the current value. Q-learning, SAC, PPO, and the critic-and-actor objectives of every VLA paper involving RL are all this same pattern with different approximations and different truncations.

The choice between value and policy iteration in practice is a matter of constant factors. Value iteration is simpler to implement, has no inner loop, and is easy to vectorize. Policy iteration converges in many fewer outer steps but each step is expensive, especially with an exact linear solve. For small tabular problems the difference is invisible; for the function-approximation versions of Chapter 7, the two algorithms morph into the value-target and policy-target sides of an actor-critic loop, and choosing which to lean on is what gives TD3, SAC, and PPO their distinct flavors.

What dynamic programming gives, and what it cannot

Both algorithms share three properties worth pulling out. They're off-line: they need the full P and R in advance, not sampled experience. They're exact under their assumptions: at convergence, the policy is provably optimal, not approximately so. And they're exponential in state-space dimension: a 4×4 gridworld has 16 states, a $20 \times 20 \times 20$ block-stacking task on a discretized table has $8{,}000$, and a robot arm with even a moderate discretization of joint angles has more states than there are atoms in something dramatic. Bellman (1957) coined the phrase "curse of dimensionality" for exactly this: every classical dynamic-programming algorithm scales polynomially in $|\mathcal{S}|$, and $|\mathcal{S}|$ scales exponentially in the number of state variables.

Two consequences shape the rest of the book. Sampling replaces enumeration. Q-learning in §5.3 and the deep actor-critic methods of Chapter 7 estimate $Q^\star$ from sampled transitions, never writing P down. And function approximation replaces tabular storage: a neural network with a few hundred thousand parameters represents a V or Q defined over millions of states without ever materializing the table. The elegance is lost; the scaling isn't. Every deep RL method we discuss from Chapter 7 onward runs approximate value or policy iteration with neural networks standing in for the table, and the analytical guarantees falling out of the contraction argument here get weakened or lost in the process. That's the price of leaving the tabular world.

Section 5.3 takes the first step in that direction: it drops the assumption that P is known, keeps the tabular setting, and shows that a single sampled transition is enough to do approximate value iteration in the form known as Q-learning.

5.3 Q-learning and the role of exploration

Value iteration in §5.2 needed the full transition tensor P and reward tensor R. A robot doesn't have those. What it has is a stream of experience: at each time step it sees a state, picks an action, lands somewhere, and collects a reward. Q-learning, due to Watkins (1989), is the algorithm that drops the model assumption and keeps almost everything else. It's the single most important tabular RL algorithm in the book, not because anyone runs tabular Q-learning on a real robot, but because every deep RL method we touch in Chapter 7, and the critic inside every actor-critic VLA paper, is some approximation of it.

This section does two things. The first half derives the Q-learning update from the Bellman optimality equation and states the conditions under which it converges. The second half covers the part of the algorithm those conditions hide: how the agent decides which actions to try in the first place. That problem, exploration, is the part of RL that becomes the rate-limiting step once you leave the gridworld.

From Bellman to a sampled update

Recall the Bellman optimality equation for $Q^\star$:

$$Q^\star(s,a) \;=\; \mathbb{E}_{s'\sim P(\cdot|s,a)}\big[\,R(s,a,s') + \gamma \max_{a'} Q^\star(s',a')\,\big].$$

Value iteration computes the right-hand side by enumeration, summing over all s'. Q-learning computes it by sampling: take one transition (s,a,r,s') from the world, treat $r + \gamma \max_{a'} Q(s',a')$ as a one-sample estimate of the expectation, and nudge $Q(s,a)$ toward it. The update is

$$Q(s,a) \;\leftarrow\; Q(s,a) + \alpha\,\big[\,r + \gamma \max_{a'} Q(s',a') - Q(s,a)\,\big].$$

The bracketed quantity is the temporal-difference error, or TD error: the gap between the current estimate $Q(s,a)$ and what one new piece of evidence says it should be. The step size $\alpha \in (0,1]$ controls how much the estimate moves on each update. With $\alpha = 1$ the new value just overwrites the old one, high-variance and unstable. With α small, the estimate becomes a slow running average of TD targets, which is what we want.

A few features of this update repay attention. It doesn't depend on P, since the transition is drawn from the world, not from a model. It doesn't depend on the policy the agent is following, either; the $\max_{a'}$ inside the target runs over the agent's current Q, not over the action it actually took next. That last property is what makes Q-learning off-policy: the learned Q converges to the optimum even if the data-collecting policy is bad, as long as the data covers every state-action pair sufficiently. The behaviour policy and the target policy are decoupled, and that decoupling is what lets DQN later (Mnih et al. 2015) train from a replay buffer of stale transitions.

Tabular Q-learning, end to end

Here's the algorithm. Initialize Q as a table, usually zeros, or small random values to break ties:

```
Q = np.zeros((n_states, n_actions))
alpha, gamma, eps = 0.1, 0.95, 0.1

for episode in range(n_episodes):
    s = env.reset()
    done = False
    while not done:
        # behaviour policy: epsilon-greedy
        a = (np.random.randint(n_actions)
             if np.random.rand() < eps
             else int(Q[s].argmax()))
        s_next, r, done = env.step(a)
```

```
        # off-policy TD update
        td_target = r + (0 if done else gamma * Q[s_next].max())
        Q[s, a] += alpha * (td_target - Q[s, a])
        s = s_next
```

Twenty lines, two of which do the actual learning. Run this on the §5.1 4×4 gridworld for a few thousand episodes and Q converges to within rounding of the $Q^\star$ that value iteration produced from the known P in §5.2, but with one consequential difference: you never wrote P down. The convergence guarantee, due to Watkins & Dayan (1992), requires three things. Every (s, a) must be visited infinitely often. The step sizes must satisfy the standard stochastic-approximation conditions $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$, large enough to keep moving, small enough to settle. And the rewards must be bounded. Under those conditions $Q_t \to Q^\star$ with probability 1. The proof is a stochastic-contraction argument generalizing the deterministic contraction of §5.2; the intuition stays unchanged.

The first of those three conditions looks innocent on the page and is the source of every hard problem in the rest of this chapter.

Why "visit every (s, a) infinitely often" is the hard part

The convergence theorem assumes a stream of transitions covering the whole state-action space. The agent's job is generating that stream. If the policy it follows during learning never tries a particular action in a particular state, the Q-value for that pair never updates, and the algorithm can't find out whether that pair was secretly the best one.

This is the exploration problem, and it has two faces. The shallow one is the multi-armed bandit: with k arms and unknown rewards, how do you balance pulling the arm that looks best so far (exploitation) against pulling under-tried arms whose true mean might be higher (exploration)? Lai & Robbins (1985) proved that the regret of any reasonable algorithm grows at least logarithmically in the number of pulls; Auer, Cesa-Bianchi & Fischer (2002) showed that the upper confidence bound (UCB) algorithm matches that lower bound up to constants. UCB picks the arm a that maximizes $\hat{\mu}_a + \sqrt{2 \log t / n_a}$, where $\hat{\mu}_a$ is the empirical mean and n_a is the visit count: optimistic in the face of uncertainty, with the optimism shrinking as you sample more.

The deep face of the problem is that an MDP isn't a bandit. To discover that an action is good you may have to take it, then take five more downstream actions correctly, then notice a reward at the end. The credit-assignment delay turns the visit-everywhere requirement from "try every action a few times" into "produce trajectories that reach every relevant region of state space." On a gridworld this is easy. On a robot arm trying to open a drawer, an agent that has never closed its hand on the handle never sees the positive reward and can't learn to want to do so. Reward sparsity amplifies the exploration problem until it dominates everything else, and the next section, §5.4, is largely about that.

Four ways to explore

The simplest exploration strategy is ε-greedy: take the greedy action with probability $1 - \varepsilon$, a uniformly random action with probability ε. It satisfies the visit-everywhere condition asymptotically, it's one line of code, and it's what the listing above uses. Its weakness is that the random fraction gets spent equally on actions you already know are bad; the agent that has discovered the goal is in the top-right corner of the gridworld still spends 10% of its actions moving west.

Boltzmann (softmax) exploration fixes that by making the randomness graded:

$$\pi(a \mid s) \;=\; \frac{\exp(Q(s,a)/\tau)}{\sum_{a'} \exp(Q(s,a')/\tau)}.$$

A high temperature τ makes the policy nearly uniform; a low one makes it nearly greedy. The agent now wastes less effort on actions it already knows are losses, but it can still get stuck if a single suboptimal action holds a slight Q-value lead.

Optimism in the face of uncertainty generalizes UCB to MDPs. Add a bonus to the Q-target that grows with how rarely (s, a) has been visited. A common form is $r + \gamma \max_{a'} Q(s', a') + \beta / \sqrt{n(s,a)}$. The bonus pulls the agent

toward state-action pairs it hasn't seen, and shrinks to zero in the limit, so the asymptotic Q-values are still $Q^\star$. The challenge, and the reason this stops being trivial in deep RL, is counting visits in a state space where you never see the same state twice.

Intrinsic motivation drops counts and uses a learned signal instead. The agent maintains a prediction model of the environment and rewards itself for transitions its model finds surprising. Pathak et al. (2017) train a forward-dynamics model on visited states and use its prediction error as an intrinsic reward, the intrinsic curiosity module, ICM. Burda et al. (2018) instead distill a fixed random network and use the distillation error as a surprise signal, random network distillation, RND, in arXiv:1810.12894, which avoids ICM's pathology of being surprised by stochastic TVs. Both methods let an agent make progress in environments where the extrinsic reward is sparse enough that ε-greedy never sees it, and both come back in Chapter 7 when we discuss exploration in the deep RL setting and again in Chapter 18 when we ask how to explore on real robots without breaking them.

What changes with function approximation

Two pieces of the story above are tabular. The convergence proof assumes Q is stored as a table; the visit-count exploration bonuses assume you can count visits to individual states. Replace the table with a neural network, DQN (Mnih et al. 2015) was the moment this became routine, on Atari from raw pixels, and both guarantees soften. The TD update is now a gradient step on a regression loss

$$\mathcal{L}(\theta) \;=\; \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\big[\,(r + \gamma \max_{a'} Q_{\bar\theta}(s', a') - Q_\theta(s,a))^2\,\big],$$

with a separate target network $Q_{\bar\theta}$ to keep the moving target from making training oscillate, and a replay buffer $\mathcal{D}$ to break temporal correlations. The Watkins-Dayan contraction argument no longer applies: function approximation can in principle cause Q-learning to diverge, and a fair fraction of the deep RL literature is engineering around the gap between the tabular guarantee and the practical reality. Chapter 7 takes this up in detail.

The exploration side gets harder too. Counting visits in pixel space is meaningless; two frames an agent has clearly seen before differ in JPEG noise. Pseudo-count methods, density models, and the distillation tricks of RND all exist because the elementary state-action counter tabular optimism relies on has no straightforward analogue in a continuous, high-dimensional state space. The robotics literature, when it talks about exploration at all, has mostly given up on intrinsic-bonus methods and replaced exploration with demonstrations: behavior cloning (Chapter 6) and fine-tuning a pretrained VLA (Chapter 16) sidestep the problem by copying somebody else's reachable trajectories. That's one of the reasons modern foundation action models look less like Q-learning than the historical lineage of §1.3 might suggest.

What Q-learning leaves on the table

Two limitations are worth naming before we move on. Q-learning is value-only: it never represents the policy explicitly. To act, you compute $\arg\max_a Q(s,a)$, which in a continuous action space turns into an inner optimization problem at every step. The policy-gradient methods of Chapter 7, and the actor-critic methods built on top of them, give up the off-policy guarantee in exchange for a parametric policy you can evaluate in constant time. And Q-learning assumes the reward signal exists. That assumption is doing a lot of work, and §5.4 takes apart what happens when it doesn't, or when it does, but at a cost in design effort exceeding whatever it would have cost to just teach the robot by demonstration.

For now, the takeaway is the algorithm. Sampled TD updates replace the enumerated Bellman sweep of §5.2, and the off-policy property allows training from any reasonable data stream. The asymptotic guarantees rest on the agent visiting every relevant state-action pair, and engineering that coverage is the practical bulk of using Q-learning on anything more complicated than a gridworld.

The next section, §5.4, makes that practical bulk concrete: it's about why reward design, choosing what the agent gets paid for, and shaping the signal so it actually drives learning, is the part of RL that consumes the most engineering time and produces the most surprising failures.

5.4 Why reward design is the hardest part

The Bellman equations in §5.2 and the Q-learning algorithm in §5.3 both take the reward function $R(s, a, s')$ as a given. The math from that point on is clean: if the reward says what we want, and the agent maximizes it, we win. But where does the reward function come from? Somebody writes it. And writing a reward function that correctly captures what you actually want a robot to do turns out much harder than the convergence proofs make it look.

This section is about that difficulty. Three distinct problems compound on each other: sparsity (the agent rarely encounters rewards at all, so learning is slow or impossible), shaping (adding intermediate rewards to help learning, which can accidentally teach the wrong thing), and misspecification (the reward you wrote isn't, in fact, the reward you meant). Each has a partial fix; none has a complete one. Understanding why is what makes the rest of Part 2 intelligible. Behavior cloning in Chapter 6 and the fine-tuning recipes in Chapter 16 both exist partly because the reward-design problem is, in practice, extremely hard to solve.

The sparse-reward problem in concrete terms

Return to the drawer-opening robot from §5.3. The natural reward is $+1$ when the drawer is open, 0 otherwise. That reward is sparse; the agent receives signal only when it succeeds at the terminal goal. For a robot arm with, say, 7 degrees of freedom and a continuous action space, the probability that random action sequences open the drawer sits close to zero. The agent spends thousands of episodes in a world where $r = 0$ on every step, the Q-values for every action stay near zero, and the policy never improves. The agent isn't stuck in a local minimum; it hasn't found any minimum at all, because the loss landscape is perfectly flat in the region it has explored.

The shallow fix is exploration-with-demonstration: give the agent a handful of expert trajectories that successfully open the drawer and seed the replay buffer with them. The agent now sees positive reward on those trajectories and begins learning the surrounding region. This is essentially behavior cloning under a different name, and it's one of many places where the RL-versus-imitation-learning distinction blurs in practice.

The slightly deeper fix is reward shaping: replace the sparse $+1$ at success with a dense function that gives credit for intermediate progress.

Reward shaping and why it is dangerous

A shaped reward for drawer opening might be

$$r'(s, a, s') \;=\; r(s, a, s') + d_{\text{shaped}}(s')$$

where d_{shaped} is something like the negative distance from the end-effector to the drawer handle. Now every step that moves the gripper closer to the handle gives a small positive reward, and the agent can learn before it ever opens the drawer.

The problem is that this only helps if your intuition about "closer is better" is correct. In a robot task with an obstacle between the gripper and the handle, a negative-distance bonus teaches the agent to slam directly into the obstacle. More subtly, in a task where you need to grasp before you pull, a distance-to-handle bonus teaches the robot to hover its gripper over the handle rather than close its fingers, because closing the fingers and then pulling moves the fingertip farther from the handle in some joint-space metric.

These are failures of specificity: the shaped reward captured part of what progress looks like but not all of it. The agent isn't dumb; it's doing exactly what you asked. It's maximizing your reward function. The mismatch between "the reward you wrote" and "the thing you actually wanted" is called reward misspecification, and Krakovna et al. (2020) catalogued a striking number of ways it appears in practice.

Specification gaming: some famous examples

Specification gaming refers to a class of misspecification failures where the agent finds a solution satisfying the literal reward function but violating the designer's intent. A few illustrative cases from the literature:

The boat-racing game. In CoastRunners, an Atari-style boat racing game, the reward is a score computed from in-game collectibles and finishing position. An OpenAI agent found it could loop in a small circle collecting a cluster of respawning point-tokens and achieve a higher total reward than any human, all while never finishing the race and catching fire in the process. The reward function didn't say "finish the race"; it said "maximize the score." The agent maximized the score.

The simulated hand. In an early OpenAI experiment, a simulated robot hand was trained to grasp and move an object using a reward based on the measured position of the object. The agent learned to slide the object by pushing it with the back of its palm rather than grasping it, sufficient to change the object's position and collect the reward, without learning anything resembling a grasp.

The Hopper standstill. In several continuous-control benchmarks, an agent rewarded for moving forward and penalized for falling over discovers that the safest policy is leaning just slightly, enough not to fall over but never far enough to risk it, collecting survival-time reward indefinitely without meaningfully locomoting.

None of these agents are buggy. From the standpoint of the reward function given, they're optimal. The flaw is in the specification.

Potential-based shaping: one principled fix

The danger of reward shaping is that it can change which policy is optimal. Ng, Harada & Russell (1999) proved a precise characterization of the safe cases. A shaping term $F(s, a, s')$ preserves the optimal policy if and only if it has the form

$$F(s, a, s') \;=\; \gamma \Phi(s') - \Phi(s)$$

for some potential function $\Phi : \mathcal{S} \to \mathbb{R}$. This is called potential-based shaping. The intuition is that the bonus is the discounted future potential you gain minus the potential you give up, a zero-sum accounting of progress that cancels out along any closed loop, so it can't create artificial cycles for the agent to exploit.

In practice, Φ is often the negative distance to a goal state, the output of a value function from a simpler related problem, or a hand-designed progress measure. The key constraint is that the difference must be computable from the current and next state alone, and must not depend on the action. When that holds, the Bellman optimality of the original reward is preserved, and the shaped reward does nothing more harmful than make the landscape less flat.

This is a genuine result and worth knowing. Its practical limitation is that choosing a good Φ requires roughly the same domain knowledge as choosing the reward function in the first place; you're trading one design problem for a related one. On a robot task with a hundred-dimensional state space and a goal depending on object pose, contact forces, and task stage, writing a potential that doesn't accidentally introduce cycles or plateau regions is non-trivial.

When the reward is there but measuring it is expensive

A different failure mode: you know, in principle, what success looks like, but you can't cheaply measure it during training. Consider a robot that should sort recyclable materials from a bin. A true reward would require knowing whether the sorted item was correctly classified, information that typically needs a human to verify or a separate computer vision pipeline to estimate. If that pipeline takes 3 seconds per query and you need 100,000 queries to train, you have a 3,500-hour bill. Dense rewards measured from sensors are cheap; semantically correct rewards are expensive.

One class of solutions collects a human preference signal instead of a numeric reward. Christiano et al. (2017) showed a reward model can be trained from pairwise human comparisons, "which of these two trajectory clips looks better?", and that this reward model can then supervise a policy without ever requiring the human to specify a closed-form function. The idea is influential in language models (RLHF), but it also appears in robotics when the true reward is too expensive to instrument directly.

Another approach is inverse reward design (Hadfield-Menell et al., 2017): rather than treating the written reward as ground truth, treat it as a noisy observation of the designer's true intent, and infer the likely true reward under a

prior over plausible objectives. The policy optimizes an expectation over possible true rewards rather than the stated one, making it more cautious in regions where the two might diverge.

Both approaches share an assumption that the human designer's preferences are coherent and consistent. That assumption is usually not too badly wrong, but it's an assumption.

Practical implications for robotics

Put together, the three problems, sparsity, shaping hazard, and measurement cost, explain why pure RL from scratch is rarely the first tool reached for in modern robot learning. A few observations worth making explicit before Chapter 6.

Dense rewards are almost always shaped in practice, which means they almost always carry misspecification risk. The standard mitigation isn't better shaping theory; it's evaluation diversity, checking the learned policy on the target task with a different evaluator than the one used for training. If the robot looks like it's succeeding on the training signal but doing something strange on the real task, that gap is a specification error and the shaped reward needs revising.

Sparse rewards are philosophically cleaner, harder to game, but require either excellent exploration or seeding with demonstrations. In robotics, where reset-free training is an active research problem and each episode takes real time on real hardware, the exploration requirement of sparse rewards is often prohibitive without a simulator.

The most commonly used practical solution, which we examine in Chapter 6, bypasses the reward entirely: copy what an expert does, and hope the expert was doing the right thing. This isn't free of failure modes, but the failure modes are different, and in many robotics applications, substantially easier to identify.

The broader point is that the reward function is a specification of intent encoded as a scalar signal. Writing specifications is hard. Writing scalar specifications that remain resistant to optimization pressure is harder still. Reward design is the hardest part not because the math is deep (the Bellman equations don't care where R came from), but because the engineering problem of saying clearly what you want, in a form a learning algorithm can't misread, is one the field hasn't solved.

The next section, §5.5, takes up a related but distinct problem: even with a reasonable reward function in hand, translating the MDP abstraction into a working robot system requires decisions about state representation, action space, episode structure, and sim-to-real transfer that are each capable of silently undermining everything the reward design got right.

5.5 The MDP-to-robot translation problem

The previous section worked on the reward function. It assumed the rest of the MDP tuple was already correctly constructed, that the state space made sense, the action space was a reasonable thing to optimize over, and episodes had sensible boundaries. On a gridworld with sixteen cells, those assumptions are free. On a physical robot, each one is a design decision with its own failure modes.

This section is about that translation step. You have a robot, a task, and now an understanding of MDPs. How do you turn the robot-and-task into a tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ that a learning algorithm can actually make progress on? Four decisions dominate: what goes into the state, what counts as an action, how episodes are structured, and what to do about the fact that the P you can train on in simulation isn't the P you'll face on hardware. Getting any one of these wrong is sufficient to make a well-implemented algorithm completely fail to learn.

State representation: what is Markov?

The Markov property says the state s_t must contain everything relevant about the past. Formally: $P(s_{t+1} \mid s_t, a_t) = P(s_{t+1} \mid s_0, a_0, \dots, s_t, a_t)$. The future is independent of history given the current state. On a gridworld with full visibility, this holds trivially, since the cell you're in tells you everything. On a robot, it almost never holds perfectly, and your choice of state representation determines how badly it's violated.

Consider a robot arm trained to place a cup on a coaster. A minimal state candidate is the joint position and velocity vector, say $q \in \mathbb{R}^{14}$ for a 7-DOF arm. That's Markov for the arm's own dynamics, but not for the task, since it doesn't include the cup's pose or the coaster's location. Add those and you get something like $\mathbb{R}^{14+7+3}$, but now you need a pose estimator measuring cup and coaster pose at 20 Hz with low enough noise to avoid corrupting the policy's input. Sensor noise and observation delay both break the Markov property in practice.

The end-to-end alternative is feeding the policy a raw image, 224×224×3, or a stack of three frames for temporal context, and letting it learn its own implicit state. This is what OpenVLA (arXiv:2406.09246) and RT-2 (arXiv:2307.15818) do. The image isn't Markov either: a single frame can't distinguish an arm that's accelerating from one at rest, which is why frame-stacking exists. But the failure is bounded and well-understood, and for most manipulation tasks the information lost by discarding velocity costs less than the engineering effort of maintaining a reliable 6-DOF pose estimator for every object in the scene.

When the state representation is systematically incomplete, not just noisy but structurally missing information the policy needs, the MDP becomes a partially observable Markov decision process, or POMDP (Kaelbling, Littman & Cassandra, 1998). A POMDP adds an observation space $\mathcal{O}$ and an observation function $\Omega(o \mid s, a)$: the agent sees o, not s. Formally this is the right model for most robotics problems. Practically, POMDP solvers are harder to scale, and most practitioners handle partial observability with one of three engineering approximations: add a recurrent layer so the policy can summarize its own history, stack recent observations as a pseudo-state, or choose a richer observation that makes the missing information less load-bearing. None of these recovers the full Markov guarantee; all of them work well enough in practice for a wide class of tasks.

A specific failure mode worth naming: if the task requires remembering what happened several seconds ago, which drawer was opened, which object was already picked, and the state doesn't include that memory, a Markovian policy can't solve the task. No amount of Q-learning will help; the algorithm converges, but to a policy that can't access the information it needs. The correct fix is changing the state representation, not the learning algorithm.

Action space design

The action space $\mathcal{A}$ isn't fixed by the robot; it's chosen by the engineer. A 7-DOF arm can be controlled through joint torques, joint velocities, joint positions, Cartesian end-effector positions, Cartesian end-effector poses, or sequences of any of the above. Each choice gives a different $\mathcal{A}$, and the choice affects learning difficulty in ways that interact with the task.

Torque control is the most physically fundamental: the action is a vector of joint torques, and the robot's dynamics are governed by the equations of motion from §4.3. This gives the policy maximum expressive power, since any motion the robot is mechanically capable of is reachable. It also places the highest learning burden on the policy, because the mapping from torques to Cartesian endpoint motion passes through the robot's full inertia tensor and gravity compensation. A policy that outputs random torques will flail destructively; the exploration problem from §5.3 becomes a safety problem.

Position control wraps a low-level PD controller around the joint angles, so the policy sets target positions and the controller handles force. This is by far the most common choice in manipulation research. It sacrifices the ability to modulate contact forces (relevant for assembly, insertion, and any task where compliance matters) in exchange for a much calmer action space where random actions produce slow, bounded motion rather than uncontrolled flailing. OpenVLA outputs a 7-vector of joint-position deltas at 5 Hz for exactly this reason.

Chunked or trajectory actions push the abstraction level higher still. Instead of outputting a single control command per step, the policy outputs a short sequence of commands, a 16-step position trajectory, say, and the low-level controller executes the whole chunk before the next policy call. Diffusion Policy (Chapter 10) and ACT both operate in this regime. The MDP timestep is now one chunk, not one control cycle, which reduces the effective horizon and makes credit assignment easier. The cost is reduced reactivity: the robot commits to 16 steps before it can observe the environment again.

Control frequency is a related design variable. A policy running at 1 Hz sees the world as a sequence of slowly changing states; one running at 200 Hz sees rapid oscillations mostly irrelevant to the task. Mismatched frequencies, training at 10 Hz and deploying on hardware that samples at 200 Hz without synchronization, are a common source of sim-to-real failures that look like policy degradation but are actually a different MDP entirely.

There's no universally correct choice. The practical heuristic: use the lowest-level action space you can train stably, since it preserves the most expressiveness for fine-grained tasks. In practice this usually means position deltas at 10 to 20 Hz for most tabletop manipulation, with a separate compliance layer if the task involves contact-rich assembly.

Episode structure

Every RL training loop has episodes: a start state, a sequence of steps, and a terminal condition. On a gridworld, episodes are tidy, since the agent starts at a fixed cell, reaches the goal or falls into a pit, and resets. On a physical robot, all three components require design decisions.

Resets are expensive and often nondeterministic. A robot that knocks an object off the table requires a human to replace it before the next episode. At 1 minute per episode and 100,000 episodes of training, that's 70 days of human labor per run. Simulators eliminate this cost; real hardware doesn't. Practical strategies include training entirely in simulation and accepting sim-to-real error, training on tasks that are self-resetting (a drawer that can be opened and closed indefinitely, say), and learning from demonstrations rather than RL, which requires far fewer trials.

Horizon length T controls how far ahead the value function must reason. A task that takes 500 steps at 10 Hz lasts 50 seconds; value estimates must propagate over all 500 steps, and the discount factor γ must be high enough (close to 1) that the terminal reward isn't discounted to near zero. High γ slows value-function convergence; low γ causes the agent to discount future outcomes so heavily it behaves myopically. For a task with a natural horizon of 50 seconds, $\gamma = 0.99$ is typical, meaning the reward at the last step is worth about $0.99^{500} \approx 0.007$ at the start, thin but nonzero. For a task with a horizon of 10 minutes, γ must be tuned or the task reformulated with intermediate rewards.

Termination conditions control when an episode ends before the horizon. Early termination on success (the object was placed) is almost always good: it keeps episodes short and keeps the replay buffer from clogging with post-success idle steps. Early termination on failure (the arm collided, the object was dropped) is more nuanced. Terminating on every collision teaches the agent that collisions are absorbing, that nothing recoverable ever follows, which is accurate in some tasks and not in others. For contact-rich tasks, excessive early termination on collision teaches the agent to be so collision-averse it never makes contact with objects it needs to manipulate.

The sim-to-real gap

For any non-trivial task, training on real hardware end-to-end isn't feasible at the sample counts RL typically requires. The standard engineering solution is training in a physics simulator and deploying on the real robot. This introduces the sim-to-real gap: the transition function P_{sim} you trained on differs from the transition function P_{real} you deploy on, and the policy has no way to detect the difference.

The gap has several sources. Contact dynamics, what happens when two surfaces touch, are both physically complex and computationally expensive to simulate faithfully. Most simulators used for RL training (MuJoCo, Bullet, Isaac) use simplified contact models that are fast and stable but inaccurate in subtle ways: they underrepresent surface friction variation, ignore deformable objects, and smooth over fast-contact transients. Sensor noise is typically modeled as additive Gaussian; real sensor noise isn't. Actuator dynamics, backlash, delay, saturation, are often ignored entirely in simulation.

The canonical treatment is domain randomization (Tobin et al., 2017; Peng et al., 2018): during training, randomly vary the parameters of the simulator, friction coefficients, object masses, actuator delays, visual textures, so the real world is, in some probabilistic sense, one sample from the distribution the policy has already been exposed to. If the policy has learned to be robust across a wide range of friction values, it will generalize to the real robot's friction even if no single simulation run matched it exactly.

Domain randomization works when the real world is plausibly in the support of the randomized distribution. It fails when the sim-to-real gap is caused by something qualitatively absent from the simulator, compliant objects, wet surfaces, cables, rather than by parameter mismatch. For those cases, the only reliable option is real-robot data, which brings the sample-complexity discussion full circle. Chapter 9 examines world models as a partial answer: if you can learn a good model of real-robot dynamics from a small amount of hardware data, you can train in that learned model rather than in a fixed simulator.

When all four decisions interact

The practical failure mode usually isn't one bad choice in isolation; it's four mediocre choices interacting. A policy trained with position-control at 10 Hz on a task with a 200-step horizon, using an observation that omits object velocity, in a simulator with no friction randomization, will fail to deploy for at least three simultaneous reasons, and figuring out which one is load-bearing requires ablating each design decision separately. In a physics lab this is called controlling variables. In robot learning it's rarely done systematically, which is one reason published results are hard to reproduce.

The checklist before running RL on a new task runs roughly: confirm the observation is sufficient for the task (if it isn't, fix the observation before writing a reward), choose an action abstraction at the right granularity for the task's required precision, set the episode horizon and reset strategy with explicit knowledge of the training budget, and identify the top two or three sources of sim-to-real gap before training, so you know what to randomize.

None of this is specific to MDP theory. It's engineering. But it's the engineering the theory assumes has been done correctly, and that assumption is almost never true on the first pass.

The next section rounds out Chapter 5 with a summary of what the MDP formalism gives the reader and what it conspicuously leaves open, the latter being, in large part, the motivation for everything in Chapters 6 through 10.

5.6 Summary

Chapter 5 was the reward chapter. Every section from §5.1 to §5.5 circled the same object, the Markov decision process, from a different angle, and the cumulative message runs more nuanced than "reward works." It works under specific conditions, fails in specific ways, and the engineering cost of applying it to a physical robot is non-trivial in ways the theory doesn't advertise. This summary collects the load-bearing ideas and flags which ones the rest of the book keeps reaching for.

The four ideas worth carrying forward

The MDP tuple is a modeling choice, not a fact about the world. §5.1 introduced the five-tuple $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ as a mathematical object, but the more important point was that you, the engineer, have to choose every component of it for each specific task. The state space doesn't arrive pre-labeled; you decide whether to represent a tabletop scene as joint angles, end-effector poses, a raw image, or all three. The action space doesn't arrive with a preferred granularity; you decide whether the agent controls joint torques at 200 Hz or Cartesian poses at 10 Hz. The reward function doesn't arrive with the task; you invent it. This design burden is the tax the MDP formalism charges for its analytical tractability, and §5.5 spent an entire section itemizing the failure modes of each component in the robotics context. When you read a paper that says "we trained an RL agent on this task," the first question is always what they chose for $\mathcal{S}$, $\mathcal{A}$, and R, and whether each choice makes sense for the task. The theory provides no answer; only the engineer does.

Optimal value functions are defined recursively, and all practical RL algorithms are algorithms for computing them. §5.2 derived the Bellman optimality equations and showed that both value iteration and policy iteration can be read as algorithms for finding their fixed point. §5.3 showed that Q-learning is the same fixed-point iteration with P replaced by sampled transitions. The value function in a DQN critic (Chapter 7) is the same object, approximated by a neural network rather than stored in a table. The critic inside SAC is the same object again, extended to a continuous action space. None of this is coincidence: the Bellman equations are the structure of discounted sequential decision-making, and every algorithm that learns from reward inherits that structure. When you see an actor-critic update in Chapter 7, or a value-function baseline in a policy-gradient, recognizing that these are Bellman fixed-point iterations with function approximators and variance reduction bolted on is the insight that makes the full family of deep RL methods legible rather than a catalogue of unrelated tricks.

Q-learning is off-policy, and off-policy is the property that makes large replay buffers possible. §5.3 showed that Q-learning's TD update converges to $Q^\star$ regardless of the behaviour policy, as long as every state-action pair is visited sufficiently. That decoupling between the data-collection policy and the learned policy is the fundamental property letting DQN (Chapter 7) train from a buffer of a million transitions from stale policies, and letting offline RL

(referenced in Chapter 12 in the context of data-driven pretraining) train from a fixed dataset with no environment interaction at all. The convergence guarantee softens under function approximation, since the tabular argument doesn't carry over cleanly to neural networks, but the architectural motivation doesn't budge. Every time you see a replay buffer in a deep RL system, you're seeing the practical consequence of Q-learning's off-policy property.

Reward design is an engineering problem disguised as a specification problem. §5.4 made the case that writing a correct reward function is structurally harder than it looks, for two distinct reasons. Sparse rewards provide no gradient until the agent accidentally succeeds, rare enough on manipulation tasks to be practically impossible without exploration engineering or prior knowledge. Dense rewards solve the gradient problem but introduce specification gaming: the agent optimizes whatever you measure, and the gap between what you measure and what you want is nearly always nonzero. Potential-based shaping (Ng, Harada & Russell, 1999) gives a principled way to add intermediate signal without corrupting the optimal policy, but it still requires a useful potential function, and designing one correctly is nearly as hard as designing the dense reward directly. The practical upshot is that reward engineering is expensive, iterative, and prone to surprising failures even in competent hands, one of the two main reasons behavior cloning, covered in Chapter 6, displaced RL as the dominant training signal for manipulation in the 2020s. The other main reason is sample efficiency.

What you should be able to do now

Four concrete capabilities, in roughly the order they're needed in later chapters.

You should be able to write down a task as an MDP tuple and identify all the modeling choices you made. Pick any task, a robot pouring water, a navigation agent in a maze, an arm unscrewing a bolt, and enumerate $\mathcal{S}$, $\mathcal{A}$, P, R, γ explicitly. Name the elements of the state, the resolution and frequency of the action, the form of the transition (deterministic? stochastic? known? learned?), and the reward signal including any shaping terms. Then list the three most dangerous modeling choices, the ones where a different choice would have made the task substantially easier or harder to learn. This exercise isn't academic. Every design review of an RL system in industry is, at its core, this exercise, and doing it fluently is the skill separating people who can debug RL failures from people who can only name them.

You should be able to implement tabular Q-learning on a small problem, run it to convergence, and verify the result against the value function produced by value iteration. The 4×4 gridworld in §5.2 is the right test bed. The Q-learning implementation from §5.3 fits in under twenty lines; adding ε-greedy exploration, a decay schedule for ε, and a convergence check adds another ten. The diagnostic is comparing $Q_\infty(s, a)$ from Q-learning against $Q^\star(s, a)$ from value iteration over the known P: they should agree to within the noise introduced by the finite episode count and the stochastic-approximation step sizes. If they don't, the Q-learning implementation is wrong, and the most common reason is that the transition tuple (s, a, r, s') is being formed incorrectly, with r being the reward for arriving in s' but incorrectly assigned to departing s. Getting this right on a gridworld before touching deep RL is the exercise that saves the most debugging time later.

You should be able to recognize reward gaming in a published result and propose a potential-based fix. Given a description of a task and a reward function, identify at least two behaviors that would achieve high reward without achieving the task. For each, either propose a potential Φ whose gradient discourages the exploit, or argue that the exploit can't be suppressed without a fundamentally different reward structure. This skill matters because every deployment story in Chapter 17 includes at least one reward-gaming incident, and the difference between catching the gaming before deployment and catching it after comes down almost entirely to whether the engineer thought carefully about the reward function at design time.

You should be able to read a robotics RL paper and identify the four MDP-to-robot design decisions, state representation, action space, episode structure, sim-to-real strategy, and classify each as well-motivated, underspecified, or suspect. §5.5 named the failure modes: a state representation insufficient for the task, an action space at the wrong level of abstraction, episode resets that don't match real-world cost, and a sim-to-real gap that was never measured. Papers that underspecify these choices are papers whose results may not hold on different hardware or under slightly different task conditions, and identifying the gap is the first step toward reproducing or extending the work.

Where the chapter has set up the rest of the book

Chapter 5 hands off in three directions. The most direct is Chapter 7, deep RL for control, which takes the tabular Q-learning of §5.3 and the policy iteration of §5.2 and replaces every table with a neural network. DQN is tabular Q-learning with a replay buffer and a target network. PPO is policy iteration with a surrogate objective and a clipped gradient. SAC is actor-critic with soft Bellman backups. All of Chapter 7 is Chapter 5 with function approximation, and a reader who understands the tabular case will find the deep-RL case a set of engineering choices rather than new theory.

The less direct handoff is Chapter 6, learning from demonstrations. §5.4 ended by arguing that reward design is expensive enough that behavior cloning often wins on practical grounds. Chapter 6 makes that case formally: it introduces the imitation-learning framing, shows why behavior cloning is a supervised problem that sidesteps reward design entirely, and shows where it fails (compounding error). Chapter 6 is, in part, the chapter explaining why the VLAs in Part 4 look the way they do, trained on demonstrations, not on reward.

The third handoff runs longer-range. §5.5 described the sim-to-real gap and noted that world models (Chapter 9) are one partial answer: if you can learn a good model of real-robot dynamics from a small amount of hardware data, you can train in that learned model rather than in a fixed simulator, updating it as the robot encounters new environments. The MDP formalism is also the starting point for offline RL, which trains from a static dataset without any environment interaction; that approach becomes relevant in Chapter 12 when we discuss how OpenVLA and Octo were pretrained on large heterogeneous datasets collected by other researchers.

What the chapter has not covered

Two omissions are worth naming. Chapter 5 covered MDPs but not POMDPs in any depth. §5.5 noted that most robotics problems are technically POMDPs, since the agent doesn't have direct access to the full state, but treated partial observability as an engineering issue handled by observation design and frame-stacking rather than by POMDP algorithms. That's the right practical stance for the systems in this book, but a reader who wants the formal theory should consult Kaelbling, Littman & Cassandra (1998), which established the POMDP framework for robotics, and the approximate-POMDP literature that followed.

The chapter also hasn't covered hierarchical RL: the idea of constructing a two-level MDP where the high-level agent selects subgoals and the low-level agent achieves them. Hierarchical RL is a natural partial answer to both long-horizon credit assignment and reward sparsity, and it appears implicitly in Chapter 14 when dual-system architectures (Helix, GR00T N1) are discussed. The connection is real, but we defer it to the chapter where concrete examples appear rather than introducing the abstraction here without a system to attach it to.

Chapter 5's contribution to the book's overall argument is the reward family from §1.4: the action models that learn by interacting with the world under an extrinsic signal. The three central findings, that the Bellman equations are the shared backbone of all reward-based learning, that reward design is hard enough to be a limiting factor in practice, and that the MDP tuple is a modeling choice with high-stakes design decisions embedded in every component, are the findings Parts 3 and 4 return to when explaining why foundation action models are trained on demonstrations rather than reward, and what would need to change for reward-based pretraining to become practical at scale.

§5.x closes Chapter 5 with a hands-on exercise, implementing Q-learning on the FrozenLake-v1 environment, verifying the result against a known optimal policy, and then watching it fail on a slightly modified reward function, and the full reading list for the chapter.

5.x Hands-on exercise + chapter references

Chapter 5 was the reward chapter; the exercise is where you write the reward and watch what an agent does with it. The four drills below take a combined two hours on a laptop CPU. No GPU required, no robot required, no internet required after the initial dependency install. The point is leaving Chapter 5 with a working Q-learning implementation you trust, a first-hand example of reward gaming you designed yourself, and a practiced eye for the four MDP-to-robot design decisions from §5.5 as they appear in a real paper. Chapter 7 builds on all three.

Install `gymnasium` (the maintained fork of OpenAI Gym, Brockman et al., 2016, arXiv:1606.01540) if you do not have it:

```
pip install gymnasium
```

FrozenLake-v1 with `is_slippery=False` is the test environment for Exercises 5.x.1 and 5.x.2; the slippery variant appears in 5.x.2. No other dependencies are needed for the first three exercises.

Exercise 5.x.1 — Implement tabular Q-learning and verify against value iteration

Write two Python files: `value_iteration.py` and `q_learning.py`. Both operate on the 4×4 FrozenLake-v1 grid with `is_slippery=False`, so the transition tensor P is deterministic.

Value iteration (`value_iteration.py`): gymnasium's `FrozenLakeEnv` exposes the full transition tensor as `env.P[state][action]`, a list of `(probability, next_state, reward, done)` tuples. Use that to run the Bellman optimality update

$$V^{(k+1)}(s) = \max_a \sum_{s'} P(s' \mid s, a) \left[R(s, a, s') + \gamma \, V^{(k)}(s') \right]$$

with $\gamma = 0.99$ until $\max_s |V^{(k+1)}(s) - V^{(k)}(s)| < 10^{-8}$. Extract the greedy policy $\pi^\star(s) = \arg\max_a Q^\star(s, a)$, where you derive $Q^\star$ from $V^\star$ in the usual one-step lookahead. Save $V^\star$ and $\pi^\star$ as reference objects.

Q-learning (`q_learning.py`): implement the update from §5.3,

```
Q[s, a] += alpha * (r + gamma * np.max(Q[s_next]) - Q[s, a])
```

with $\alpha = 0.1$, $\gamma = 0.99$, ε-greedy exploration starting at $\varepsilon = 1.0$ and decaying by a factor of 0.999 per episode to a floor of 0.01. Run for 5,000 episodes. Record the episode-return rolling average (window 100 episodes) and save it as `q_learning_curve.png`. After training, extract the greedy policy $\hat{\pi}(s) = \arg\max_a Q(s, a)$.

The diagnostic is a side-by-side comparison: for each of the 16 states, print $\pi^\star(s)$ from value iteration beside $\hat{\pi}(s)$ from Q-learning. On the deterministic grid they should agree on every non-terminal state. If they don't, the most common causes are the Q-table being indexed (s, a) without handling the `done` flag (bootstrapping on a terminal state gives a corrupt target) and the episode return not resetting between episodes. Fix these before Exercise 5.x.2.

Wall clock: about twenty-five minutes.

Exercise 5.x.2 — Watch Q-learning fail with a shaped reward and then succeed with a principled one

This exercise runs in two parts and demonstrates §5.4's central argument in miniature.

Part A, a shape that backfires. Make a copy of `q_learning.py` called `q_learning_bad_shape.py`. Add a shaping term to the reward:

```python
# Manhattan distance to goal (state 15), lower is better
def manhattan_to_goal(s):
    row, col = s // 4, s % 4
    return abs(row - 3) + abs(col - 3)

r_shaped = r - 0.1 * manhattan_to_goal(s_next)
```

Train for 5,000 episodes on the slippery variant (`is_slippery=True`, so transitions are stochastic: the intended action fires with probability 1/3, and each orthogonal direction fires with probability 1/3). Plot the episode-return curve.

You'll observe that the agent learns to hover near the goal rather than stepping onto it: stepping onto the goal gives $r = +1$ but also terminates the episode, ending the stream of small negative shaping bonuses for being close. The agent has found a policy that scores higher under your shaped reward than the intended optimal policy does. This

is the §5.4 specification-gaming failure. Describe it in one sentence: "the agent did X because Y metric was higher that way than the correct way."

Part B, potential-based shaping that doesn't corrupt the optimal policy. Make another copy, `q_learning_good_shape.py`. Replace the raw shaping term with the potential-based version from §5.4 (Ng, Harada & Russell, 1999):

```
# Potential: negative Manhattan distance, so Phi(s) is higher near goal
def phi(s):
    row, col = s // 4, s % 4
    return -0.1 * (abs(row - 3) + abs(col - 3))

r_shaped = r + gamma * phi(s_next) - phi(s)
```

Train for 5,000 episodes on the slippery variant. Plot the curve on the same axes as Part A. The potential-based agent should converge to the correct policy, the one that steps onto the goal rather than camping adjacent to it, while still converging faster than the un-shaped baseline from 5.x.1.

Save the comparison plot as `shaping_comparison.png`. The three curves, no shaping (from 5.x.1), bad shaping, and potential-based shaping, on the same axes are the visual summary of why the Ng et al. result matters. The no-shaping and good-shaping curves should end near the same asymptotic performance; the bad-shaping curve should end lower.

Wall clock: about forty minutes including the two training runs and the plot.

Exercise 5.x.3 — Exploration ablation: ε-greedy vs. constant ε vs. no exploration

This exercise puts a number on the §5.3 claim that exploration scheduling matters. Use the slippery FrozenLake-v1 from Exercise 5.x.2 with potential-based shaping. Train five agents:

1. $\varepsilon = 1.0$ (pure random — never greedy; the agent collects experience but cannot converge).
2. $\varepsilon = 0.0$ (no exploration — greedy from a zero-initialized Q-table; never leaves the first local path it finds).
3. $\varepsilon = 0.1$ fixed (constant low exploration for the full 5 000 episodes).
4. ε decays from 1.0 to 0.01 over 5 000 episodes (the schedule from 5.x.1).
5. ε decays from 1.0 to 0.01 over 500 episodes, then fixed at 0.01 (fast decay).

Plot all five episode-return rolling averages (window 100) on the same axes. Save as `exploration_ablation.png`. Write one sentence about each curve: what did the agent learn, and why does the curve look the way it does? Then answer one question in plain prose: which of the five would you pick for a real robot, and why?

There's a correct answer for the gridworld, agent 4 or 5 depending on how patient you are, but the reasoning matters more than the number. The §5.5 argument that episode length and reset cost interact with exploration is the argument you're making in miniature here: on a slippery grid, agent 2 (no exploration) gets stuck in the top-left corner and never discovers the goal, exactly as a real robot with a deterministic "always move right" policy never discovers that the grasp has to be approached from above.

Wall clock: about twenty-five minutes for five training runs and the plot.

Exercise 5.x.4 — Read a robotics RL paper through the MDP-to-robot lens

Open one of the following: Haarnoja et al. (2018), "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Learned Reward Function," which includes experiments on a simulated half-cheetah and ant; or any deep-RL-for-manipulation paper from the last three years you have available. Read the experiment section and the appendix. In a text file `paper_audit.txt`, answer the following five questions, one paragraph each:

1. State representation. What is $\mathcal{S}$? Is it Markov for the task? If the paper uses a raw image, what temporal information is discarded and how does the system compensate?
2. Action space. What is $\mathcal{A}$? What control frequency is used? Is this joint torques, joint positions, Cartesian end-effector poses, or something else? Is the choice motivated in the text?

3. Reward function. What is R? Is it sparse, dense, or shaped? If shaped, is the shaping potential-based? Can you identify at least one behavior the reward might accidentally incentivize that differs from the intended task?
4. Episode structure. How are episodes reset? What does the reset cost? Does the reset cost match real-world deployment? If trained in simulation, is the sim-to-real gap discussed?
5. Overall judgment. Of the four design decisions, state, action, reward, episode, which is the most load-bearing and which is the most underspecified? What one change would most improve the validity of the result?

There's no scoring rubric. The point is that the five-question template from §5.5 stops being abstract once you've applied it to a paper that matters to you. Every design-review conversation you'll have in the rest of your career about a robotics RL system is some version of these five questions.

Wall clock: about thirty minutes for the read plus the written audit.

Chapter 5 reading list

The works below are cited across §5.1 through §5.6, grouped by purpose. Full bibliographic entries for everything cited in the book live in Appendix E.2; this list is just the chapter-local subset.

MDP theory: the formal backbone

- Bellman, R. (1957). *Dynamic Programming.* Princeton University Press. The source of the Bellman equations §5.2 derives. Worth reading the first chapter even in 2026.
- Howard, R. A. (1960). *Dynamic Programming and Markov Processes.* MIT Press. The first systematic treatment of policy iteration; §5.2 follows Howard's presentation more closely than Bellman's.
- Puterman, M. L. (1994). *Markov Decision Processes — Discrete Stochastic Dynamic Programming.* Wiley. The authoritative technical reference; Chapter 4 has the contraction proof that §5.2 sketches.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning — An Introduction*, 2nd ed. MIT Press. Free online. The indispensable textbook; Chapters 3–6 cover the material in §5.1–§5.3.
- Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). "Planning and Acting in Partially Observable Stochastic Domains." *Artificial Intelligence* 101(1–2). The POMDP reference §5.5 cites when it explains why most robotics problems are technically POMDPs and why practitioners treat them as MDPs anyway.

Q-learning: derivation and convergence

- Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* PhD thesis, University of Cambridge. The original derivation.
- Watkins, C. J. C. H., & Dayan, P. (1992). "Q-learning." *Machine Learning* 8(3–4):279–292. The published version; four pages long, entirely worth reading.
- Lai, T. L., & Robbins, H. (1985). "Asymptotically Efficient Adaptive Allocation Rules." *Advances in Applied Mathematics* 6(1):4–22. The bandit theory §5.3 draws on when discussing the exploration-exploitation tradeoff; UCB is a direct descendant.

Reward design and shaping

- Ng, A. Y., Harada, D., & Russell, S. (1999). "Policy Invariance under Reward Transformations: Theory and Application to Reward Shaping." *ICML 1999.* The result Exercise 5.x.2 Part B depends on: only potential-based shaping preserves the optimal policy.
- Amodei, D., Olah, C., Steinhardt, J., et al. (2016). "Concrete Problems in AI Safety." The specification-gaming section of §5.4 draws on the framework in this paper.
- Krakovna, V., et al. (2020). "Specification Gaming: The Flip Side of Intelligent Behavior." DeepMind blog. A catalogue of real-world reward-gaming failures; every example in §5.4 is a mild version of something on this list.
- Christiano, P., Ziegler, J., Stiennon, N., et al. (2017). "Deep Reinforcement Learning from Human Preferences." *NeurIPS 2017.* The RLHF paper; the reward-learning framing §5.4 describes as an alternative to hand-specified reward is this paper's contribution.

Deep RL: the bridge to Chapter 7

- Mnih, V., Kavukcuoglu, K., Silver, D., et al. (2015). "Human-Level Control through Deep Reinforcement Learning." *Nature* 518:529–533. The DQN paper; Chapter 7 starts here. §5.3's discussion of the replay buffer is a preview of this paper's two key engineering contributions.
- Schulman, J., Wolski, F., Dhariwal, P., et al. (2017). "Proximal Policy Optimization Algorithms." arXiv:1707.06347. The PPO reference for Chapter 7's policy-gradient section.
- Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Learned Reward Function." *ICML 2018*. The SAC reference for Chapter 7's off-policy actor-critic section. Exercise 5.x.4 uses this paper as one of two options.

The MDP-to-robot translation problem

- Brockman, G., Cheung, V., Pettersson, L., et al. (2016). "OpenAI Gym." arXiv:1606.01540. The standard benchmark environment library, and the conceptual predecessor to the gymnasium package used in this chapter's exercises.
- Todorov, E., Erez, T., & Tassa, Y. (2012). "MuJoCo: A Physics Engine for Model-Based Control." *IROS 2012*. The simulation environment underlying most of the deep-RL manipulation results Chapter 7 cites.
- Tobin, J., Fong, R., Ray, A., et al. (2017). "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World." *IROS 2017*. The sim-to-real strategy §5.5 names as the pragmatic alternative to system identification.
- Peng, X. B., Andrychowicz, M., Zaremba, W., & Abbeel, P. (2018). "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization." *ICRA 2018*. The companion result that shows dynamics randomization, not just visual randomization, is necessary for real-world transfer.

Chapter summary

Chapter 5 gave you the theoretical substrate for reward-based learning, and the four exercises made it concrete. You can now implement tabular Q-learning from scratch, verify it against value iteration on a known environment, and trust that the implementation is correct before touching a neural network. You've seen, with your own hands, a reward-gaming failure, understood exactly why it happened, and seen how potential-based shaping avoids it. You can run an exploration ablation and read its results rather than taking on faith that ε-decay schedules matter. And you have a reusable five-question template for auditing the MDP design choices in any robotics RL paper you encounter. Chapter 6 takes the question §5.4 raised, what do you do when writing a reward function is too expensive or too error-prone, and answers it with demonstrations instead of rewards.

Chapter 6 — Learning from demonstrations: behavior cloning and imitation learning

Why imitation is the dominant signal in modern robotics

Chapter 5 closed with an honest balance sheet. MDPs and reinforcement learning give you a rigorous training signal, but the signal is a scalar, the scalar is hard to design, and the sample count required to learn from it is large. Most of those samples have to come from a simulator that is not quite the robot you are deploying on. That balance sheet explains what actually powers RT-1 (arXiv:2212.06817), OpenVLA (arXiv:2406.09246), and nearly every commercially relevant robot learning system built in the last five years: demonstrations, not reward. Not instead of reward forever (Chapter 7 returns to deep RL), but as the *primary* signal on which production-grade systems are trained today.

This section makes the case for why that happened, and why it is not simply a matter of RL being difficult. The shift comes down to the economics of data collection, the expressiveness of human demonstrations as a supervisory signal, and the way scale has tipped the tradeoffs toward imitation in ways that were not obvious a decade ago.

The cost of a reward label vs. the cost of a demonstration

Ask yourself what it takes to train a policy to pick a strawberry off a plant and place it in a punnet without bruising it. The task requires detecting a ripe berry, planning a grasp that avoids the stem, applying exactly the right contact force, and releasing it gently. Defining a reward function for this is not impossible; you could reward task completion detected by a camera-based classifier. But writing a reward that is dense enough to get learning started, resistant to reward hacking, and transferable across different berry sizes and positions takes weeks of engineering. Even after all that work, the signal you get is a single bit per episode: success or failure.

A skilled human demonstrator, by contrast, provides a *trajectory* of continuous sensorimotor data: joint positions, end-effector forces, wrist camera images, and gripper state at 50 Hz for the fifteen seconds the task takes. That single demonstration carries roughly 750 labeled time steps, each with an action the expert considered correct for the state observed. The supervisory content per minute of human effort is orders of magnitude higher than a sparse reward. The human already knows how hard to grasp, how to adapt when the berry slips, and when to abort; that information never travels through a reward signal.

This is the core asymmetry. A reward signal labels *outcomes*. A demonstration labels *every decision on the path to the outcome*. For tasks where the path matters as much as the destination, which covers nearly all of robot manipulation, demonstration data is simply a richer supervisory signal per unit of collection effort.

A second asymmetry: demonstration data scales

Chapter 5 discussed sample complexity for reward-driven learning. Q-learning on a 16-cell gridworld converges in hundreds of episodes; Q-learning on a continuous 20-dimensional manipulation state requires millions. The sample count scales with the complexity of the state-action space, not with the number of tasks being learned, and there is no known way around this.

Demonstration data follows a different curve. Collect 100 demonstrations of pick-and-place and you have 100 data

points. Collect 200 and you have 200. The relationship is linear, and the collection is parallelizable: ten teleoperators working simultaneously produce data ten times faster than one. By 2022, the Open X-Embodiment collaboration (arXiv:2310.08864) had aggregated over 1 million demonstration episodes across 22 robot embodiments and 527 skill categories. No RL dataset offers comparable coverage for real-robot manipulation, and there is no obvious path to creating one, because autonomous RL data collection at scale requires autonomous robots already operating in the real world, which is the thing you are trying to build in the first place.

The data flywheel works in favor of imitation. Every hour of teleoperation produces new training data directly usable by a behavior-cloning objective. Every new robot that comes online can contribute to the same dataset, whether or not it shares reward semantics with the others. Aggregation stays simple because demonstration labels are states and actions, which are physically meaningful across embodiments, at least approximately, rather than task-specific scalars.

What "imitation learning" covers

The phrase "imitation learning" gets used broadly in the literature to cover several distinct approaches that share one property: using expert behavior as training signal. It helps to place them upfront.

Behavior cloning (BC) treats demonstration data as a supervised regression problem: given a state, predict the action the expert took. No explicit model of the expert's intentions, no reward function, no environment interaction during training. BC is §6.2's subject, and it is the simplest and most widely deployed variant.

DAgger and variants address BC's main failure mode, compounding error, by interleaving policy execution with expert correction. The learner runs in the environment, and the expert labels the states the learner reaches rather than only the states the expert visited. §6.3 covers this in detail.

Inverse reinforcement learning (IRL) inverts the MDP problem. Rather than assuming a reward and computing the optimal policy, IRL assumes the demonstrations are approximately optimal and infers a reward function that would explain them. That inferred reward can then drive RL. §6.4 examines IRL and its adversarial formulation (GAIL).

Offline RL treats demonstration data as a fixed dataset and runs a modified RL algorithm that stays close to the demonstrated behavior while optimizing a reward signal. It sits between BC and RL in the learning landscape; we visit it briefly in §6.5 when comparing the approaches.

The chapter focuses on BC and DAgger because those two are the workhorses of current VLA training. RT-1, OpenVLA, Octo, $\pi 0$: all are trained primarily with behavior cloning on large demonstration datasets, with no reward signal in the loop. Understanding why BC works at scale, and where it breaks, is prerequisite knowledge for reading any modern VLA paper.

The historical arc: ALVINN to modern VLAs

Tracing how imitation reached this dominant position is worth doing, because the arc is not a smooth one. It involves two distinct inflection points separated by decades.

Dean Pomerleau's ALVINN system (1988) demonstrated autonomous car driving by training a three-layer neural network on 45 minutes of human steering data. Input: a 30×32 image from a forward-facing camera. Output: a single steering angle. Training signal: the human's wheel position at each frame. The network generalized to novel roads at up to 55 mph. This was behavior cloning before it had a name, and it worked remarkably well for 1988.

ALVINN faded from view because it generalized poorly. A network trained in one lighting condition failed in another, and the small dataset meant the model had encountered very few of the situations it would face on deployment. The decade that followed was dominated by explicit rule-based autonomy and then, once DARPA's Urban Challenge revived interest in autonomous driving, by a probabilistic pipeline of perception, mapping, and planning modules: the opposite of end-to-end learning.

The second inflection was the arrival of large, high-quality teleoperation datasets alongside the transformer architecture. The 2022–2023 period delivered both at once: Open X-Embodiment aggregating cross-embodiment data at scale, and RT-1 demonstrating that a transformer trained on 130,000 demonstration episodes could learn a single policy covering over 700 tasks. The difference from ALVINN was not the algorithm (BC in both cases) but the *data*

volume and the *model capacity*. Pomerleau's network had fewer than a thousand parameters; RT-1's had 35 million; OpenVLA's has 7 billion. That gap in representational capacity is what lets modern systems interpolate across diverse demonstrations rather than overfit to narrow distributions.

Why not just use RL then?

A reasonable objection: if RL and BC both work, why not use RL more? RL does not require a human demonstrator on the critical path, and in principle it can discover solutions better than any human demonstration given enough exploration time.

The answer is not that RL is broken. Chapter 7 shows that SAC and PPO work very well in simulation for tasks where reward design is tractable and sample counts are not a constraint. The answer is that the *preconditions* for RL are rarely met in practical robot deployment. You need a reward signal, a simulator, enough samples to explore effectively, and a sim-to-real transfer procedure that works for your task. All four preconditions must hold simultaneously, and for the overwhelming majority of tasks encountered in service and manufacturing robotics, at least one fails.

The Mandlekar et al. (2021) study, "What Matters in Learning from Offline Human Demonstrations for Robot Manipulation," is instructive on the comparison. They found that a BC policy trained on 200 demonstrations outperformed SAC trained from scratch on the same tasks when the reward was sparse, which it is in most real settings. The implication is not that BC is generically better than RL. It is that BC's performance curve reaches acceptable levels at demonstration counts that are actually achievable, while RL's performance curve reaches acceptable levels at sample counts that are not.

The practical synthesis of the field, circa 2024–2025, is to use imitation as the primary training signal and RL as a fine-tuning signal when reward is available and the policy is already close to competent. $\pi 0$ (arXiv:2410.24164) is trained with a BC-style flow-matching objective on demonstration data, then optionally fine-tuned with online RL for specific downstream deployments. This ordering is not accidental. A policy that has never seen a demonstration of the task will explore poorly under RL, because good exploration requires already knowing roughly what good behavior looks like. Starting from BC gives RL a sensible initialization.

What this chapter builds

The remainder of Chapter 6 develops the mechanics of imitation learning in order of increasing sophistication. §6.2 sets up behavior cloning formally and walks through the training loop. §6.3 analyzes the compounding-error failure mode, the reason BC alone is insufficient for tasks with long horizons, and presents DAgger as the fix. §6.4 examines what happens when you want to extract a reward function from demonstrations rather than a policy directly. §6.5 synthesizes the comparison, giving you the decision criteria for choosing between BC, IRL, offline RL, and online RL on a real task.

The framing throughout stays practical: given a dataset, a robot, and a task deadline, which algorithm and how much data. That framing matters because imitation learning is not one algorithm but a family, and picking the wrong member of the family for your constraints is a common and avoidable failure.

Section 6.2 opens the BC training loop and works through it line by line.

Behavior cloning, step by step

Section 6.1 made the economic case for demonstrations. This section makes the algorithm concrete. Behavior cloning is the simplest member of the imitation family, and its simplicity is exactly the point: it is supervised learning, the machinery you already built in Chapter 3, applied to a dataset whose labels happen to be expert actions. There is no environment interaction during training, no reward, no value function. If you can train an image classifier, you can train a BC policy. The subtleties, and there are several, live in the choices you make around that supervised core: how observations are encoded, how actions are represented, which loss you minimize, and how you decide whether the result is any good.

We walk through those choices in the order you would face them on a real project, using a running example: training a policy to lift a block off a table, with the robomimic `lift` dataset from the Mandlekar et al. (2021) study

(arXiv:2108.03298) as the concrete dataset. It contains 200 teleoperated demonstrations of a simulated Franka arm lifting a cube, small enough to train in minutes, real enough to exhibit every failure mode this chapter cares about.

The objective, formally

Let the demonstration dataset be

$$\mathcal{D} = \{(o_i, a_i)\}_{i=1}^{N},$$

where each pair is an observation and the action the expert took when seeing it. The pairs come from slicing expert trajectories into individual time steps; a single 15-second demonstration at 20 Hz contributes 300 pairs. Behavior cloning fits a parameterized policy $\pi_\theta(a \mid o)$ by maximizing the likelihood of the expert's actions:

$$\theta^* = \arg\max_\theta \sum_{i=1}^{N} \log \pi_\theta(a_i \mid o_i).$$

That is the entire algorithm. When the policy is a Gaussian with fixed variance, maximizing log-likelihood reduces to minimizing mean squared error between predicted and demonstrated actions, which is why much of the literature, and most quick-start code, just writes `mse_loss`. Keep the likelihood view in mind anyway; it becomes important the moment a single Gaussian is the wrong distributional choice, which on real robot data is most of the time. We return to that below.

Notice what the objective does *not* contain: any term involving what happens when the policy's actions are executed. BC treats the dataset as i.i.d. samples from some distribution over observations, exactly as image classification treats photos. The dataset is not i.i.d.; it is a set of trajectories, and at deployment time the policy's own actions determine which observations it sees next. That mismatch is the deepest problem in imitation learning, and it gets its own section (§6.3). For now we make the standard move of ignoring it, which works better than it has any right to, provided the data is plentiful and the horizon is short.

Step 1: know what is in your dataset

A teleop dataset is a set of trajectories, each a time-indexed record of observations and actions. For robomimic `lift`, one time step contains: an 84×84 RGB image from a front camera, a second image from a wrist camera, the 7-dimensional joint configuration, the end-effector pose, the gripper aperture, and the 7-dimensional action the operator commanded (6-DoF end-effector velocity plus gripper open/close). Multiply by roughly 50 steps per demonstration and 200 demonstrations: about 10,000 training pairs.

Before writing any model code, look at the data. Plot action distributions per dimension. You will find the gripper channel is nearly binary and the translation channels are roughly zero-mean with occasional large spikes. Replay a few demonstrations and watch them. The Mandlekar study found that *demonstrator quality*, whether the data came from one practiced operator or many inconsistent ones, moved task success by tens of percentage points, more than most architectural choices. A BC policy is a mirror: it reproduces the dataset's habits, including hesitation, retries, and the operator's idiosyncratic approach angles. No later step compensates for confused data.

Step 2: choose the observation encoding

The policy needs a function from raw observations to a feature vector. Three standard options, in increasing order of capability and cost:

Low-dimensional state. Concatenate joint angles, end-effector pose, and object poses (available in simulation) into one vector and feed an MLP. Trains in minutes, useless on a real robot where object poses are not directly observable.

Visual encoder from scratch. A small CNN over the camera images, trained jointly with the policy head. This is what most robomimic baselines do, and at the 200-demonstration scale it works because the visual variety is low: one table, one cube, one lighting condition.

Pretrained visual backbone. A frozen or fine-tuned encoder pretrained on internet-scale data is the route every modern VLA takes. OpenVLA (arXiv:2406.09246) uses fused SigLIP and DINOv2 features feeding a 7B-parameter language-model backbone. The pretrained option buys robustness to visual variation your demonstrations never covered, and Chapter 11 traces how this became the default recipe.

For the running example, a four-layer CNN per camera, features concatenated with proprioception, is plenty.

Step 3: choose the action representation

This choice matters more than newcomers expect, and Chapters 10–13 are in large part a study of its consequences. The options:

Continuous regression. The network outputs a 7-vector directly; train with MSE. Simplest, and the right starting point for `lift`.

Discretized tokens. Bin each action dimension (RT-1, arXiv:2212.06817, uses 256 bins per dimension) and predict bins with a cross-entropy loss. This turns control into classification, which plays well with transformer architectures and, critically, can represent multimodal action distributions, since a softmax over bins can place mass on two distant bins at once.

Action chunks. Predict the next k actions (say, 16 steps) as one output instead of one step at a time. Chunking reduces compounding error by cutting the number of decisions per episode, and smooths control; it is standard in ACT and Diffusion Policy (Chapter 10).

Generative heads. Represent $\pi_\theta(a \mid o)$ with a diffusion or flow-matching model that can sample from arbitrarily complex action distributions. This is the π0 route (arXiv:2410.24164) and the subject of Chapter 13.

Why so much machinery for what regression handles in one line? Because demonstrations are multimodal. Suppose half your demonstrators pass left around an obstacle and half pass right. MSE-trained regression predicts the average, straight into the obstacle. Each option above the first is, at heart, a way of representing "left or right, but not the mean" as a distribution. On `lift` the demonstrations are single-moded enough that plain regression works; on almost any longer-horizon real-robot dataset, it eventually does not.

Step 4: the training loop

With encoder and action head chosen, the loop is the one from §3.3 with the labels swapped:

```
policy = BCPolicy(obs_encoder, action_dim=7)
opt = torch.optim.AdamW(policy.parameters(), lr=1e-4)

for epoch in range(num_epochs):
    for obs, act in loader:            # (B, ...), (B, 7)
        pred = policy(obs)             # (B, 7)
        loss = F.mse_loss(pred, act)
        opt.zero_grad()
        loss.backward()
        opt.step()
```

Three practical notes save a day each. Normalize actions per dimension to zero mean and unit variance using *dataset* statistics, and store those statistics with the checkpoint; at deployment the network's outputs must be un-normalized with the same numbers, and a mismatch produces a policy that moves confidently in the wrong direction. Weight the gripper dimension, or give it its own binary cross-entropy head: it is one of seven dimensions, but failing to close it is 100% of task failure. Use augmentation (random crops, color jitter) on images too; with 10,000 frames of one table, the encoder will otherwise memorize pixels.

Step 5: evaluation, the part that surprises people

Here is the trap: validation loss on held-out (observation, action) pairs is only weakly correlated with task success. The Mandlekar study documents checkpoints whose validation MSE is nearly identical while their success rates differ by 20 points, and the lowest-validation-loss checkpoint is rarely the best policy. The reason is the i.i.d. fiction from the objective: validation pairs are drawn from *expert* trajectories, but at deployment the policy visits its own states. A policy can match the expert almost everywhere and still drift into states the dataset never covered, where its behavior is undefined.

The only evaluation that counts is the rollout: execute the policy from fresh initial conditions and measure success over enough episodes to get a meaningful estimate. Fifty is a common floor, and Chapter 15 treats the statistics properly. The practical consequence is that checkpoint selection must be done by rollout, in simulation if you have one. This is the first place BC's "it's just supervised learning" slogan breaks down. The training is supervised; the evaluation is not.

Run the full recipe on `lift`, CNN encoder, MSE regression, 200 demonstrations, rollout-based checkpoint selection, and you get a policy succeeding roughly 90% of the time, matching the robomimic baseline. Scale the same recipe up, more data, bigger encoder, discretized actions, and you arrive, with surprisingly few new ideas, at RT-1: 130,000 episodes, 35M parameters, over 700 tasks, and the same maximum-likelihood objective we wrote at the top of this section.

What the recipe does not fix is the drift problem we waved away after the objective: each small imitation error moves the policy slightly off the expert's distribution, where errors get larger, which moves it further off. Section 6.3 quantifies that feedback loop, and shows why its cost grows quadratically with the horizon.

Compounding error and DAgger

Section 6.2 ended on a debt: we wrote the BC objective as if (observation, action) pairs were i.i.d. samples, knowing they are not. This section pays that debt. The mismatch between training on expert states and deploying on the policy's own states is not a technicality. It is the central failure mode of behavior cloning, it has a name (*covariate shift*, or in the imitation literature, *distribution shift*), and its cost can be quantified: a per-step imitation error of ε costs you on the order of εT^2 over a horizon of T steps, not εT. We derive the intuition for that quadratic, then present the classical fix, DAgger, and examine why modern VLA training uses DAgger's *idea* far more than its *algorithm*.

The feedback loop nobody trained for

Start with the oldest concrete example in the field. ALVINN (Pomerleau, 1988) learned to steer from 45 minutes of human driving. A human driver, being competent, keeps the car centered in the lane. Consequently the training data contains thousands of frames of well-centered road and essentially zero frames of the car drifting onto the shoulder, because the expert never let that happen. Now deploy the network. It is a function approximator, so it makes small errors; suppose it steers slightly wide on one curve. The camera now shows a view the dataset barely covers: lane markings at an angle the expert never produced. On this off-distribution input, the network's output is less reliable, so the next steering command is likely worse, which produces a still-stranger view, and the loop runs away. One small error did not cost one small penalty. It relocated the policy to a region of state space where *all* its subsequent decisions are unreliable.

Pomerleau saw this on real roads and patched it with a hack that has aged remarkably well: he synthetically shifted and rotated the camera images to simulate the car being off-center, and computed the corrective steering label geometrically. ALVINN was trained on recovery behavior the human never demonstrated. Hold that thought; the modern equivalents of this trick still do a lot of quiet work in current pipelines.

The general structure is worth stating plainly, because it separates imitation learning from every supervised problem you have met so far. In image classification, your prediction does not change the next image you are shown. In control, it does. The policy's inputs at time $t+1$ are a consequence of its output at time t. Errors do not average out over a trajectory; they accumulate, and worse, they *correlate*: each error increases the probability of the next.

Why the cost is quadratic in the horizon

Ross and Bagnell (2010) made the accumulation precise, and the argument fits in a paragraph. Suppose the learned policy disagrees with the expert with probability at most ε on states drawn from the *expert's* distribution; this is what your validation loss (imperfectly) measures. Walk the trajectory step by step. At each step where the policy has so far behaved like the expert, it errs with probability ε. But once it has erred, all bets are off: it is now off the expert's distribution, where its error rate is unbounded, and the analysis can only assume the worst for the remaining steps. An error at step 1 can poison $T-1$ subsequent steps; an error at step T poisons none. Summing over the horizon, expected total cost is bounded by $\varepsilon \cdot T^2$ in the worst case, and the bound is tight: there are MDPs that actually realize it.

Contrast the supervised baseline. If the problem really were i.i.d., if someone handed the policy states drawn from the expert distribution at every step regardless of its past actions, total cost would be εT. The gap between εT and εT^2 is the price of the feedback loop, and it explains an everyday observation from §6.2: BC policies look great on short-horizon tasks and degrade sharply on long ones. Lifting a cube is a 50-step problem; the policy that succeeds 90% of the time there will not survive a 500-step kitchen-tidying task with ten times the polish. The horizon enters *squared*.

It is worth being honest about what the bound does and does not say. It is a worst case; real environments are often forgiving, with recoverable states and self-correcting dynamics (a slightly misaligned gripper above a cube often still descends into a workable grasp). The quadratic is not destiny. But it identifies the right villain: not the size of ε, which more data and bigger models steadily shrink, but the *support* of the training distribution, meaning what the policy has never seen.

DAgger: ask the expert about the learner's mistakes

If the problem is that the dataset only covers expert states, the fix suggests itself: get labels on the states the *learner* visits. DAgger, Dataset Aggregation, from Ross, Gordon and Bagnell (2011), turns that into an algorithm:

```
D ← initial expert demonstrations
π₁ ← train BC on D
for i = 1, 2, ..., N:
    roll out πᵢ in the environment
    for every state s visited, query the expert's action a*(s)
    D ← D ∪ {(s, a*(s))}          # aggregate, never discard
    πᵢ₊₁ ← train BC on D
return best πᵢ under rollout evaluation
```

Two details matter. First, the *learner* drives and the *expert* labels. The states in the new data are exactly the off-distribution states that the quadratic bound worried about, including the botched approaches and near-misses that no expert demonstration contains. After a few iterations, the dataset covers the learner's mistakes and, crucially, the expert's corrections from them. Second, the aggregation: each policy is retrained on the union of *all* data so far, not just the latest batch. This is what supports the theory. DAgger is analyzed as a no-regret online learning procedure, and the punchline is that the quadratic goes away: with enough iterations, expected cost is $O(\varepsilon T)$ plus terms that shrink with iteration count. Linear in horizon, like proper supervised learning. (In early iterations the rollout policy is sometimes mixed with the expert, executing the expert's action with probability β_i, decayed toward zero, to keep the first, worst policies from spending whole episodes in useless corners of state space.)

On the robomimic `lift` setup from §6.2 the effect is easy to reproduce in simulation, because there the "expert" can be a scripted or pretrained policy that is queryable for free: a BC policy trained on 200 demonstrations and stuck at 90% will, with two or three DAgger iterations of a few dozen rollouts each, typically clear the failures caused by drift, the slow sideways slide of the gripper that ends centimeters from the cube, because the dataset now contains exactly those slides, labeled with the correction.

The catch: who is this expert that answers queries?

Now the bad news, which is the reason this section is not titled "DAgger: problem solved." DAgger's expert must label *arbitrary states on demand*. In simulation with a scripted expert, fine. With a human expert and a real robot, the query model is awkward in two distinct ways.

The first is cost. Expert effort is per-state, on the critical path, every iteration; the parallel, offline collection economics that §6.1 credited for imitation's dominance do not apply.

The second is subtler and worse: humans are bad at the query itself. Shown a frozen frame of a mid-failure state and asked "what action would you take here?", a teleoperator gives noisy, mutually inconsistent answers. Humans demonstrate well in closed loop, with the robot responding, and label poorly out of context. Practical variants therefore restructure the interaction: let the human watch the policy run and *take over* when it goes wrong (gated or intervention-based DAgger), so the expert provides closed-loop corrections only on the segments where the policy actually needs help. The takeover states are precisely the off-distribution states DAgger wants labeled, and the human supplies them by demonstrating, which humans are good at, rather than by annotating, which they are not.

If that interaction pattern sounds familiar, it should: it is a deployment data flywheel. A fleet of robots runs the current policy at customer sites; human operators intervene on failures; the interventions are logged and folded into the next training run. That is intervention-based DAgger at industrial scale, and it is, as far as public information allows one to tell, roughly how commercial robot fleets improve after deployment. The algorithm from a 2011 AISTATS paper survives as an operations playbook.

What VLA training actually does about compounding error

Look at the training recipe of RT-1 (arXiv:2212.06817) or OpenVLA (arXiv:2406.09246) and you will find no DAgger loop, just pure BC on a fixed dataset. Are the theorists wrong, or are the practitioners lucky? Neither. Modern pipelines attack the same villain, training support, by other means, and it is worth naming them, because they look like unrelated engineering choices until you see them through this section's lens.

Coverage by brute diversity. A dataset like Open X-Embodiment (arXiv:2310.08864), with a million episodes from dozens of buildings, operators, and embodiments, has vastly wider state support than 200 demonstrations of one cube on one table, including plenty of accidental near-failure states, since across thousands of hours operators wobble, retry, and recover. The recoveries that Pomerleau synthesized geometrically, scale collects by accident.

Shorter effective horizons. Action chunking (§6.2, and properly in Chapter 10) predicts k steps per decision, cutting the number of closed-loop decisions per episode by a factor of k. If the cost is quadratic in the number of decisions, chunking buys a k^2 improvement in the worst case, a large part of why ACT and Diffusion Policy hold up on long-horizon manipulation.

Robust features. A pretrained visual backbone maps superficially novel observations, new lighting, new clutter, near familiar ones in feature space, so the policy is effectively "on-distribution" in feature space more often than raw pixel statistics would suggest. This shrinks how much off-distribution drift it takes to reach truly undefined behavior.

None of these eliminates compounding error; they postpone it. Push any current VLA on a long enough task (Chapter 15's evaluation chapter shows this quantitatively on LIBERO) and the familiar signature reappears: success rates that decay with episode length, failures that begin with one small slip followed by increasingly confident nonsense. When you see that signature, you now know its name, its scaling law, and the two families of remedy: widen the data, or shorten the loop. And when fine-tuning a VLA on your own robot in Chapter 16, the single highest-value data you can collect is intervention data, DAgger's idea, wearing work clothes.

There remains a question BC and DAgger both dodge: they clone *what* the expert does without ever asking *why*. The next section takes up the alternative, inferring the reward the expert seems to be optimizing, and letting the policy pursue that instead.

A glance at IRL and adversarial imitation

Behavior cloning, and the DAgger machinery built on top of it, share a blind spot: they copy *what* the expert does and never ask *why*. A cloned policy reproduces the expert's actions on states it has seen and flails everywhere else, because it has learned a mapping from observations to actions, not the objective that mapping was serving. Inverse reinforcement learning (IRL) flips the problem around. Instead of fitting the policy directly, it asks what reward function would make this expert's behavior optimal. Recover that reward, and you can hand it to any of the RL machinery from Chapter 5 to produce a policy, one that, in principle, generalizes to states the expert never visited, because it is pursuing the expert's *goal* rather than mimicking the expert's *reflexes*.

This section is a glance, not a manual. IRL and its modern adversarial descendants form a large subfield, and most production VLAs do not use them. But the ideas explain a real limitation of cloning, they connect imitation back to the reward-centric view of Chapter 5, and adversarial imitation in particular keeps resurfacing in robot-learning research. You should know what the words mean and when the approach earns its considerable extra cost.

Why recover a reward at all

Consider a concrete contrast. An expert demonstrates parking a car: it approaches the spot, reverses, straightens, stops. A BC policy trained on these trajectories learns "when the view looks like *this*, turn the wheel like *that*." Move the spot two meters, change the approach angle, and the policy is off-distribution, §6.3's failure mode. Now suppose instead you could recover the reward the driver was optimizing: something like "end up inside the lines, parallel to the curb, without hitting anything, with minimal maneuvering." That reward is *portable*. It scores any state in any parking lot. Plan or learn a policy against it and you get sensible behavior in configurations no demonstration covered, because the objective, not the trajectory, transfers.

That is the promise. The reward is a vastly more compact and generalizable description of a task than a pile of trajectories, in the same way that "minimize travel time subject to traffic laws" is a more useful description of good driving than ten thousand recorded drives. IRL tries to extract that compact description from the drives.

The catch, and it is a deep one, is that the problem is badly ill-posed. Ng and Russell (2000), the paper that named the field, opened by observing that infinitely many reward functions make any given policy optimal, including the degenerate reward that is zero everywhere, under which *every* policy, the expert's included, is trivially optimal. Demonstrations underdetermine the reward. Every IRL algorithm is, at heart, a different answer to the question "which of the many consistent rewards should we pick?", and the quality of that answer is what separates the methods.

From feature matching to maximum entropy

The first practical answers came from Abbeel and Ng (2004) under the banner of *apprenticeship learning*. Assume the reward is linear in some features of the state; for driving, features might be "distance from lane center," "speed," "proximity to other cars." A policy's value under such a reward depends only on its *expected feature counts*: how much lane-deviation it accumulates, how fast it tends to go, and so on, in expectation over trajectories. The algorithm then searches for a policy whose expected feature counts match the expert's. If your features capture what matters, a policy that drives with the same average lane-deviation and speed profile as the expert is, for practical purposes, driving like the expert, without anyone ever writing down the weights that trade those features off.

Feature matching left one nagging slack: many policies and many reward weightings match a given set of feature counts, so which do you commit to? Ziebart et al. (2008) supplied the answer that became the field's workhorse: *maximum entropy* IRL. Among all trajectory distributions consistent with the expert's feature counts, pick the one that is otherwise as random, as high-entropy, as possible. This is the same principle that picks the least-committal probability distribution subject to known constraints, imported into trajectory space. It has a clean probabilistic reading: trajectories are assumed to occur with probability proportional to $\exp(\text{reward})$, so high-reward behavior is exponentially more likely but nothing is ever assigned probability zero. MaxEnt IRL handles the suboptimality and noise in real human demonstrations gracefully (the expert is allowed to be imperfect), and it removed the arbitrary tie-breaking that plagued earlier methods. For roughly a decade it was the default, and it still underlies much of what follows.

The practical wall MaxEnt IRL hit is computational. Recovering the reward requires, in its inner loop, solving the *forward* RL problem, finding the optimal policy for the current reward estimate, and doing so repeatedly as the reward is refined. In a small discrete world this is a tabular dynamic-programming sweep (Chapter 5). On a robot with continuous states and unknown dynamics, each inner solve is itself a full, expensive RL run. IRL was, for years, an algorithm that contained reinforcement learning as a subroutine, which made it roughly as hard as RL times the number of reward updates.

Adversarial imitation: skip the reward, match the distribution

The breakthrough that made imitation-via-objectives practical at scale came from reframing it as a *distribution-matching* problem and borrowing the machinery of generative adversarial networks. Generative Adversarial Imitation Learning, GAIL, Ho and Ermon (2016), is the pivot, and the idea is elegant enough to state in one breath: train a *discriminator* to tell expert state-action pairs apart from the learner's, and train the *policy* to fool it.

The two play the familiar adversarial game. The discriminator D is a classifier outputting the probability that a given (s, a) came from the expert rather than the policy. The policy is rewarded for producing state-action pairs the discriminator mistakes for expert data; concretely, the policy's reward at each step is something like $-\log(1 - D(s, a))$, high when D is fooled. As the policy improves, the discriminator is retrained to find the remaining tells; as the discriminator sharpens, the policy is pushed to close the remaining gaps. At equilibrium the learner's state-action distribution is indistinguishable from the expert's, which is exactly the goal: matching the occupancy of the demonstrations, not echoing individual actions.

```
initialize policy π, discriminator D
for each iteration:
    roll out π to collect trajectories
    update D to classify expert (s,a) vs. policy (s,a)
    set per-step reward r(s,a) = -log(1 - D(s,a))
    update π with an RL step (e.g., PPO) to maximize that reward
```

Two things make this matter for robotics. First, GAIL never explicitly recovers a reward function; the discriminator *is* the reward, learned and updated on the fly, which sidesteps the ill-posedness that haunted classical IRL. Second, and decisively, the policy update is just an ordinary RL step (PPO, from Chapter 7, is the standard choice). So instead of solving a full RL problem inside every reward update, GAIL interleaves *one* RL step with *one* discriminator step. It is dramatically more sample-efficient in expert demonstrations than behavior cloning; a handful of trajectories can suffice, at the cost of needing many environment interactions for the RL inner loop, which is why GAIL lives mostly in simulation.

The structural payoff against §6.3 is worth naming. BC only ever sees expert states; GAIL's policy is rolled out in the environment, so the discriminator scores the learner's *own* visited states, including the off-distribution ones, and pushes the policy back toward expert-like behavior there. The mechanism that makes covariate shift catastrophic for BC is, for GAIL, simply part of the training loop. It buys robustness to distribution shift in exchange for online interaction.

If you want the recovered reward back, because a discriminator that classifies (s, a) pairs entangles the task objective with the particular dynamics it was trained under, and so transfers poorly to a new robot or a changed environment, Adversarial Inverse Reinforcement Learning (AIRL, Fu, Luo and Levine 2018) restructures the discriminator so that a genuine, dynamics-disentangled reward falls out of it. That reward is portable in the way the parking example wanted: train in one setting, recover the reward, re-optimize in another.

Where this sits relative to VLAs

Now the honest accounting. Open the training recipe of RT-1 (arXiv:2212.06817), OpenVLA (arXiv:2406.09246), or Octo (arXiv:2405.12213) and you will find no IRL, no discriminator, no adversarial game. They are behavior cloning at scale, for reasons §6.1 laid out. Adversarial imitation requires online environment interaction and a stable two-player optimization, and both are liabilities at foundation-model scale. The GAN-style minimax is notoriously finicky to tune, and "collect a million teleop episodes and fit them offline" is operationally far simpler than "stand up a simulator

the policy can safely explore in for millions of steps while two networks chase each other." When demonstrations are abundant, cloning their actions is cheaper and more stable than inferring their intent.

So why spend a section on it? Three reasons. First, IRL is the cleanest statement of what cloning gives up, the objective behind the behavior, and naming that gap sharpens your judgment about when cloning will generalize and when it will not. Second, adversarial distribution matching is the right tool precisely when demonstrations are scarce but a simulator is cheap, the inverse of the foundation-model regime; it recurs in sim-heavy locomotion and dexterous-manipulation research for exactly that reason. Third, the distribution-matching framing, making the learner's occupancy look like the expert's rather than copying actions pointwise, is the conceptual seed of ideas you will meet again when offline RL re-enters the picture, and it is the cleanest bridge from "imitate the expert" to "optimize a reward," which is where the next section's decision lives.

This section deliberately stayed at the level of ideas; the algorithms here each deserve their own chapter and will not get one, because the book's spine runs through cloning, not reward inference. With behavior cloning (§6.2), its compounding-error failure mode and DAgger fix (§6.3), and now reward-inference and adversarial alternatives in view, you are equipped for the practical question that closes the chapter: given a particular dataset and a particular robot, which of behavior cloning, inverse reinforcement learning, and ordinary reinforcement learning should you actually reach for first?

Choosing between BC, IRL, and RL

You have now seen four ways to turn experience into a policy: behavior cloning (§6.2), DAgger (§6.3), inverse reinforcement learning and its adversarial cousin GAIL (§6.4), and, from Chapter 5, ordinary reinforcement learning against a hand-specified reward. A fifth, offline RL, was named in §6.1 and gets its due below. The natural question after meeting all of them is the one a working engineer actually faces: given *this* dataset, *this* robot, and *this* deadline, which do I reach for first?

The honest short answer, and the one the rest of this book is built around, is that you almost always start with behavior cloning. But "start with BC" is a default, not a law, and the point of this section is to make the default legible: to say precisely which question each method answers, what each one demands from you in return, and what observations should push you off the default toward something more expensive.

The three questions that decide it

Every method in this chapter is, underneath, an answer to a different question about what you have and what you can get.

Behavior cloning asks: *do I have demonstrations of the behavior I want?* If yes, and if I will mostly be deploying in situations the demonstrations cover, BC fits a state-to-action map and is done. It needs no reward, no simulator, and no online interaction. Its weakness is the one §6.3 dissected: it has no mechanism to recover from states the demonstrations never visited, so its error compounds over a long horizon.

DAgger asks a follow-up: *can I put an expert back in the loop while the learner runs?* If a human (or a privileged scripted controller) can label the states the policy actually reaches, DAgger closes BC's covariate-shift gap directly. It needs interactive expert access during training, cheap in simulation, expensive and sometimes unsafe on hardware.

Inverse RL and GAIL ask: *do I want the expert's objective rather than the expert's reflexes?* If the task will be deployed in configurations no demonstration covers, and a portable reward would generalize where a cloned trajectory will not, reward inference earns its cost. The price, from §6.4, is a simulator the policy can explore in for many steps and a two-player optimization that is finicky to stabilize.

Reinforcement learning asks the bluntest question of all: *do I have a reward and a place to fail safely?* If you can write a reward and afford millions of trial-and-error samples, which in practice means a simulator, RL can discover behavior no demonstrator produced. Chapter 5 laid out why those two preconditions are rarely both met on a real robot.

Notice that the questions form a rough ladder of cost. BC asks only for data you may already have. Each rung above it asks for something harder to obtain, interactive expert access, a simulator, a reward function, in exchange for a

specific kind of generalization that the rung below cannot give you. The engineering discipline is to climb no higher than your problem forces you to.

Where offline RL sits

The ladder has a rung between pure cloning and full online RL that is worth naming on its own, because it is increasingly where serious robot systems live. *Offline RL* uses a fixed dataset, the same demonstrations BC would consume, but optimizes a reward signal against it, without ever interacting with the environment. It answers a question BC cannot address: *my demonstrations are mixed quality, and I have at least a sparse reward; can I do better than copying the average?*

BC copies every demonstration with equal weight, good and bad alike; if half your teleop episodes fumbled the grasp, BC dutifully learns to fumble half the time. Offline RL, given a reward that distinguishes the good episodes from the bad, can prefer the actions that led to high return and downweight the rest. The catch is subtle and was the field's central difficulty for years: a value function trained on a fixed dataset will confidently overestimate the value of actions the dataset never contains, and a naive offline RL run chases those phantom values straight off a cliff. Conservative methods such as CQL (Kumar et al. 2020) fix this by explicitly penalizing value estimates for out-of-distribution actions, keeping the learned policy honest about what the data actually supports. The result is a method that extracts more than BC from the same fixed dataset when a reward is available, while sidestepping the safety and sample problems of online exploration. It is the natural choice when you have demonstrations *and* a reward but no safe way to collect fresh experience.

A worked decision

Make it concrete. Suppose you are deploying a tabletop arm that must pick assorted parts from a bin and place them on a fixture, and you have collected 300 teleop demonstrations.

Start at the bottom of the ladder. You have demonstrations, so behavior cloning is the first thing to train: it is a day of work, it needs nothing you do not already have, and it gives you a baseline that tells you how hard the task actually is. Mandlekar et al. (2021, arXiv:2108.03298) is the empirical anchor here: across a suite of manipulation tasks, BC on a few hundred human demonstrations was a strong baseline, and outperformed RL trained from scratch under the sparse rewards typical of real tasks. Do not skip the baseline on a hunch that you need something fancier; measure first.

Now read the failure mode. If the BC policy succeeds on parts and poses like those in the data but drifts and stalls on novel ones, you are looking at compounding error (§6.3), and the cheapest fix is more coverage, either more demonstrations of the failure regions, or DAgger if you can afford to have an operator correct the policy as it runs. Reach for DAgger before reaching for reward inference; it solves the covariate-shift problem you actually have without forcing you to invent a reward.

You climb past DAgger only when the problem changes character. If the fixture moves to positions no demonstration covered, and you want the policy to *figure out* a placement rather than interpolate a remembered one, you are now asking for the expert's objective, not the expert's trajectories, the §6.4 situation, and IRL or GAIL becomes defensible, assuming you can stand up a simulator to optimize in. And if your demonstrations are a mixed bag and you happen to have a reward, say a force sensor that flags bruised parts, offline RL lets you prefer the gentle episodes over the rough ones without discarding the rough ones entirely.

The thing to resist is starting at the top. Online RL from scratch on this task would mean writing a dense bin-picking reward (weeks, per §5.4), building a simulator faithful enough to transfer (§5.5), and burning millions of samples, to solve a problem that 300 demonstrations and an afternoon of BC may largely dispatch.

A checklist you can actually apply

Strip the discussion to the signals that should move your decision:

- **You have demonstrations and will deploy near them**. Behavior cloning. This is the default and it is the default for good reasons.

- **BC works in-distribution but drifts off it, and you can correct it live.** Add DAgger. Cheapest possible fix for compounding error.
- **Your demonstrations are mixed quality and you have a reward.** Offline RL (e.g., CQL). Squeezes more out of the same fixed data without online risk.
- **You need the expert's goal to transfer to unseen configurations, and you have a simulator.** IRL / GAIL. You are buying a portable objective, at the cost of a two-player optimization.
- **You have a reward, a simulator, and a sample budget, and want behavior no human demonstrated.** Online RL (Chapter 7). The most powerful and the most demanding.
- **You have a competent BC policy and want to push past human performance on a specific deployment.** BC-then-RL fine-tuning. This is the pattern modern systems actually use.

That last row is where the field has converged, and it is worth stating plainly because it dissolves the false choice the section title implies. The production answer is rarely "BC *or* RL." It is BC *then* RL: pretrain a policy on a large demonstration corpus, where imitation's data-scaling advantage (§6.1) does the heavy lifting, then fine-tune with online RL on the narrow slice of behavior where a reward is cheap to specify and the policy is already competent enough to explore sensibly. $\pi 0$ (arXiv:2410.24164) is exactly this: a flow-matching BC objective on broad demonstration data, with optional RL fine-tuning per deployment. The imitation signal teaches the policy what good behavior roughly looks like; the reward signal sharpens it where it matters. Neither alone is enough, and the ordering is not arbitrary. A policy that has never seen the task explores hopelessly under RL, the same observation §6.1 made and the reason this book spends four chapters on imitation before returning to deep RL.

Why the default holds for VLAs

For the foundation action models this book is building toward, the ladder collapses almost entirely onto its bottom rung, and it is worth being explicit about why. Foundation VLAs are trained across hundreds of tasks and dozens of embodiments at once. There is no single reward function that spans "fold a towel," "stack blocks," and "wipe a counter," so reward inference and online RL, both of which presuppose a task-specific objective, do not scale to the multi-task, cross-embodiment regime. Behavior cloning does, because its supervisory signal is just states and actions, which are comparable across tasks and robots in a way that scalar rewards are not. That is the structural reason RT-1, OpenVLA, Octo, and $\pi 0$ are all, at their core, cloning at scale. The methods higher on the ladder do not disappear; they re-enter as *fine-tuning* tools for a single robot and a single task, which is precisely the subject of Chapter 16.

With this section, the chapter's core question is answered. You can now train a behavior-cloning policy and read its failure modes; explain compounding error and the DAgger fix in your own words; distinguish cloning from reward inference and from offline RL, and say what each demands; and decide, given a dataset and a robot, whether BC is the right place to start, which, more often than not, it is. The summary that follows consolidates these into the handful of claims worth carrying into the rest of the book.

6.6 Summary

Chapter 6 was the imitation chapter, and its through-line is the claim that opened it: for modern robot manipulation, demonstrations are the dominant training signal, not reward. Every section developed one facet of that claim: why imitation won, how the simplest version works, where it breaks, what the more sophisticated alternatives buy you, and how to choose among them. This summary collects the load-bearing ideas and marks which ones Parts 3 and 4 will keep reaching for.

The four ideas worth carrying forward

A demonstration labels every decision on the path; a reward labels only the outcome. §6.1 built the case for imitation on two asymmetries. The first is informational: a single fifteen-second teleoperation episode at 50 Hz carries hundreds of state-action labels, each one an action the expert judged correct for the state observed, whereas a sparse reward delivers a single bit per episode. The second is economic: demonstration data scales linearly and parallelizes across teleoperators and embodiments, which is how the Open X-Embodiment collaboration (arXiv:2310.08864) aggregated over a million episodes, while reward-driven data collection at scale requires the autonomous robot you are trying

to build in the first place. These two asymmetries are the reason RT-1 (arXiv:2212.06817), OpenVLA, Octo, and $\pi 0$ are all trained by cloning, and they are the reason this part of the book spends four chapters on imitation before returning to deep RL.

Behavior cloning is supervised regression, and that is both its strength and its trap. §6.2 reduced BC to its essence: collect (state, action) pairs, fit a network to predict the action from the state, minimize a standard supervised loss. No reward, no simulator, no environment interaction during training. That reduction is why BC is the first thing you train on any new task; it inherits the entire mature toolkit of supervised learning. But the same reduction hides a violated assumption. Supervised learning assumes the test inputs are drawn from the training distribution, and a deployed policy violates that assumption the instant its own actions steer it somewhere the expert never went. Recognizing BC as supervised learning *with a broken i.i.d. assumption* is the insight that makes its central failure mode predictable rather than mysterious.

Compounding error is the structural failure of cloning, and DAgger is the structural fix. §6.3 made the failure quantitative: a small per-step probability of error drives the policy off the expert's state distribution, where its error rate is even higher, so mistakes compound across a horizon rather than averaging out. The cost grows with the *square* of the horizon in the worst case, not linearly, which is why BC that looks fine on short tasks falls apart on long ones. DAgger (Ross, Gordon & Bagnell, 2011) attacks the cause directly: roll the learner out, have the expert label the states the learner *actually visits*, and aggregate those corrections into the training set. It converts the i.i.d.-violating deployment distribution into part of the training distribution. The price is interactive expert access during training, cheap in simulation, expensive and sometimes unsafe on hardware, which is exactly the tradeoff §6.5 asked you to weigh.

Cloning copies behavior; reward inference recovers intent, and which you want depends on what has to generalize. §6.4 introduced inverse reinforcement learning and its adversarial descendant GAIL (Ho & Ermon, 2016) as the answer to a question BC cannot pose: what objective was the expert pursuing? A recovered reward is portable in a way a cloned trajectory is not; it scores configurations no demonstration covered, which is precisely why it is worth its considerable extra cost when the deployment distribution will stray far from the data. But IRL presupposes a simulator to optimize in and a two-player optimization that is finicky to stabilize, which is why no production VLA uses it and why §6.5 placed it high on a cost-ordered ladder you should climb only when the problem forces you to.

What you should be able to do now

Four concrete capabilities, in roughly the order later chapters will need them.

You should be able to *implement behavior cloning on a demonstration dataset and read its training curves honestly.* Given a set of (observation, action) pairs, set up a network with the right action parameterization, choose a loss that matches it, and train to convergence, the §6.2 loop. The honest part is interpretation: a low training loss tells you the policy fits the demonstrations, and it tells you nothing about whether the policy will hold together on its own rolled-out states. The §6.x exercise drives this home by having you train a BC policy whose offline loss looks healthy and whose rollout success rate does not, which is the single most common surprise for newcomers to imitation learning.

You should be able to *explain compounding error and DAgger in your own words, with the horizon argument intact.* Not "BC drifts" but *why* it drifts: the policy's own errors move it off the expert's state distribution, error rates are higher off-distribution, and the effect accumulates super-linearly in the horizon. And not "DAgger helps" but *how*: by labeling the learner's own visited states with expert actions, it folds the deployment distribution back into training. This is the chapter's named learning objective from the TOC, and being able to state it cleanly is the difference between reading a VLA paper's data-collection section as a list of arbitrary choices and reading it as a set of answers to compounding error.

You should be able to *distinguish behavior cloning from inverse reinforcement learning and from offline RL, and say what each demands.* BC fits actions from a fixed dataset and asks for nothing else. Offline RL optimizes a reward from a fixed dataset and asks for a reward plus a way to stay honest about out-of-distribution actions (CQL, §6.5). IRL recovers a reward and asks for a simulator. Naming the demand each method makes, not just the mechanism, is what lets you rule methods out quickly: no simulator means no IRL, no reward means no offline RL, and what is left is cloning.

You should be able to *decide, given a dataset and a robot, whether BC is the right place to start*, and recognize that the answer is usually yes. §6.5's ladder gives the procedure: start with BC because it asks only for data you may already have, climb to DAgger when in-distribution success coexists with off-distribution drift and you can correct live, climb to offline RL when demonstrations are mixed-quality and a reward exists, and reserve IRL and online RL for when the goal must transfer to unseen configurations and you have a simulator. The discipline is to climb no higher than the problem forces.

Where the chapter has set up the rest of the book

Chapter 6 hands off in three directions. The most immediate is Chapter 7, deep RL for control, the other half of the choice §6.5 framed. Chapter 6 explained why imitation is the *primary* signal; Chapter 7 explains the RL machinery that fine-tunes a cloned policy past human performance when a reward and a simulator are available, the BC-then-RL pattern that §6.5 identified as where the field actually converged, and that π0 (arXiv:2410.24164) instantiates.

The deeper handoff is to Part 4. Every foundation VLA the book builds toward is, at its core, the behavior cloning of §6.2 scaled to hundreds of tasks and dozens of embodiments. When Chapter 11 introduces RT-1's training recipe, and Chapter 12 introduces OpenVLA's and Octo's, the reader who internalized this chapter will recognize the objective immediately: it is supervised regression on demonstrations, with the chapter's machinery, action parameterization (§6.2), the data scale that suppresses compounding error (§6.1, §6.3), doing the heavy lifting. The diffusion and flow-matching action heads of Chapter 10 and π0 (Chapter 13) are not departures from cloning; they are richer ways to parameterize the action distribution that BC fits, chosen to capture the multimodality a simple regression head smears out.

The third handoff is to Chapter 16, fine-tuning a VLA for your robot, where the methods §6.5 placed high on the ladder re-enter. They do not scale to multi-task pretraining, but for a *single* robot and a *single* task, collect a small teleop dataset, fine-tune a foundation model on it, optionally correct it with DAgger-style relabeling or sharpen it with RL, they are exactly the right tools. The decision procedure from §6.5 is the spine of that chapter.

What the chapter has not covered

Two omissions are worth naming. The chapter treated the *action representation* as a solved detail, predict a continuous action vector, minimize a regression loss, but glossed over the fact that a unimodal regression head averages across multiple valid expert behaviors and can produce an action that is the mean of two good choices and itself a bad one. This multimodality problem is real and it is why Chapter 10 exists; the chapter deferred it because the fix (diffusion and flow-matching heads) needs machinery not yet introduced. Read §6.2's regression loss as a placeholder that Chapter 10 upgrades.

The chapter also stayed almost entirely within single-step, single-policy imitation. It did not cover hierarchical imitation, a high-level policy that emits subgoals and a low-level policy that achieves them, which is a natural answer to the long-horizon compounding-error problem and which reappears concretely in Chapter 14's dual-system architectures (Helix, GR00T N1). The connection is real, but as with the hierarchical-RL omission in Chapter 5, the book defers it to the chapter where a concrete system makes the abstraction worth introducing.

Chapter 6's contribution to the book's overall argument is the imitation family from §1.4: the action models that learn from expert behavior rather than from extrinsic reward or self-supervision. The three central findings, that demonstrations are a richer and more scalable signal than reward for manipulation, that cloning's compounding-error failure is structural and is fought with data and DAgger rather than wished away, and that the choice among BC, IRL, and RL is a cost-ordered decision with BC as the near-universal starting point, are the findings that Part 4 will lean on when it explains why foundation action models look the way they do.

§6.x closes the chapter with a hands-on exercise, training a BC policy on a block-pushing dataset in MuJoCo, re-training it with DAgger, and quantifying the gap between them, and the full reading list for the chapter.

6.x Hands-on exercise + chapter references

Chapter 6 argued that cloning is where you start and that compounding error is what eventually bites you. The exercise is the one where you see both with your own eyes: train a behavior-cloning policy whose offline loss looks healthy, watch its rollout success rate disappoint, then close the gap with DAgger and put a number on the improvement. The four drills below take a combined two hours or so. A GPU shortens the training runs but nothing here requires one, and after the initial dependency install no internet is needed.

The test bed is a 2D block-pushing task: a point or planar end-effector must push a block to a goal region. The task is deliberately simple, you can watch every rollout, but it has the one property that makes the exercise worth doing: a horizon long enough for compounding error to show up. Install the dependencies:

```
pip install torch gymnasium gymnasium-robotics
```

If you prefer the manipulation-flavored version, `robomimic` ships the block-pushing and lift datasets used in Mandlekar et al. (2021, arXiv:2108.03298) and the workflow below maps onto it with minimal changes; the gymnasium `PointMaze`/`Push` environments are the lighter-weight option and are assumed in the instructions.

Exercise 6.x.1 — Collect (or load) a demonstration dataset

You need expert demonstrations before you can clone them. Two routes:

Route A: scripted expert. Write a short scripted controller that solves the push task using privileged state (the block pose and goal, which the *learner* will not see in raw form). A proportional controller that drives the end-effector to a pre-push pose behind the block and then toward the goal is enough. Roll it out for 200 episodes, recording at each step the *observation* the learner will receive (e.g., the rendered image or the low-dimensional observation vector, not the privileged state) and the *action* the scripted expert took. Save as `demos.npz` with arrays `obs` and `act`. The split between what the expert sees (full state) and what the learner sees (observation) is the whole point; it is what makes the cloned policy imperfect and therefore interesting.

Route B: load robomimic. Download the robomimic `lift` or `can` proficient-human dataset and use its 200 demonstrations directly. This skips the scripting and gives you genuine human data with the multimodality §6.6 warned about.

Record one number before moving on: the expert's own success rate over 50 fresh episodes. It should be at or near 100%. That is your ceiling.

Wall clock: about twenty minutes (Route A) or ten (Route B).

Exercise 6.x.2 — Train a behavior-cloning policy and measure the offline/online gap

Write `bc.py`. Build a small MLP (two or three hidden layers of 256 units is plenty for the low-dimensional observation; a tiny CNN if you are cloning from pixels) that maps observation to action. Train it as straight supervised regression on `demos.npz` with a mean-squared-error loss and Adam, the §6.2 loop:

```
for epoch in range(num_epochs):
    for obs, act in loader:
        pred = policy(obs)
        loss = F.mse_loss(pred, act)
        opt.zero_grad(); loss.backward(); opt.step()
```

Hold out 10% of the demonstrations as a validation set and plot training and validation loss. Both should fall to a small value and track each other; the policy fits the demonstrations and is not badly overfitting. This is the reassuring picture.

Now run the policy in the environment for 50 episodes and record the *rollout success rate* and the average episode length. Here is the lesson: the success rate will be meaningfully below the expert's 100%, often dramatically so on the longer episodes, even though the offline loss looked fine. Save the two numbers, offline validation loss and online success rate, side by side. The gap between "fits the data" and "succeeds on its own rollouts" is §6.3's compounding

error, made concrete. Write one sentence diagnosing where the rollouts fail; you will typically see the policy do well until a small error pushes the block or the end-effector into a configuration the demonstrations never showed, after which it flails, exactly the covariate-shift story.

Wall clock: about thirty minutes including training and evaluation.

Exercise 6.x.3 — Re-train with DAgger and quantify the gap closed

Write `dagger.py` starting from your BC policy. Implement the DAgger loop from §6.3:

```
dataset = load("demos.npz")          # start from the BC demonstrations
policy = train_bc(dataset)
for iteration in range(num_dagger_iters):
    new_obs, new_states = [], []
    for episode in range(rollouts_per_iter):
        # roll out the CURRENT policy, recording the states it visits
        obs, states = rollout(policy)
        new_obs += obs; new_states += states
    # the EXPERT relabels the learner's visited states
    new_act = [scripted_expert(s) for s in new_states]
    dataset = aggregate(dataset, new_obs, new_act)
    policy = train_bc(dataset)        # retrain on the aggregated set
```

The keystone is that the *expert* (your scripted controller from 6.x.1, which can see privileged state) labels the states the *learner* actually reached, including the off-distribution ones the pure-BC policy stumbled into. Run five DAgger iterations, 20 rollouts each. After each iteration, record the rollout success rate over 50 fresh episodes.

Plot success rate versus DAgger iteration, with the pure-BC result from 6.x.2 as iteration zero and the expert's 100% as a dashed ceiling line. Save as `dagger_curve.png`. The curve should climb from the disappointing BC number toward the expert ceiling over the five iterations. The gap it closes, BC success rate at iteration zero versus DAgger success rate at iteration five, is the single number this whole exercise exists to produce. Report it in one sentence: "DAgger raised rollout success from X% to Y% by relabeling N additional states with expert actions."

A worthwhile variant if you have time: track the *dataset size* alongside success rate, and note how many expert labels DAgger spent to buy the improvement. That label budget is the real cost §6.5 weighed when it placed DAgger one rung above BC; on hardware those labels are a human's time, not a free function call.

Wall clock: about forty-five minutes for five iterations and the plot.

Exercise 6.x.4 — Read an imitation-learning paper through the chapter's lens

Open one of: Mandlekar et al. (2021), "What Matters in Learning from Offline Human Demonstrations for Robot Manipulation" (arXiv:2108.03298), which is the empirical anchor for most of this chapter's claims; or RT-1 (arXiv:2212.06817), to see cloning at foundation scale. In a text file `imitation_audit.txt`, answer five questions, one paragraph each:

1. **Signal**. What is the training signal, pure BC, DAgger, IRL, offline RL, or a mixture? If BC, what action representation and loss? If the paper claims to fight compounding error, how?
2. **Data**. How many demonstrations, collected how (teleop, scripted, kinesthetic), at what frequency? Is the data single-task or multi-task, single-embodiment or cross-embodiment? Estimate the total labeled state-action count, the §6.1 quantity.
3. **Failure mode**. Where does the paper report the policy failing? Map the reported failures onto the chapter's vocabulary, covariate shift, multimodality smearing, insufficient coverage, or note that they do not fit and say what is missing.
4. **The counterfactual**. Would RL or IRL plausibly have done better on this task, per §6.5's ladder? What would each have demanded (reward, simulator, online interaction) that the authors apparently lacked or chose to

avoid?

5. **Judgment**. What single change, more data, DAgger relabeling, a richer action head, a reward signal, would most improve the reported result, and why?

There is no scoring rubric. The point is that the chapter's framing stops being abstract the moment you apply it to a paper you care about, and these five questions are the ones a design review of an imitation system actually asks.

Wall clock: about thirty minutes for the read plus the written audit.

Chapter 6 reading list

The works below are cited in §6.1–§6.6, grouped by purpose. Full bibliographic entries for everything cited in the book live in Appendix E.2; this is the chapter-local subset.

Behavior cloning: origins and modern practice

- Pomerleau, D. A. (1988). "ALVINN: An Autonomous Land Vehicle in a Neural Network." *NeurIPS 1988*. Behavior cloning before it had the name; §6.1's historical anchor. Three-layer net, 45 minutes of steering data, drove on real roads.
- Mandlekar, A., et al. (2021). "What Matters in Learning from Offline Human Demonstrations for Robot Manipulation." arXiv:2108.03298. The empirical study §6.1 and §6.5 lean on; the robomimic datasets and the finding that BC on a few hundred demos beats from-scratch RL under sparse reward. The companion codebase is the test bed for this chapter's exercises.
- Florence, P., et al. (2022). "Implicit Behavioral Cloning." *CoRL 2021*. The energy-based alternative to regression BC; relevant to §6.6's note that a unimodal regression head smears across multimodal expert behavior.

Compounding error and DAgger

- Ross, S., & Bagnell, J. A. (2010). "Efficient Reductions for Imitation Learning." *AISTATS 2010*. The analysis that makes the quadratic-in-horizon cost of compounding error precise; §6.3's central result.
- Ross, S., Gordon, G., & Bagnell, J. A. (2011). "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." *AISTATS 2011*. The DAgger paper; §6.3's fix and Exercise 6.x.3's algorithm.

Inverse RL and adversarial imitation

- Ng, A. Y., & Russell, S. (2000). "Algorithms for Inverse Reinforcement Learning." *ICML 2000*. The paper that named IRL and pointed out its ill-posedness; §6.4's starting point.
- Abbeel, P., & Ng, A. Y. (2004). "Apprenticeship Learning via Inverse Reinforcement Learning." *ICML 2004*. Feature-expectation matching; §6.4's first practical IRL.
- Ziebart, B. D., et al. (2008). "Maximum Entropy Inverse Reinforcement Learning." *AAAI 2008*. The MaxEnt formulation that became the field's workhorse; §6.4.
- Ho, J., & Ermon, S. (2016). "Generative Adversarial Imitation Learning." *NeurIPS 2016*. GAIL; the distribution-matching pivot in §6.4.
- Fu, J., Luo, K., & Levine, S. (2018). "Learning Robust Rewards with Adversarial Inverse Reinforcement Learning." *ICLR 2018*. AIRL; recovers a dynamics-disentangled, portable reward; §6.4's closing idea.

Offline RL: the rung between cloning and online RL

- Kumar, A., et al. (2020). "Conservative Q-Learning for Offline Reinforcement Learning." *NeurIPS 2020*. CQL; the out-of-distribution value-overestimation fix §6.5 names when it places offline RL on the ladder.
- Levine, S., et al. (2020). "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems." arXiv:2005.01643. The survey to read if §6.5's offline-RL paragraph leaves you wanting the full story.

Imitation at foundation scale: the bridge to Part 4

- Brohan, A., et al. (2022). "RT-1: Robotics Transformer for Real-World Control at Scale." arXiv:2212.06817. Cloning of 130k demonstrations over 700+ tasks; the §6.1 proof that imitation scales, revisited in Chapter 11.
- Collaboration, O. X.-E., et al. (2023). "Open X-Embodiment: Robotic Learning Datasets and RT-X Models." arXiv:2310.08864. The million-episode cross-embodiment dataset behind §6.1's scaling argument; detailed in Chapter 15.
- Black, K., et al. (2024). "π0: A Vision-Language-Action Flow Model for General Robot Control." arXiv:2410.24164. The BC-then-RL, flow-matching-head VLA §6.5 and §6.6 use to show where the chapter's ideas land in a current system; Chapter 13.

Chapter summary

Chapter 6 established imitation as the dominant training signal for modern robot learning and gave you the tools to use it well. You can now train a behavior-cloning policy and read its curves without being fooled by a healthy offline loss; explain compounding error with the horizon argument intact and apply DAgger to close the gap, having measured that gap yourself; distinguish cloning from inverse RL and offline RL by the demand each one makes; and decide, given a dataset and a robot, whether behavior cloning is the right starting point, which, as §6.5 argued and the exercises confirmed, it almost always is. Chapter 7 picks up the other half of that decision, developing the deep-RL machinery that fine-tunes a cloned policy past human performance when a reward and a simulator are at hand.

Chapter 7 — Deep RL for control: DQN to SAC and PPO

7.1 Function approximation: from Q-tables to Q-networks

The Q-learning of §5.3 stored one number for every state-action pair. That works on a 4×4 gridworld with 16 states and 4 actions: 64 numbers, all visited many times over a few thousand episodes. It does not work on anything a robot sees. The state of a manipulator with a camera is an image plus joint angles; the number of distinct states is effectively infinite, and the agent will never see the same one twice. A table indexed by state has no row to look up, no row to update, and nothing to generalize from. The whole apparatus of §5.3, the TD target, the off-policy property, the convergence theorem, assumed a representation that the real problem does not permit.

This section replaces the table with a parametric function. The good news is that almost everything from §5.3 survives the substitution syntactically: the update still pushes $Q(s, a)$ toward $r + \gamma \max_{a'} Q(s', a')$. The bad news is that the theorem does not survive, and the gap between "the update looks the same" and "the update still converges" is where deep RL lives. We will set up the function-approximation view, name the precise way it can blow up, and then walk through the three engineering fixes that turned it from a known-divergent idea into DQN, the network that learned to play Atari from pixels (Mnih et al. 2015).

A table is just a one-hot linear model

Start by noticing that the table was never special. Storing $Q(s, a)$ as an array is identical to a linear model $Q_\theta(s, a) = \theta^\top \phi(s, a)$ where the feature vector $\phi(s, a)$ is a one-hot indicator: a single 1 in the slot for (s, a), zeros everywhere else. The parameter vector θ is the table, flattened. The tabular update of §5.3 is exactly a gradient-descent step on the squared TD error under those one-hot features, with the step size α as the learning rate.

Seen this way, function approximation is one change: pick a ϕ that is *not* one-hot, so that updating θ for one state-action pair also changes the predicted value of nearby pairs. That sharing is the entire point. It is what lets the agent say something about a state it has never visited, by interpolating from states it has. A linear model with hand-built features ("distance to goal," "gripper-open flag") was the standard tool through the 1990s. A neural network replaces the hand-built ϕ with a learned one: $Q_\theta(s, a)$ is a multilayer network, θ its weights, and the features in the second-to-last layer are discovered by gradient descent rather than designed. For a robot taking pixels as input, the network is a convolutional encoder followed by a head that outputs one Q-value per discrete action.

The fitted-Q view: RL as a sequence of regressions

The cleanest way to think about training a Q-network is to treat it as a sequence of ordinary supervised regressions. This is *fitted Q iteration*, made neural by Riedmiller (2005) under the name NFQ. Suppose you have a batch of transitions (s, a, r, s'). Freeze the current network, and for each transition compute a regression target

$$y \;=\; r + \gamma \max_{a'} Q_{\theta^-}(s', a'),$$

where θ^- is the *frozen* copy of the weights. Now fit a fresh network Q_θ to the dataset $\{((s,a),\, y)\}$ by minimizing mean-squared error, exactly as in §3.3; this is supervised learning, nothing more. Then set $\theta^- \leftarrow \theta$ and repeat. Each round is a single application of the Bellman optimality operator from §5.2, projected onto the space of functions your network can represent.

The loss for one round is the regression objective we previewed at the end of §5.3:

$$\mathcal{L}(\theta) \;=\; \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\Big[\big(r + \gamma \max_{a'} Q_{\theta^-}(s',a') - Q_\theta(s,a)\big)^2\Big].$$

Two details in this expression carry all the weight, and both are easy to skip past. First, the target uses θ^-, not θ: we do not differentiate through the max term. The target is treated as a fixed label even though it depends on the network's own weights. Pretending it is fixed is what makes the step a regression rather than something stranger; it is also a lie, and the size of the lie is what we have to manage. Second, the expectation is over whatever distribution $\mathcal{D}$ the transitions were drawn from, and that distribution is not the one the greedy policy induces. We are about to see why that combination is dangerous.

The deadly triad

Tabular Q-learning converges. Fitted Q with a neural network can diverge, the Q-values run off to infinity and the policy becomes garbage, and it can do so on problems where a table would have been fine. Sutton & Barto (2018) call the culprit the *deadly triad*: the simultaneous presence of (1) function approximation, (2) bootstrapping, building the target from the network's own estimate rather than from a full Monte Carlo return, and (3) off-policy training, learning about the greedy policy from data collected by a different, exploratory one. Any two of the three are usually safe. All three together can be unstable.

The mechanism is a feedback loop. Because features are shared, raising $Q_\theta(s,a)$ to hit its target also raises Q_θ at the very states s' that appear inside other targets. Those targets move up, so the next regression chases them, which raises the values again. With a table the shared-feature step is absent and the loop cannot form; with on-policy Monte Carlo returns there is no bootstrap to amplify; but with all three, nothing pins the values down. Baird (1995) built a tiny seven-state MDP, *Baird's counterexample*, where linear TD with off-policy updates provably diverges, which is worth remembering precisely because it is linear: the instability is not a quirk of deep networks, it is structural. Tsitsiklis & Van Roy (1997) gave the matching theory, proving convergence for on-policy linear TD and exhibiting the off-policy divergence. None of this says deep Q-learning *will* diverge on your problem; it says nothing prevents it, so the algorithm needs explicit machinery to stay stable.

DQN: three fixes that made it work

DQN (Mnih et al. 2013; Mnih et al. 2015) is fitted Q-learning plus three modifications, each one a direct countermeasure to the instability above. It is worth knowing them individually, because the same three ideas reappear, lightly disguised, in every off-policy actor-critic method in §7.4.

A replay buffer. Rather than learn from each transition once, in the order it arrives, DQN stores transitions in a large buffer $\mathcal{D}$ (a million frames, in the Atari work) and samples minibatches uniformly from it, an idea due to Lin (1992). This does two things. It breaks the temporal correlation between consecutive samples, which a stochastic-gradient step assumes away, and it lets each expensive transition be reused many times, which matters enormously on a robot where collecting data means moving real motors. Replay is the single most important reason off-policy methods are sample-efficient, and §7.4 leans on it hard.

A target network. The moving-target problem, the label depends on the weights we are updating, is tamed by holding θ^- fixed for a stretch (in DQN, a few thousand gradient steps) and only then copying the live weights into it. The regression now has a stationary target for the duration of each stretch, which is the condition under which the fitted-Q view is honest. Later methods replace the periodic copy with a slow exponential average, $\theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$ with small τ, but the purpose is identical: slow the target down so the network is chasing something that is nearly still.

A regression loss with clipping. DQN minimizes the squared TD error by SGD, clipping the error term to $[-1, 1]$ (equivalently, using a Huber loss) so that a single wild target cannot produce a gradient large enough to wreck the weights. This is the same defensive move as gradient clipping in §3.5, applied at the level of the TD error.

Here is the inner loop, stripped to its essentials and small enough to read in one pass:

```
# s, a, r, s2, done: a minibatch sampled from the replay buffer
with torch.no_grad():
    q_next = q_target(s2).max(dim=1).values          # max_a' Q_theta-(s', a')
    y = r + gamma * q_next * (1.0 - done)            # TD target, frozen target net
q_pred = q_online(s).gather(1, a.unsqueeze(1)).squeeze(1)
loss = F.smooth_l1_loss(q_pred, y)                   # Huber: clipped regression
opt.zero_grad(); loss.backward(); opt.step()
# every C steps: q_target.load_state_dict(q_online.state_dict())
```

Run this on CartPole and the pole stays up within a few hundred episodes; run the convolutional version on Atari frames and you reproduce, with patience, the result that made the field pay attention in 2013. The behaviour policy is still ε-greedy from §5.3, now annealed from near-1 down to a small floor, and the off-policy property from §5.3 is exactly what licenses learning from a buffer full of transitions collected by stale, more-random versions of the policy.

The leftover bias, and what this still cannot do

One crack is worth naming because it motivates a later fix. The $\max_{a'}$ in the target both selects the best next action and estimates its value from the same noisy network, which biases the estimate upward: the network's overestimates get selected preferentially. van Hasselt, Guez & Silver (2016) remove most of this with *double Q-learning*: use the online network to pick the argmax action and the target network to score it. It is a two-line change with a measurable effect, and it is the kind of correction that distinguishes a Q-network that limps from one that works.

The deeper limitation is the one §5.3 already flagged and this section has not removed. A Q-network still acts by computing $\arg\max_a Q_\theta(s, a)$. With a handful of discrete actions, Atari's eighteen joystick positions, the argmax is a cheap forward pass over every action. With a robot's continuous action space, a seven-dimensional vector of joint velocities, the argmax is itself an optimization problem at every control step, and enumerating actions is hopeless. That single obstacle is why value-based methods, for all their sample efficiency, are not the end of the story for control, and why the next section turns to methods that represent the policy directly.

The takeaway from this section is the substitution and its price. Swapping the Q-table for a Q-network buys generalization across an enormous or continuous state space, at the cost of the convergence guarantee; the deadly triad means the update that was provably stable in §5.3 can now diverge. DQN's replay buffer, target network, and clipped regression loss are the standard countermeasures, and they recur throughout the chapter. What they do not fix is the argmax over actions, which §7.2 confronts by abandoning value-only learning and parameterizing the policy itself.

7.2 Policy gradients and the variance problem

Section 7.1 ended on an obstacle. A Q-network acts by computing $\arg\max_a Q_\theta(s, a)$, and for a robot whose action is a seven-dimensional vector of joint velocities, that argmax is an optimization problem solved afresh at every control step. Enumerating actions is hopeless when there are infinitely many of them. The fix in this section is to stop learning values and then reading off a policy, and instead parameterize the policy directly: a network $\pi_\theta(a \mid s)$ that takes a state and emits an action, or, more usefully, a distribution over actions, with no inner maximization anywhere. For a continuous action space the network outputs the parameters of a distribution, typically the mean and standard deviation of a Gaussian, $\pi_\theta(a \mid s) = \mathcal{N}(\mu_\theta(s), \sigma_\theta(s))$; sampling from it is one cheap forward pass, and acting requires no argmax at all.

The question this raises is the one the rest of the section answers. If the policy is just a network with weights θ, and we want to maximize expected return, can we get a gradient of the return with respect to θ and do ordinary gradient ascent? The answer is yes, the gradient has a clean closed form, and the catch, the reason policy gradients are delicate where supervised learning is not, is that the obvious unbiased estimator of that gradient has enormous variance. Most of the engineering in modern policy-gradient methods, including the PPO of §7.3, is variance reduction.

The objective and the log-derivative trick

Write a trajectory $\tau = (s_0, a_0, s_1, a_1, \dots)$ and its return $R(\tau) = \sum_t \gamma^t r_t$. The policy induces a distribution over trajectories, $p_\theta(\tau)$, and the objective is the expected return,

$$J(\theta) \;=\; \mathbb{E}_{\tau \sim p_\theta}\big[R(\tau)\big].$$

We want $\nabla_\theta J$. The difficulty is that θ sits inside the distribution we are averaging over, not inside the thing being averaged: moving θ changes *which trajectories appear*, not their returns, and you cannot differentiate a sum of fixed numbers with respect to a parameter that only reweights them, at least not directly. The way through is an identity sometimes called the log-derivative trick or the score-function estimator. For any distribution, $\nabla_\theta p_\theta(\tau) = p_\theta(\tau)\,\nabla_\theta \log p_\theta(\tau)$, simply because $\nabla \log p = (\nabla p)/p$. Substitute it into $\nabla_\theta J = \int \nabla_\theta p_\theta(\tau) R(\tau)\, d\tau$ and the integral folds back into an expectation:

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_{\tau \sim p_\theta}\big[\nabla_\theta \log p_\theta(\tau)\, R(\tau)\big].$$

This is already useful, because $\log p_\theta(\tau)$ decomposes. The trajectory probability is the product of the (unknown) environment dynamics and the (known) policy: $p_\theta(\tau) = p(s_0) \prod_t p(s_{t+1} \mid s_t, a_t)\, \pi_\theta(a_t \mid s_t)$. Taking the log turns the product into a sum, and every term that does not depend on θ, the initial-state distribution, every dynamics factor, has zero gradient and drops out. What survives is only the policy:

$$\nabla_\theta J(\theta) \;=\; \mathbb{E}_{\tau \sim p_\theta}\Big[\Big(\textstyle\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\Big)\, R(\tau)\Big].$$

This is the policy gradient theorem (Sutton et al. 2000), and the remarkable thing about it is the absence: nowhere does $\nabla_\theta p(s_{t+1} \mid s_t, a_t)$ appear. We never need to know, or differentiate, the dynamics of the robot or the simulator. The gradient is an expectation over trajectories the current policy generates, of a quantity we can compute entirely from the policy's own output. We estimate it by running the policy, collecting trajectories, and averaging, the same Monte Carlo idea from §3.2.

REINFORCE, and why it is a starting point and not an answer

The estimator that drops out of the theorem, with the return-to-go refinement that an action cannot influence rewards that already happened, is Williams' (1992) REINFORCE:

$$\hat{g} \;=\; \frac{1}{N} \sum_{i=1}^{N} \sum_t \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i)\, G_t^i, \qquad G_t = \sum_{k \ge t} \gamma^{k-t} r_k.$$

Read the term $\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, G_t$ as a weighted likelihood update and the intuition becomes plain. The gradient $\nabla_\theta \log \pi_\theta(a_t \mid s_t)$ points in the direction that makes the action actually taken more probable. Multiply it by the return that followed: if the outcome was good (G_t large and positive), take a step that makes that action more likely; if it was bad, step the other way. The policy is nudged toward actions that preceded high return and away from those that preceded low return. No value function, no argmax, no model, just "do more of what worked."

In a PyTorch loop this is three lines once trajectories are collected, and the structure is worth seeing because it makes the connection to ordinary supervised training concrete:

```
# logp: log pi_theta(a_t | s_t) for each step, with grad
# returns: G_t (return-to-go), detached, shape [T]
loss = -(logp * returns).mean()      # ascent on J => descent on -J
opt.zero_grad(); loss.backward(); opt.step()
```

It is a weighted negative-log-likelihood: the same loss as supervised classification, except the "labels" are the agent's own actions and each example is weighted by how well things went. That framing, policy gradient as supervised

imitation of your own lucky rollouts, is the thread that ties this chapter to the behavior cloning of Chapter 6 and, much later, to how VLAs are trained.

REINFORCE is unbiased: in expectation $\hat{g}$ equals the true gradient. But unbiased is not the same as usable. The estimator's variance is large enough that on most control tasks plain REINFORCE either learns glacially or not at all, and understanding why is the real content of this section.

The variance problem

Three things make $\hat{g}$ noisy, and they compound. First, G_t is a sum of many random rewards over a long horizon, so its own variance grows with episode length; a single rollout that happened to end well or badly swings the whole estimate. Second, the credit assignment is indiscriminate: the weight on $\nabla \log \pi_\theta(a_t \mid s_t)$ is the *entire* return-to-go G_t, which lumps together the consequences of action a_t with a great deal of reward that a_t had nothing to do with. The signal about any one action is buried in noise from all the others. Third, and this is the deep one, the scale of G_t is arbitrary in a way that corrupts the direction of the update.

The third point deserves a concrete example, because it is the cleanest motivation for everything that follows. Suppose every trajectory in a task earns a return between $+100$ and $+110$. The good actions and the bad actions are now both multiplied by a large positive number, so REINFORCE makes *all* of them more probable, merely by different amounts. The gradient still points the right way on average, but the useful signal is the ± 5 spread riding on top of a $+105$ pedestal, and the pedestal contributes pure variance. Shift every reward down by 105 and the task is unchanged, the optimal policy is identical, yet the estimator behaves completely differently. An estimator whose quality depends on an additive constant that does not change the problem is one we should not trust as written.

Baselines and the advantage

The repair follows directly from the example: subtract off the pedestal. For any function $b(s)$ that depends on the state but not the action, one can show that

$$\mathbb{E}_{a \sim \pi_\theta}[\nabla_\theta \log \pi_\theta(a \mid s)\, b(s)] \;=\; 0,$$

because $b(s)$ pulls out of the action-expectation and what remains, $\mathbb{E}_a[\nabla_\theta \log \pi_\theta(a \mid s)] = \nabla_\theta \sum_a \pi_\theta(a \mid s) = \nabla_\theta 1 = 0$. So we may subtract any such *baseline* from the return without changing the gradient in expectation, it stays unbiased, while changing the variance, often dramatically. The best simple choice of $b(s)$ is the state-value function $V(s)$, the expected return from s under the current policy. Subtracting it replaces G_t with $G_t - V(s_t)$, which estimates the *advantage*

$$A(s_t, a_t) \;=\; Q(s_t, a_t) - V(s_t),$$

the amount by which taking a_t beat the average action from s_t. Advantage is exactly the quantity the example was crying out for: positive when an action was better than the state's baseline expectation, negative when worse, and invariant to the $+105$ pedestal. Now "do more of what worked" means "do more of what worked *better than expected*," which is the signal we actually wanted.

Actor-critic: learn the baseline

We do not know $V(s)$, so we learn it, with exactly the value-regression machinery of §7.1. This is the *actor-critic* architecture (Konda & Tsitsiklis 2000): an *actor* π_θ that is updated by the policy gradient, and a *critic* V_ϕ that is trained by TD regression to predict returns and supplies the baseline the actor needs. The two learn together, the critic chasing the actor's current return distribution while the actor leans on the critic to tell good actions from merely good situations. Note the two halves descend from the two branches of this chapter: the critic is the value learning of §7.1, the actor is the policy gradient of this section. Every method in §7.4 is a variation on how to wire them together.

Using V_ϕ also lets us trade variance for a little bias. Instead of the full Monte Carlo G_t (unbiased, high variance) we can use a short-horizon, bootstrapped advantage such as the one-step TD residual $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$ (lower variance, biased because V_ϕ is imperfect). Generalized advantage estimation (GAE; Schulman et al. 2016, arXiv:1506.02438) makes this a dial: an exponentially weighted average of n-step residuals with a parameter $\lambda \in [0, 1]$ that interpolates from the high-bias one-step estimate at $\lambda = 0$ to the high-variance Monte Carlo estimate at $\lambda = 1$. In practice $\lambda \approx 0.95$ is a near-universal default, and GAE is the advantage estimator inside virtually every modern on-policy method, including the PPO of the next section. The asynchronous actor-critic A3C (Mnih et al. 2016, arXiv:1602.01783) added a second variance-reducing trick still in use: running many parallel actors so each gradient is averaged over decorrelated experience, the on-policy analogue of §7.1's replay buffer.

None of this changes the fundamental shape of the update. We still ascend $\mathbb{E}[\nabla_\theta \log \pi_\theta(a_t \mid s_t)\, \hat{A}_t]$; we have only replaced the raw, noisy G_t with a centered, lower-variance advantage estimate $\hat{A}_t$. That single substitution, return-to-go becomes advantage, is the difference between an estimator that is technically correct and one that trains a robot.

One sharp edge remains, and §7.3 exists to file it down. The policy gradient is an *on-policy* expectation: it is only valid for trajectories drawn from the current π_θ. The moment we take a gradient step, the data we collected is stale, so each batch of rollouts buys us exactly one update before it must be thrown away. Push that one step too hard and the policy can collapse; too softly and learning crawls. Managing the size of that step, so we extract as much improvement as possible from each precious batch of on-policy data without falling off a cliff, is precisely the problem PPO solves, and it is where we turn next.

7.3 PPO in 100 lines

Section 7.2 left us with a single sharp problem. The policy gradient is an on-policy expectation, valid only for trajectories drawn from the current π_θ. Collect a batch of rollouts, take one gradient step, and the data is stale; in principle every batch buys exactly one update before it must be discarded. That is ruinously sample-inefficient when each rollout costs a simulator episode, and the obvious fix, just take a bigger step, or several steps, on the same batch, is exactly what destabilizes training. Push the policy too far on one batch and it can land in a region where the new action distribution bears no resemblance to the one that generated the data, the advantage estimates become meaningless, and the run collapses into a flat or thrashing return curve that never recovers. Proximal Policy Optimization (PPO; Schulman et al. 2017, arXiv:1707.06347) is the method that lets you squeeze many updates out of each batch *without* falling off that cliff, and it does so with so little machinery that a complete, working implementation fits in about a hundred lines. It is, for that reason, the default first thing to reach for in on-policy deep RL, and the algorithm most likely to be running inside a robotics RL paper you pick up today.

The step-size problem, stated precisely

To control the step we need to measure it. The natural yardstick is not the change in the weights θ, two networks with very different weights can encode nearly the same policy, and vice versa, but the change in the *policy's output distribution*. PPO's predecessor, Trust Region Policy Optimization (TRPO; Schulman et al. 2015, ICML), made this exact, optimizing the policy improvement subject to a hard constraint that the KL divergence between the new and old action distributions stay below a small threshold. TRPO works well and comes with a monotonic improvement guarantee, but the constraint is enforced with a conjugate-gradient solve and a line search that are awkward to implement and do not slot cleanly into a standard autograd training loop. PPO's contribution is to get most of TRPO's stability from a plain first-order objective that any optimizer can descend, with no second derivatives and no constrained-optimization solver.

The object both methods center on is the *probability ratio* between the new and old policy for an action that was actually taken:

$$r_t(\theta) \;=\; \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}.$$

Here θ_{old} are the weights that collected the batch, held fixed; θ is the policy we are currently optimizing. At the start of a batch the two are identical and every $r_t = 1$. As we take update steps on the batch, r_t drifts away from 1: above 1 for actions the new policy now favors more than the old one, below for those it favors less. The ratio is precisely how far we have moved on the data we already collected, measured per action, and it is the quantity PPO watches.

The clipped surrogate objective

The importance-sampling form of the policy gradient says we can estimate the new policy's expected advantage using old data by weighting each sample's advantage $\hat{A}_t$ by the ratio: maximize $\mathbb{E}_t[r_t(\theta)\,\hat{A}_t]$. Left unconstrained this is exactly the trap; the objective rewards driving r_t arbitrarily large whenever $\hat{A}_t > 0$, and the advantage estimate is only trustworthy near $r_t = 1$. PPO's fix is almost crude in its simplicity. It clips the ratio so that, once it leaves a small interval $[1-\epsilon, 1+\epsilon]$, moving it further stops helping the objective:

$$L^{\text{CLIP}}(\theta) \;=\; \mathbb{E}_t\Big[\min(r_t(\theta)\,\hat{A}_t,\; \text{clip}(r_t(\theta),\,1-\epsilon,\,1+\epsilon)\,\hat{A}_t)\Big],$$

with ϵ typically 0.2. Read the two cases. When $\hat{A}_t > 0$, the action beat the baseline, the term grows as r_t rises, but the clip caps it at $(1+\epsilon)\hat{A}_t$; past that point the gradient is zero, so the optimizer has no incentive to push the action's probability any higher on this batch. When $\hat{A}_t < 0$, the same logic floors the ratio at $1-\epsilon$, removing the incentive to drive a bad action's probability to zero in one batch. The outer min is what makes the clip a genuine pessimistic bound rather than a cosmetic one: it ensures that when the ratio has moved in the *unhelpful* direction (for example $r_t < 1$ while $\hat{A}_t > 0$, meaning we accidentally made a good action less likely), the unclipped term is selected and the full corrective gradient flows. The effect is a flat, no-reward plateau in the objective for any action whose probability has already moved "enough," which removes the optimizer's reason to walk off the cliff while still letting it correct mistakes. No KL constraint is solved; the clip approximates the trust region with one line of arithmetic.

In practice many implementations also add an explicit KL term as a diagnostic and an early-stopping trigger: if the measured KL between old and new policy over the batch exceeds a target, stop updating on this batch. But the clip alone is what carries the method.

The full objective and the loop

PPO trains the actor and critic together, so the loss minimized is the clipped policy objective plus a value-function regression and an entropy bonus:

$$L(\theta) \;=\; -\,L^{\text{CLIP}}(\theta) \;+\; c_v\,\mathbb{E}_t\big[(V_\phi(s_t) - \hat{R}_t)^2\big] \;-\; c_e\,\mathbb{E}_t\big[\mathcal{H}[\pi_\theta(\cdot \mid s_t)]\big].$$

The first term is the policy objective (negated, since optimizers descend). The second is the critic loss from §7.2, a mean-squared error between the value head's prediction and the empirical return target $\hat{R}_t = \hat{A}_t + V_\phi(s_t)$, weighted by $c_v \approx 0.5$. The third is an entropy bonus, weighted by a small c_e (often 0.0 to 0.01): rewarding high entropy keeps the action distribution from collapsing to a near-deterministic policy too early, preserving the exploration that on-policy methods otherwise lose quickly. These three terms, clipped policy gain, value regression, entropy, are the whole of what PPO optimizes.

The surrounding loop is the part worth committing to memory, because its shape is the same in every implementation:

```
for iteration in range(num_iters):
    # 1. COLLECT: run current policy for T steps across N parallel envs
    obs, acts, logp_old, rews, dones, vals = rollout(policy, envs, T)

    # 2. ESTIMATE: GAE advantages + value targets, then normalize A
    adv, ret = gae(rews, vals, dones, gamma=0.99, lam=0.95)
    adv = (adv - adv.mean()) / (adv.std() + 1e-8)
```

```
    # 3. OPTIMIZE: K epochs of minibatch SGD on the SAME batch
    for epoch in range(K):                  # K ~ 4-10
        for mb in minibatches(batch, size=64):
            logp, entropy, value = policy.evaluate(mb.obs, mb.acts)
            ratio = (logp - mb.logp_old).exp()
            unclipped = ratio * mb.adv
            clipped   = ratio.clamp(0.8, 1.2) * mb.adv   # eps = 0.2
            policy_loss = -torch.min(unclipped, clipped).mean()
            value_loss  = 0.5 * (value - mb.ret).pow(2).mean()
            loss = policy_loss + 0.5 * value_loss - 0.01 * entropy.mean()
            opt.zero_grad(); loss.backward()
            nn.utils.clip_grad_norm_(policy.parameters(), 0.5)
            opt.step()
```

That is the algorithm. Step 1 collects a fixed budget of on-policy experience, typically by running N environments in parallel for T steps each (the A3C-style decorrelation trick from §7.2) and recording, critically, the log-probability $\log \pi_{\theta_{\text{old}}}(a_t \mid s_t)$ at collection time; this is the denominator of the ratio, frozen. Step 2 turns rewards and value predictions into GAE advantages (Schulman et al. 2016, arXiv:1506.02438) with the usual $\gamma = 0.99$, $\lambda = 0.95$, and normalizes the advantages to zero mean and unit variance across the batch, a small step that stabilizes the gradient scale enormously. Step 3 is where PPO earns its sample efficiency: it sweeps the *same* batch K times in shuffled minibatches, taking many gradient steps where REINFORCE would take one. The clip is the only thing that makes those repeated steps safe; on the first inner step every ratio is 1 and PPO is an ordinary advantage policy gradient, but by the last, the clip is actively zeroing the gradient for actions that have moved far enough.

Why the details are not optional

PPO has a reputation for being finicky, and the reputation is earned but misdirected: the core objective is robust, while a handful of "engineering" choices in the loop above are what actually separate a run that hits expert performance on a MuJoCo control task from one that plateaus. Advantage normalization, observation normalization (tracking a running mean and variance of the inputs and standardizing them), global gradient-norm clipping to 0.5, orthogonal weight initialization with a small gain on the policy's output layer, and learning-rate annealing are the usual suspects. None of them appear in the objective; all of them matter in practice, sometimes more than the clip parameter itself. Studies that reproduced PPO carefully found that much of its measured advantage over plain policy gradients came from this surrounding scaffolding rather than the clipped objective alone, a useful reminder that in deep RL the algorithm on the page and the algorithm that works are separated by a layer of unglamorous normalization.

A concrete target makes this real. On HalfCheetah-v4, a continuous MuJoCo control task with a 17-dimensional observation and a 6-dimensional torque action, a textbook PPO with the loop above, $N = 8$ to 16 parallel environments, $T \approx 2048$ steps per rollout, $K \approx 10$ epochs, and the normalization tricks in place will climb from a random-flailing return near zero to a competent running gait over a few million environment steps. Strip out advantage normalization or let the gradient norm run free and the same code will often stall. That gap, same objective, different scaffolding, is the single most important practical lesson of this section, and it is why the chapter's hands-on exercise asks you to build PPO on exactly this task.

PPO's strength is also its boundary. Because it is on-policy, it must throw away every batch after its K epochs and collect fresh experience; it cannot reuse the millions of transitions sitting in a replay buffer the way a value-based method can. On a fast simulator, where samples are nearly free and parallel environments are cheap, that trade is fine and PPO's stability wins. On a real robot, where every sample is a slow and possibly destructive interaction, throwing data away is a luxury you cannot afford, and that is precisely the pressure that motivates the off-policy actor-critic methods, which keep and reuse old experience, that we turn to next.

7.4 Off-policy actor-critic: DDPG, TD3, SAC

Section 7.3 ended on a complaint disguised as a fact. PPO is on-policy: after it sweeps a batch for its handful of epochs, the data is stale and must be thrown away. On a fast simulator that is fine, because samples are nearly free.

On a real robot it is close to fatal; every transition is a slow, wear-inducing, possibly destructive physical interaction, and collecting the millions of them PPO wants is not an option. What we need is a method that keeps every transition it has ever seen and squeezes many gradient updates out of each one. That is the promise of *off-policy* learning, and the family that delivers it for continuous control is the off-policy actor-critic: DDPG, its hardened successor TD3, and the maximum-entropy variant SAC that is now the default reach-for algorithm when sample efficiency matters.

The idea: marry Q-learning to a policy network

Recall the two obstacles. Q-learning (§5.3, §7.1) is beautifully off-policy; it learns from a replay buffer of transitions collected by any policy, because its Bellman target $r + \gamma \max_{a'} Q(s', a')$ depends only on the transition, not on who chose the action. But that $\max_{a'}$ is an optimization over the action space, and in a continuous space, a seven-joint arm with a six-dimensional torque vector, you cannot enumerate actions to maximize over them. Policy gradients (§7.2) sidestep the argmax by parameterizing the policy directly, but the vanilla version is on-policy and sample-hungry.

Deterministic Policy Gradient (DPG; Silver et al. 2014) and its deep incarnation DDPG (Lillicrap et al. 2016, arXiv:1509.02971) fuse the two. The trick is to replace the intractable $\max_{a'} Q(s', a')$ with a *learned* maximizer: train a deterministic actor network $\mu_\phi(s)$ whose job is to output the action that maximizes the critic. Then the Bellman target becomes

$$y \;=\; r \;+\; \gamma\, Q_\theta(s',\, \mu_\phi(s')),$$

with no maximization to solve; the actor *is* the approximate argmax. The critic Q_θ is trained by ordinary temporal-difference regression toward y (the same squared-error Bellman loss as DQN). The actor is trained to push its output in whatever direction makes the critic's estimate larger, which is just gradient ascent on the critic through the action input:

$$\nabla_\phi J \;=\; \mathbb{E}_{s\sim\mathcal{D}}\Big[\nabla_a Q_\theta(s,a)\big|_{a=\mu_\phi(s)}\, \nabla_\phi \mu_\phi(s)\Big].$$

Read that chain rule carefully, because it is the whole method. The gradient of the critic with respect to the action, $\nabla_a Q$, says "to raise the Q-value, nudge the action this way"; the actor Jacobian $\nabla_\phi \mu$ then translates that desired action-nudge into a weight update. Because both terms are computed on states s sampled from the replay buffer $\mathcal{D}$, states collected by *old* policies, the whole scheme is off-policy. DDPG inherits DQN's two stabilizers wholesale: a replay buffer that decorrelates samples and lets each transition be reused many times, and slowly-updated *target networks* $Q_{\theta'}$ and $\mu_{\phi'}$ that supply the bootstrap target y so the regression is not chasing a target that moves in lockstep with the weights being trained. DDPG's target networks use a soft update, $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$ with $\tau \approx 0.005$, rather than DQN's periodic hard copy.

Exploration needs a separate mechanism, because a deterministic actor left alone always emits the same action in a given state and never tries anything new. DDPG handles this by adding noise to the actor's output at collection time, $a = \mu_\phi(s) + \mathcal{N}$, and storing the noisy action in the buffer. The original paper used temporally correlated Ornstein-Uhlenbeck noise; later work found plain Gaussian noise works as well or better, which is one less moving part.

TD3: DDPG that does not lie to itself

DDPG works, when it works, but it earned a reputation for being fragile: wildly sensitive to hyperparameters, prone to a return curve that climbs promisingly and then collapses. The diagnosis, made precise by Fujimoto et al. (2018, arXiv:1802.09477), is *overestimation bias*. The actor is trained to maximize the critic, so it actively seeks out whatever actions the critic happens to overvalue; those errors then feed straight into the next Bellman target through $Q(s', \mu(s'))$, get bootstrapped forward, and compound. The critic, in effect, believes its own most optimistic mistakes, and the actor chases them off a cliff.

TD3, Twin Delayed DDPG, is DDPG plus three targeted fixes, and it is worth knowing them individually because each addresses a distinct failure and each shows up again in SAC.

The first is *clipped double-Q learning*. Train two independent critics $Q_{\theta_1}, Q_{\theta_2}$ and form the Bellman target using the *minimum* of the two:

$$y \;=\; r \;+\; \gamma \min_{i=1,2} Q_{\theta'_i}(s', \tilde{a}).$$

Taking the min is a deliberate pessimism. Two critics will not overestimate the same action in the same way, so the smaller of the two is a conservative estimate that systematically counteracts the upward bias. It can underestimate instead, but underestimation does not get chased and amplified the way overestimation does, so the trade is heavily favorable.

The second is *target policy smoothing*. The action $\tilde{a}$ in the target above is not $\mu_{\phi'}(s')$ exactly but that action plus a small clipped noise: $\tilde{a} = \mu_{\phi'}(s') + \mathrm{clip}(\epsilon, -c, c)$. The reasoning is that a good action should have a similar value to its near neighbors; without smoothing, a critic can develop a sharp, spurious spike at one action that the deterministic actor then exploits. Averaging the target over a little noise around the action regularizes those spikes away. This is a regularizer on the *value target*, not on exploration, a subtle but important distinction.

The third is *delayed policy updates*. Update the actor (and the target networks) less often than the critics, typically once every two critic updates. The point is to let the critic settle toward an accurate estimate before the actor takes a step that depends on it; chasing a noisy, half-trained critic is what produces the thrashing. The three fixes together turn DDPG from a method you coax into working into one that trains reliably on standard MuJoCo benchmarks, and TD3 remains a strong, simple baseline.

SAC: reward entropy, not just return

Soft Actor-Critic (Haarnoja et al. 2018, arXiv:1801.01290) keeps the off-policy, replay-buffer, twin-critic skeleton but changes the objective itself. Where every method so far maximizes expected return, SAC maximizes return *plus* the entropy of the policy:

$$J(\pi) \;=\; \sum_t \mathbb{E}\big[\, r_t \;+\; \alpha\, \mathcal{H}(\pi(\cdot \mid s_t))\big].$$

The temperature α sets the exchange rate between reward and randomness. The reframing has real consequences. The policy is now *stochastic*, it outputs a distribution, typically a squashed Gaussian, rather than a single action, and it is rewarded for staying as random as it can while still collecting return. This bakes exploration directly into the objective instead of bolting it on as injected noise the way DDPG and TD3 do: the agent explores because spreading probability mass is literally part of what it is maximizing. It also tends to learn more robust policies, because a high-entropy policy that still succeeds cannot be relying on one brittle sequence of actions.

Mechanically, SAC borrows TD3's clipped double-Q (it had a version of the idea concurrently) and adds the entropy bonus to the critic's target:

$$y \;=\; r \;+\; \gamma\Big(\min_i Q_{\theta'_i}(s', a') \;-\; \alpha \log \pi_\phi(a' \mid s')\Big), \qquad a' \sim \pi_\phi(\cdot \mid s').$$

The actor is trained with the reparameterization trick, sample $a = \tanh(\mu_\phi(s) + \sigma_\phi(s) \odot \xi)$ with $\xi \sim \mathcal{N}(0, I)$, so the gradient of the expected (Q minus log-probability) objective flows back through the sampled action into the network, exactly the pathwise gradient used in §3.2's discussion of reparameterization. The single most consequential practical refinement came shortly after: rather than tune α by hand, make it *automatic* by treating it as a dual variable and adjusting it to hold the policy's entropy near a target value (a reasonable default target is $-\dim(a)$, the negative of the action dimension). With automatic temperature, SAC has remarkably few knobs to turn, which is much of why it became the default.

The following sketch shows the off-policy update loop the three methods share; the bracketed comments mark where they diverge.

```
for step in range(total_steps):
    a = actor.act(s, explore=True)        # DDPG/TD3: mu(s)+noise; SAC: sample
    s2, r, done = env.step(a)
    buffer.add(s, a, r, s2, done)         # off-policy: keep everything
    s = env.reset() if done else s2

    batch = buffer.sample(256)            # reuse old transitions
    with torch.no_grad():
        a2 = target_actor(batch.s2)       # SAC: sample + entropy term
        a2 = a2 + clip_noise(a2)          # TD3 only: target smoothing
        q_tgt = torch.min(qt1(batch.s2, a2), qt2(batch.s2, a2))   # twin min
        y = batch.r + gamma * (1 - batch.done) * q_tgt
    update_critics(batch, y)              # TD-regression for Q1, Q2

    if step % policy_delay == 0:          # TD3: delayed; SAC/DDPG: every step
        update_actor(batch)               # ascend Q through the action
        soft_update(targets, tau=0.005)
```

How to choose, and what comes next

The pattern across the three is a steady accretion of pessimism and self-regularization on top of one core trick, a learned actor standing in for the continuous argmax. DDPG is the foundational idea but fragile in practice. TD3 is DDPG made trustworthy with three small, well-motivated fixes, and it is a fine choice when you want a deterministic policy and minimal conceptual overhead. SAC is the one to default to: its maximum-entropy objective gives principled exploration and robustness, its automatic temperature removes the most painful hyperparameter, and on continuous-control benchmarks it is reliably as sample-efficient as or better than TD3. All three crush PPO on sample efficiency, often by an order of magnitude in environment steps to reach a given performance, which is exactly the property you want when those steps are expensive.

That sample efficiency is why off-policy actor-critic, and SAC in particular, is the workhorse behind much of the learning-on-real-hardware literature, and why its replay-buffer logic reappears inside the offline-RL and Q-learning-flavored components of some later action-generation methods. It also sets up the central tension of robot RL: even SAC's "order of magnitude fewer samples" is still tens of thousands of real interactions, which is tens of thousands too many for most robots. The usual escape, train in a simulator where samples are cheap, then transfer to hardware, brings its own problem, the reality gap, which §7.5 takes up under the heading of domain randomization.

7.5 Sim-to-real: domain randomization in one slide

Section 7.4 left us with an uncomfortable arithmetic. Soft actor-critic is roughly an order of magnitude more sample-efficient than PPO, and yet "an order of magnitude better" still lands at tens of thousands of environment transitions to learn a single manipulation skill. On a fast MuJoCo simulator that is a coffee break. On a real seven-axis arm it is days of supervised operation, joints wearing under load, and the non-negotiable possibility that a half-trained exploratory policy drives the gripper into the table. The escape hatch every robot-learning group reaches for is the same: train in a simulator where transitions are nearly free and failures cost nothing, then ship the resulting policy to hardware. The catch, and it is the whole subject of this section, is that the simulator is wrong. Its contact model is an approximation, its friction coefficients are guesses, its camera renders clean textures the real camera never sees, and its actuators respond instantly where real motors lag. A policy that overfits to those quirks works beautifully in sim and falls over on the robot. This mismatch has a name, the *reality gap*, and domain randomization is the bluntest, most effective tool we have for crossing it.

The reality gap is a distribution-shift problem

It helps to see the reality gap through the lens of §3.4 rather than as something exotic to robotics. A policy trained in simulation is a function fit on a training distribution, the distribution of states, images, and dynamics the simulator

produces. Deployed on hardware, it is evaluated on a different distribution. This is ordinary train/test distribution shift, the same failure mode that breaks any supervised model asked to generalize beyond its training set. The only thing special about the robot case is that we have unusual control over the training distribution: we built the simulator, so we get to choose what it samples.

That observation is the entire idea behind domain randomization. If you cannot make the simulator match reality precisely, and you cannot, because you do not know the real friction coefficient to three decimal places, then do not try. Instead, randomize the uncertain parameters across a wide range during training, so that the real world looks to the policy like just one more sample from a distribution it has already seen. The policy is never allowed to depend on the exact value of any quantity it cannot trust, because that quantity keeps changing under its feet. Put as a slogan that fits on the one slide the section title promises: **reality becomes just another randomized instance**.

Visual randomization: the original result

The technique entered the field through vision. Tobin et al. (2017, IROS) were trying to train an object detector in simulation that would work on a real robot, and rather than invest in photorealistic rendering they did the opposite. They rendered the same scene thousands of times with the textures, colors, lighting positions, camera pose, and distractor-object placement all randomized to deliberately unrealistic extremes: checkerboard tables, neon clutter, light sources scattered at implausible angles. The bet was that a network forced to locate an object across that chaotic range of appearances would learn features invariant to texture and lighting altogether, and would therefore treat a real, photographically mundane scene as merely one more variation. The bet paid off: the detector transferred to the real world with no real images in training at all, localizing objects to within about a centimeter. The lesson generalized fast. If the policy's input is pixels, randomize everything about how those pixels are generated that does not carry task information, and the network is pushed to ignore the rendering and attend to the geometry.

Dynamics randomization: the harder half

Vision is the easy half, because appearance is obviously a nuisance variable. The harder and more consequential form is *dynamics* randomization, where the quantities you perturb are the physics parameters that actually determine how the robot moves: link masses, joint friction and damping, motor gains, actuator latency, the coefficient of restitution at contacts, and the time delay between issuing a command and the body responding. Peng et al. (2018, ICRA) made the case directly for control, randomizing dynamics parameters while training a manipulation policy and showing transfer to a real arm. The mechanism is subtler than the visual case. A policy trained under a single fixed set of dynamics learns an open-loop-ish strategy tuned to exactly those numbers; change the friction by ten percent and the carefully calibrated motion overshoots. A policy trained across a wide band of dynamics cannot rely on any one setting, so it is forced to learn something more like feedback control: sense the current state, react to what the body is actually doing, correct continuously. Robustness to the range, in other words, tends to produce closed-loop behavior as a side effect, and closed-loop behavior is exactly what survives contact with an unmodeled real robot.

There is a recurring design subtlety worth naming. Randomizing dynamics makes the environment *partially observable*: from a single frame the policy cannot tell whether it is in a high-friction or low-friction draw of the world. Two responses are common. The first is to give the policy a short history of recent states and actions, a stack of frames or a small recurrent memory, so it can infer the current dynamics from how the robot has been responding, a trick sometimes called *implicit system identification*. The second is to accept that the policy will hedge, producing a single robust behavior that is acceptable across the whole range rather than optimal for any one setting. Both appear constantly in the literature, and the choice between an explicitly conservative policy and an adaptive, history-conditioned one is a genuine design axis, not a solved question.

Two landmark demonstrations

Two results are worth carrying in your head as concrete anchors, because they bracket what the method can do.

The first is OpenAI's in-hand manipulation work that solved a Rubik's Cube one-handed with a Shadow robotic hand (Akkaya et al. 2019, arXiv:1910.07113). The control policy was trained entirely in simulation under aggressive randomization of physics, sensor noise, and even the robot's own geometry. Their refinement, *automatic domain randomization* (ADR), is the part worth remembering: instead of fixing the randomization ranges by hand, they

widened each range automatically as the policy got good enough to handle the current one, growing the training distribution in lockstep with competence. The hand learned to adapt to perturbations it had never explicitly seen, fingers tied together, a plush giraffe shoved against the cube, because the curriculum had taught it to expect that the world's parameters were never to be trusted.

The second is quadrupedal locomotion. Lee et al. (2020, Science Robotics) trained an ANYmal robot to walk and recover over rough, slippery, and deformable terrain entirely in simulation with randomized ground properties and disturbances, then deployed it outdoors over mud, snow, and rubble with no real-world fine-tuning. Legged locomotion is the domain where sim-to-real has arguably matured most completely: the dynamics are fast and contact-rich, real-world trial-and-error is genuinely dangerous, and yet randomized-sim training followed by zero-shot deployment is now close to a standard recipe rather than a stunt.

Where it breaks, and the alternatives that bound it

Domain randomization is not free, and the section would be dishonest without its failure modes. Widening the randomization range trades peak performance for robustness: a policy that must succeed across friction coefficients from 0.3 to 1.5 will be beaten, on any single friction value, by a policy specialized to it. Push the ranges too wide and the task can become unlearnable, because no single behavior succeeds across the whole span and training stalls or collapses. The ranges themselves are hand-chosen hyperparameters, and choosing them badly, too narrow to cover reality, or centered on the wrong nominal values, reintroduces the gap you were trying to close. ADR addresses the width problem but not the centering problem; the simulator's structural errors, the physics it gets categorically wrong rather than merely imprecisely, are not fixed by sampling a parameter more widely.

This is why domain randomization usually travels with companions. *System identification* measures the real robot's parameters and narrows the randomization band around the truth, getting the best of both worlds. *Domain adaptation* methods adjust the policy or its input features using a modest amount of real data, meeting the gap from the other side. And the rapid-adaptation line of work, exemplified by RMA on legged robots, trains an explicit module that infers the environment parameters online from the recent state history and feeds them to the policy, turning the partial-observability problem from a nuisance into a designed-for input. The honest framing is that domain randomization is the cheap, strong baseline that gets a policy most of the way across the gap, and the other techniques are what you add when "most of the way" is not enough.

For the purposes of this book, the takeaway is conceptual rather than recipe-level. Domain randomization reframes the sim-to-real problem as a distribution-design problem, and it works because robustness to variation is a usable substitute for accuracy you do not have. That same instinct, train across enormous variation so that the test case is just another sample, reappears, scaled up by orders of magnitude, when we get to the data side of foundation action models: the multi-robot, multi-environment datasets of Chapter 12 and Chapter 15 are, in a sense, domain randomization performed with real robots instead of a simulator. Section 7.6 closes the chapter by tying the deep-RL toolkit together and stating plainly which parts of it survive into the VLA era and which do not.

7.6 Summary

Chapter 7 was the deep-RL chapter, and it carried two jobs at once. The first was to assemble the modern reinforcement-learning toolkit, the algorithms that turn the tabular MDP machinery of Chapter 5 into something that runs on a high-dimensional robot, in the order the field actually built them: value-based methods, then policy gradients, then the actor-critic synthesis that dominates control today. The second, quieter job was to keep asking which parts of that toolkit survive into the foundation-model era of Part 4 and which were superseded by the imitation recipe of Chapter 6. This summary collects the load-bearing ideas and marks where the later chapters keep reaching back for them.

The four ideas worth carrying forward

Function approximation breaks the comfortable guarantees of tabular RL, and DQN is the engineering that makes it work anyway. §7.1 traced the move from a Q-table with one cell per state to a Q-network that generalizes across states it has never seen, the only option once the state is an image or a continuous joint configuration. The catch is that

the convergence proofs of Chapter 5 quietly assumed a table; combine bootstrapping, off-policy data, and function approximation and you have the "deadly triad" that can diverge outright (Sutton & Barto 2018). DQN (Mnih et al. 2015) is not a new theory so much as a set of stabilizers: a replay buffer to break the temporal correlation in the data, and a slowly-updated target network to stop the regression target from chasing its own tail. Recognizing those two tricks as answers to a stability problem, rather than arbitrary implementation details, is what lets you read every later off-policy method as a variation on the same theme.

Policy gradients optimize the thing you actually want, and pay for it in variance. §7.2 made the case for a different family entirely: instead of learning a value function and acting greedily, parameterize the policy directly and push its parameters up the gradient of expected return. This is the natural fit for continuous action spaces, where the `max` over actions that Q-learning needs becomes intractable, and it is the lineage that every VLA action head ultimately belongs to. The price is variance. The REINFORCE estimator (Williams 1992) is unbiased but noisy, and the whole craft of §7.2, baselines, advantage estimation, GAE (arXiv:1506.02438), is machinery for cutting that variance without introducing bias you cannot control. The bias-variance dial introduced there is the same dial you turn when you tune any modern policy-gradient method.

PPO is the workhorse because it is hard to break. §7.3 built PPO (arXiv:1707.06347) as the answer to a practical question: how do you take the largest policy-improvement step the data supports without stepping so far that the policy collapses? The clipped surrogate objective is a cheap stand-in for the trust region of TRPO; it simply refuses to reward the update for moving the action probabilities too far from where they started. The reason PPO, and not something more sophisticated, is the default for sim-trained locomotion and for RLHF alike is robustness: it tolerates sloppy hyperparameters, parallelizes cleanly across simulated environments, and rarely diverges. When §6.5 and §7.5 spoke of "fine-tuning a cloned policy with RL," PPO is almost always the verb.

Off-policy actor-critic buys sample efficiency, which is the currency that matters on hardware. §7.4 closed the algorithmic arc with DDPG, TD3, and SAC (arXiv:1801.01290), the methods that combine a learned critic with a directly-parameterized actor and reuse a replay buffer the way DQN does. The throughline from TD3's twin critics to SAC's entropy bonus is a sequence of fixes for the same failure, an over-optimistic critic that the actor learns to exploit, and SAC's maximum-entropy objective turns exploration from a bolted-on heuristic into part of the loss. The reason the chapter cared is the arithmetic of §7.5: PPO's on-policy data hunger is fine in a free simulator and ruinous on a real arm, so when learning must touch hardware, the order-of-magnitude sample-efficiency gain of an off-policy method is the difference between feasible and not.

What you should be able to do now

Four concrete capabilities, in roughly the order the rest of the book will ask for them.

You should be able to *choose between a value-based, on-policy, and off-policy method for a given problem, and justify the choice.* The decision turns on a few questions the chapter equipped you to ask. Is the action space discrete or continuous? DQN-family methods need the former, policy-gradient and actor-critic methods handle the latter. Is data cheap (a fast simulator) or expensive (a real robot)? The first tolerates PPO's on-policy appetite, the second pushes you toward SAC's replay-buffer efficiency. Do you need a robust default you can get running today, or are you willing to tune for peak performance? Being able to walk that decision tree out loud is the chapter's central practical skill.

You should be able to *explain why deep RL is unstable and name the specific mechanisms that stabilize it.* Not "neural nets make RL hard" but the deadly triad by name, and the concrete countermeasures: replay buffers and target networks (§7.1), trust regions and clipping (§7.3), twin critics and target-policy smoothing and entropy regularization (§7.4). Each is a response to a named pathology: correlated data, a moving target, an exploitable critic, premature exploitation. Reading a new RL paper's "tricks" section as a catalogue of answers to these pathologies, rather than as folklore, is what this chapter was meant to give you.

You should be able to *read the variance-reduction machinery of a policy-gradient method and say what each piece buys.* A baseline reduces variance without adding bias; an advantage function recenters the gradient on whether an action beat expectations; GAE's λ trades a little bias for a large variance reduction; PPO's clip keeps the update honest. Naming the job of each component, rather than treating the loss function as an indivisible incantation, is what lets

you debug a policy-gradient method that will not learn, and it connects directly back to the bias-variance framing of §3.4.

You should be able to *frame sim-to-real as a distribution-design problem and apply domain randomization accordingly*. §7.5's reframing is the durable idea: the reality gap is ordinary train/test distribution shift, and because you build the simulator you control the training distribution. Randomize the parameters you cannot trust, textures and lighting for vision (Tobin et al. 2017), masses and friction and latency for dynamics, widely enough that the real robot looks like one more sample. You should also be able to state the cost: robustness trades away peak performance, the ranges are hand-chosen, and structural simulator errors are not fixed by wider sampling.

Where the chapter has set up the rest of the book

Chapter 7 hands off in three directions. The most immediate is the diffusion and flow chapters of Part 3. The policy-gradient lineage of §7.2 ends, several chapters later, at the action heads of modern VLAs: a VLA that outputs a continuous action distribution is parameterizing a policy, and the question of how to represent and train that distribution is the question §7.2 opened. Chapter 10's diffusion and flow-matching heads, and π0's flow-matching objective in Chapter 13 (arXiv:2410.24164), are richer answers to it than the Gaussian policy SAC assumes.

The deeper handoff is to the data argument that runs through Part 4. §7.5 ended by noting that the multi-robot, multi-environment datasets of Chapters 12 and 15 are, in effect, domain randomization performed with real robots instead of a simulator: train across enormous variation so that the deployment case is just another sample. That instinct, that robustness comes from the breadth of the training distribution, not from matching any single condition exactly, is the same instinct that powers foundation action models, scaled up by orders of magnitude. The reader who internalized §7.5 will recognize Open X-Embodiment (arXiv:2310.08864) as the same idea wearing different clothes.

The third handoff is to fine-tuning, in Chapter 16. The BC-then-RL pattern that §6.5 identified as where the field converged is built from this chapter's parts: clone a policy from demonstrations, then sharpen it past human performance with PPO when a simulator and reward exist, or with an off-policy method when sample budget is tight. Chapter 16's recipe for adapting a foundation model to a new robot leans on knowing which RL tool fits which budget, exactly the decision tree above.

What the chapter has not covered

Two omissions are worth naming. The chapter stayed almost entirely within *model-free* RL, methods that learn a value function or a policy directly from transitions, never building an explicit model of the environment's dynamics. That is a deliberate deferral, not an oversight: model-based RL, where the agent learns a predictive model and plans or trains inside it, is the entire subject of Chapter 9's world models, and the sample-efficiency story that §7.4 told through off-policy reuse has a second, larger chapter that only makes sense once world models are on the table. Read this chapter's sample-efficiency discussion as the model-free half of a two-part argument.

The chapter also treated exploration lightly. Beyond ε-greedy in §7.1 and SAC's entropy bonus in §7.4, it did not cover the harder exploration problem, sparse rewards, intrinsic motivation, curiosity, count-based bonuses, that dominates RL research on tasks where reward is rare. For robot manipulation the omission is defensible, because §6.1's argument holds: when demonstrations are available, imitation sidesteps exploration entirely, and that is precisely why the field's center of gravity moved from RL to imitation for manipulation. The hard-exploration literature matters most exactly where demonstrations are not available, which is not where the rest of this book lives.

Chapter 7's contribution to the book's overall argument is to complete the "learning from rewards" thread that Chapter 5 opened, and then to mark its boundary. The deep-RL toolkit is real, powerful, and load-bearing in locomotion and in fine-tuning, but it is not the engine of the foundation action models the book builds toward; that engine is the imitation of Chapter 6, scaled with data. The three findings to carry forward are that deep RL is fundamentally a stability-engineering problem dressed as a learning one, that the on-policy/off-policy split is really a data-budget decision, and that sim-to-real is distribution design rather than physics fidelity. Part 3 picks up the sequence-model and generative machinery that, combined with this RL lineage and Chapter 6's imitation lineage, finally produces a VLA.

§7.x closes the chapter with a hands-on exercise, training PPO and SAC on the same continuous-control task and comparing their sample efficiency and wall-clock cost, and the full reading list for the chapter.

7.x Hands-on exercise + chapter references

Chapter 7 made two claims that are easy to nod along to and hard to feel until you have run the code. The first is that on-policy methods like PPO are robust but sample-hungry, while off-policy methods like SAC squeeze far more learning out of the same number of environment steps, at the cost of more moving parts that can quietly break. The second is that a policy trained in one fixed simulator overfits its dynamics, and that randomizing those dynamics during training is what buys you transfer. The four drills below make both claims concrete. They take two to three hours combined. Classic control tasks run on a laptop CPU; the SAC drill is faster on a GPU but does not require one, and after the dependency install nothing here needs internet.

The test beds are the two standard classic-control environments: `CartPole-v1` (discrete actions, the natural home for a DQN- or PPO-style policy) and `Pendulum-v1` (one continuous torque action, the natural home for SAC). They are deliberately tiny, you can watch every rollout and a full training run finishes in minutes, but they are enough to expose the sample-efficiency gap and the sim-to-real failure mode without burning a day of compute. Install the dependencies:

```
pip install torch gymnasium
```

If you would rather not reimplement the algorithms from scratch, `stable-baselines3` ships vetted PPO and SAC implementations and every drill below maps onto it with a few lines; reading a known-correct implementation alongside §7.3 and §7.4 is itself a worthwhile exercise. The instructions assume you are writing your own loops, because the bugs you hit are the lesson.

Exercise 7.x.1 — PPO on CartPole, and the meaning of a learning curve

Write `ppo_cartpole.py` using the §7.3 skeleton: an actor and a critic (two small MLPs, or one shared trunk with two heads, 64 units per layer is plenty), GAE for the advantage, and the clipped surrogate objective. Collect rollouts in batches of around 2,048 environment steps, then run a handful of epochs of minibatch updates over each batch before discarding it, the on-policy discipline §7.3 insisted on.

Train until the policy reliably balances the pole for the full 500 steps. Plot two curves against environment steps: the average episode return, and the approximate KL divergence between the policy before and after each update. Record one number: the total environment steps to reach a 475-return moving average. Then read the KL curve. If you ever see return collapse from near-500 back toward 20 in a single update, look at the KL spike that preceded it; that is the trust-region violation §7.3 described, the policy stepping so far that the data it just collected no longer describes where it landed. Lowering the clip range or the learning rate should make the collapse disappear. Keeping the step-to-threshold number is the point; it is the baseline the next drill undercuts.

Wall clock: about twenty minutes including the runs.

Exercise 7.x.2 — SAC on Pendulum, and the sample-efficiency gap

Now switch to the continuous task. Write `sac_pendulum.py` using the §7.4 ingredients: a stochastic Gaussian actor with the squashing correction, twin Q-critics with the min-of-two target to fight overestimation, target networks updated by Polyak averaging, a replay buffer, and the automatic temperature adjustment that tunes the entropy bonus. This is more code than PPO and more ways to get it wrong; if the policy's return is stuck near the floor, the usual culprits are a sign error in the entropy term, forgetting the `tanh` log-prob correction, or updating the target network too fast.

Train SAC on `Pendulum-v1` and, for an honest comparison, also point your PPO implementation at the same environment (PPO handles continuous actions with a Gaussian head, §7.2). Plot both return curves on the same axes against environment steps. The expected picture, and §7.4's headline claim, is that SAC reaches good performance in far fewer environment steps than PPO, because it reuses every transition many times out of the replay buffer rather than throwing each batch away. Record both step-to-threshold numbers and the ratio between them. That ratio is

the sample-efficiency gap, and it is the reason §7.4 argued you reach for an off-policy method the moment real-robot samples get expensive, and the reason you tolerate PPO's appetite when, as in §7.5, the samples come from a fast simulator instead.

Wall clock: about forty minutes including both runs.

Exercise 7.x.3 — Domain randomization and the transfer test

This drill makes §7.5 concrete with the smallest possible "sim-to-real" gap: a mismatch between a training simulator and a different test simulator. Take the Pendulum environment and treat its physical constants, pole mass, pole length, maximum torque, gravity, as the dynamics parameters a real pendulum would differ on.

Train two SAC policies. The first trains on the default parameters only, the fixed-simulator baseline. The second resamples the parameters at the start of every episode from a band around the defaults (say ±30% on mass and length), which is domain randomization in its plainest form. Then build a held-out test set: a grid of pendulum parameter settings the policy never trained on, including a few outside the randomization band, and evaluate both policies on all of them.

The fixed-simulator policy will do well on the exact default it trained on and degrade, sometimes sharply, as the test pendulum drifts away from it. The randomized policy will be slightly worse on the nominal setting and markedly better across the rest of the grid. Tabulate the two policies' average return per test setting side by side. The gap you measure is exactly the §7.5 thesis: randomization trades a little nominal performance for a policy that survives the reality gap, by forcing a single policy to be robust across the whole band rather than tuned to one set of constants. Note also where the randomized policy itself falls off, at the grid points well outside its training band, because that boundary is the honest limit of the method: randomization buys robustness inside the distribution you randomized over, not magic generalization beyond it.

Wall clock: about forty-five minutes including training and the sweep.

Exercise 7.x.4 — Read a deep-RL paper and audit it against the chapter

Pick one algorithm paper from the reading list, DQN (arXiv:1312.5602 / Nature 2015), PPO (arXiv:1707.06347), or SAC (arXiv:1801.01290) are the cleanest choices, and read it with the chapter's vocabulary in hand. Write a short audit answering five questions:

1. **What problem in the previous method is this paper fixing?** DQN's replay buffer and target network fix the divergence §7.1 warned about; PPO's clip fixes TRPO's expensive trust region from §7.3; SAC's twin critics fix the overestimation §7.4 traced to DDPG.
2. **Which of the chapter's failure modes does it confront directly, and which does it leave to the reader?** Overestimation, on-policy staleness, the exploration-exploitation balance, brittle hyperparameters.
3. **What does the paper report its sample budget as,** and how does that compare to the step counts you measured in Exercises 7.x.1–7.x.2?
4. **Where would this method break on a real robot**, using §7.5's reality-gap framing? What does its benchmark quietly assume that a physical robot will not give you?
5. **If you had to fine-tune a cloned policy (Chapter 6) past human performance with this method, what would the hard part be?** This is the question Chapters 11–16 spend their pages answering.

There is no scoring rubric. The point is that an algorithm paper stops being a wall of equations the moment you can name which earlier failure each design choice is paying down, and these five questions are the ones a researcher actually asks when deciding whether to build on a method.

Wall clock: about thirty minutes for the read plus the written audit.

Chapter 7 reading list

The works below are cited in §7.1–§7.6, grouped by purpose. Full bibliographic entries for everything cited in the book live in Appendix E.2; this is the chapter-local subset. The classical RL references predate the arXiv mirror in `sources/` and are cited by author, year, and venue.

Value-based deep RL and function approximation

- Tsitsiklis, J. N., & Van Roy, B. (1997). "An Analysis of Temporal-Difference Learning with Function Approximation." *IEEE TAC* 42(5). The result behind §7.1's warning that off-policy bootstrapping with function approximation can diverge, the "deadly triad."
- Riedmiller, M. (2005). "Neural Fitted Q Iteration." *ECML 2005*. Batch Q-learning with a neural network; §7.1's immediate ancestor to DQN.
- Mnih, V., et al. (2013, 2015). "Playing Atari with Deep Reinforcement Learning" (arXiv:1312.5602) and "Human-level Control through Deep Reinforcement Learning" (*Nature* 518). DQN; the replay buffer and target network that made §7.1's deep value learning stable. Exercise 7.x.4's cleanest target.
- van Hasselt, H., Guez, A., & Silver, D. (2016). "Deep Reinforcement Learning with Double Q-Learning." *AAAI 2016*. The overestimation fix §7.1 and §7.4 both lean on.

Policy gradients and on-policy methods

- Williams, R. J. (1992). "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8. REINFORCE; the log-derivative estimator at the root of §7.2.
- Sutton, R. S., et al. (2000). "Policy Gradient Methods for Reinforcement Learning with Function Approximation." *NeurIPS 2000*. The policy gradient theorem §7.2 derives.
- Schulman, J., et al. (2016). "High-Dimensional Continuous Control Using Generalized Advantage Estimation." *ICLR 2016*. arXiv:1506.02438. GAE, the variance-reduction tool §7.2 and §7.3 both use.
- Schulman, J., et al. (2015). "Trust Region Policy Optimization." *ICML 2015*. TRPO; the constrained-step idea §7.3 says PPO approximates cheaply.
- Schulman, J., et al. (2017). "Proximal Policy Optimization Algorithms." arXiv:1707.06347. PPO; §7.3's main result and Exercise 7.x.1's algorithm.

Off-policy actor-critic

- Silver, D., et al. (2014). "Deterministic Policy Gradient Algorithms." *ICML 2014*. DPG; the deterministic-actor foundation §7.4 builds on.
- Lillicrap, T. P., et al. (2016). "Continuous Control with Deep Reinforcement Learning." *ICLR 2016*. arXiv:1509.02971. DDPG; §7.4's first deep off-policy actor-critic for continuous control.
- Fujimoto, S., van Hoof, H., & Meger, D. (2018). "Addressing Function Approximation Error in Actor-Critic Methods." *ICML 2018*. arXiv:1802.09477. TD3; the twin critics and delayed updates §7.4 and Exercise 7.x.2 use.
- Haarnoja, T., et al. (2018). "Soft Actor-Critic." *ICML 2018*. arXiv:1801.01290. SAC; §7.4's maximum-entropy off-policy method and Exercise 7.x.2's algorithm.

Sim-to-real and domain randomization

- Tobin, J., et al. (2017). "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World." *IROS 2017*. The randomization idea §7.5 and Exercise 7.x.3 are built on.
- Peng, X. B., et al. (2018). "Sim-to-Real Transfer of Robotic Control with Dynamics Randomization." *ICRA 2018*. Randomizing *dynamics* rather than only appearance, exactly Exercise 7.x.3's setup.
- OpenAI; Akkaya, I., et al. (2019). "Solving Rubik's Cube with a Robot Hand." arXiv:1910.07113. Domain randomization plus automatic curriculum at scale; §7.5's headline demonstration.
- Lee, J., et al. (2020). "Learning Quadrupedal Locomotion over Challenging Terrain." *Science Robotics* 5(47). Randomized-simulation training that transferred to a real legged robot; §7.5's locomotion case.

Foundations

- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning — An Introduction* (2nd ed.). MIT Press. The reference text underneath the entire chapter; Chapters 9–13 cover function approximation through policy gradients.

Chapter summary

Chapter 7 took the tabular reinforcement learning of Chapter 5 and scaled it up to the neural-network function approximators that modern control actually uses. You can now explain why naive value learning with a function approximator diverges and how DQN's replay buffer and target network tame it; derive the policy gradient and name the variance problem that GAE and a critic exist to solve; implement PPO and read its KL curve well enough to diagnose a trust-region violation, having watched one in your own run; choose between an on-policy method like PPO and an off-policy method like SAC by the sample-efficiency gap you measured rather than by reputation; and apply domain randomization to buy transfer across a dynamics gap, while staying honest about the boundary of the distribution you randomized over. That is the deep-RL toolkit that fine-tunes a cloned policy past human performance when a reward and a simulator are in reach. Part 3 now changes the question from *how do we optimize a policy* to *what architecture should the policy be*, beginning, in Chapter 8, with the sequence models that reframe control itself as a prediction problem.

Chapter 8 — Sequence models meet control

8.1 The transformer in two pages, for control

Every architecture in Parts 3 and 4 of this book is a transformer or a close relative. RT-1, RT-2, OpenVLA, Octo, $\pi 0$: strip away the robot plumbing and what remains is a stack of attention layers reading a sequence of tokens and predicting the next one. If you understand that stack, the rest of the book is mostly a matter of asking what the tokens are. This section builds the transformer from the one operation that matters for control, attention, and skips the parts you can read in Appendix C. The promise of "two pages" is a slight exaggeration, but the load-bearing idea really does fit in a paragraph.

Here is that paragraph. A transformer takes a sequence of vectors, called tokens, and produces a new sequence of the same length, where each output vector is a weighted average of all the input vectors. The weights are computed from the inputs themselves: token *i* looks at token *j*, decides how relevant *j* is to it, and mixes *j* in proportionally. Stack a few of these mixing layers with ordinary neural-network layers in between, and you have a model that can route information between any two positions in the sequence in a single step. That is the whole machine. For control, the sequence is a trajectory, states, actions, maybe rewards, maybe a language instruction, and "predict the next token" becomes "predict the next action."

Attention is a soft lookup

The operation underneath everything is called scaled dot-product attention, introduced in its modern form by Vaswani et al. (2017). It is easiest to read as a differentiable dictionary lookup.

Each token produces three vectors by multiplying its embedding by three learned matrices: a *query* q, a *key* k, and a *value* v. Think of a key as a label on a piece of information and a value as the information itself; a query is what a token is currently looking for. To compute the output for token *i*, take its query q_i, compare it against every key k_j in the sequence by dot product, turn those scores into weights with a softmax, and use the weights to average the values:

$$\text{attn}(q_i) = \sum_j \underbrace{\text{softmax}_j\left(\frac{q_i \cdot k_j}{\sqrt{d}}\right)}_{\text{weight } w_{ij}} v_j.$$

The $\sqrt{d}$ in the denominator (with d the dimension of the key vectors) keeps the dot products from growing large enough to saturate the softmax, a small numerical detail with a large effect on whether the thing trains at all, which is exactly the kind of failure §3.5 warned about. Everything else is a dot product and a weighted sum.

The single property worth memorizing: an output token is a content-addressed average. Position *i* does not pull from position *i-1* because of where it sits; it pulls from whichever positions have keys that match its query, wherever they are in the sequence. A gripper-contact event 40 steps ago and the instruction token at the very front are the same distance away as the previous frame, one attention hop. This is the structural reason transformers handle long-range dependence better than the recurrent networks they replaced, where information from 40 steps back had to survive 40 sequential overwrites.

Written as code, the entire operation for a whole sequence is five lines, and it is worth seeing how little there is:

```
import torch, torch.nn.functional as F

def attention(x, Wq, Wk, Wv, causal=True):
    q, k, v = x @ Wq, x @ Wk, x @ Wv     # each (L, d)
    scores = q @ k.T / k.shape[-1]**0.5  # (L, L) all pairs
    if causal:                           # forbid looking ahead
        L = x.shape[0]
        scores = scores.masked_fill(
            torch.triu(torch.ones(L, L), 1).bool(), float("-inf"))
    w = F.softmax(scores, dim=-1)        # (L, L) weights per row
    return w @ v                         # (L, d) mixed values
```

The `scores` matrix is the whole story: row *i*, column *j* holds how much token *i* attends to token *j*. The mask zeroes out the upper triangle so the past cannot read the future, the softmax turns each row into weights that sum to one, and the final product is the weighted average. There is no recurrence and no convolution here, just two matrix multiplies bracketing a softmax.

In practice a layer runs several attention operations in parallel, each with its own query/key/value matrices, *multi-head* attention. One head might track which object the instruction names while another tracks the gripper's state; the heads are concatenated and mixed by a final linear layer. The mechanism is identical; there are just several copies looking for different things.

A transformer is attention plus the boring parts

A full transformer block wraps that attention operation in machinery that, for our purposes, is boring and you can treat as fixed. After attention mixes information across positions, a small two-layer MLP processes each position independently. Both sub-layers are wrapped in a residual connection (add the input back to the output) and layer normalization, the same residual idea that let the deep networks of §3.1 train at all. Stack N such blocks, 6 in the original paper, 12 to 32 in the models we care about, and that is the network.

Two additions make it usable for sequences.

Positional encoding. Attention as written is permutation-invariant: shuffle the tokens and every output is shuffled the same way but otherwise unchanged, because a weighted sum does not care about order. For language or a trajectory, order is the whole point; "place the cup on the plate" is not "place the plate on the cup," and a state-then-action ordering means something different from the reverse. So each token's embedding gets a position signal added to it before the first layer, either a fixed sinusoidal pattern or a learned per-slot vector. Now "which position am I" is part of the content that queries and keys are computed from.

Causal masking. A model that predicts the next token must not be allowed to look at it. During training we feed the whole trajectory at once for efficiency, but we forbid token i from attending to any token $j > i$ by setting those attention weights to zero before the softmax. This is the difference between a *decoder* (causal, left-to-right, the GPT family of Radford et al. 2018, and the right choice for generating actions one at a time) and an *encoder* (bidirectional, used when the whole input is available up front, such as a fixed image). Control policies are almost always causal: at deployment time the robot has its past but not its future.

Why this is a natural fit for control

Reinforcement learning, as set up in Chapter 5, is the study of trajectories: sequences $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$ A policy maps history to the next action. That is, almost verbatim, the sequence-modeling problem a decoder transformer was built for: condition on the tokens so far, predict the next one. The reframing, made explicit by the Decision Transformer of Chen et al. (arXiv:2106.01345) and developed in §8.2, is to stop treating control as a search for an optimal value function and start treating it as autoregressive prediction over trajectory tokens. The Bellman backup of §5.2 disappears; in its place is the same cross-entropy or regression loss we have been minimizing since §3.4.

Three properties of attention pay off specifically for robots. First, variable and heterogeneous inputs: a token is just a vector, so an image patch, a joint-angle reading, a return value, and a word from an instruction can all be tokens in one sequence, and attention learns how much each should influence the action. This is precisely how RT-1 (Brohan et al., arXiv:2212.06817) feeds camera images and a language command into one model; we walk through its tokenizer in §8.4 and §11.3. Second, long-horizon credit: the content-addressed average means an action can attend directly to the instruction issued at $t = 0$ or to the moment a grasp first succeeded, without that signal decaying through intervening steps. Third, scale: the architecture is almost entirely matrix multiplications with no sequential recurrence inside a forward pass, so it saturates GPUs, and, the empirical fact that organizes all of Part 4, its performance keeps improving as data and parameters grow.

The cost, stated plainly

Attention's strength is also its bill. Because every token attends to every other token, an L-token sequence requires on the order of L^2 dot products. Double the trajectory length and you quadruple the compute and memory. For language with thousands of tokens this is the central engineering headache; for control it bounds how much history and how many image patches you can afford per step, and it collides head-on with the real-time deadline of a control loop, a 10 Hz policy has 100 milliseconds to see, think, and command, full stop. That tension drives several later design choices: the action *chunking* of Chapter 10, the dual-system split of Chapter 14 that runs a heavy transformer slowly and a light one fast, and the state-space-model alternative (RoboMamba, arXiv:2406.04339) we reach in §8.5, whose selling point is precisely that it scales linearly rather than quadratically with sequence length.

You now have the one operation, a content-addressed, softmax-weighted average of value vectors, and the scaffolding around it: residual blocks, positional encodings, causal masks. That is enough to read every architecture diagram in the rest of the book; when a paper shows a stack of grey boxes labeled "transformer," you know what is inside each one and why an action can come out the top. The next section spends that understanding on the first concrete instance, recasting offline RL as next-token prediction with the Decision Transformer.

8.2 Decision Transformer: control as conditional sequence modeling

Section 8.1 ended on a reframing: a policy maps a history of tokens to the next action, which is exactly the job a causal transformer was built for. The Decision Transformer (Chen et al., arXiv:2106.01345) is the clean realization of that idea. It throws out the entire apparatus of value functions, Bellman backups, and temporal-difference error that Chapters 5 and 7 spent so long building, and replaces it with a single question: given everything that has happened so far, and a statement of how well we intend to do, what is the next action? Training is ordinary supervised next-token prediction. There is no bootstrapping, no target network, no discount factor to tune. The surprising part is that this works, and on some offline benchmarks it works better than the methods it discards.

The trick: condition on the return you want

Behavior cloning (§6.2) has an obvious flaw as an offline RL method. If your dataset is a mix of good and bad trajectories, some expert, some fumbling, then imitating all of it averages the experts and the fumblers together, and you get a mediocre policy. Plain cloning has no way to ask for *only the good behavior*, because it has no notion of how good any given action was.

The Decision Transformer's one idea is to give the model that notion, at the input. Alongside each state and action, it feeds a third quantity: the *return-to-go*, the sum of rewards from the current step to the end of the episode,

$$\hat{R}_t = \sum_{t'=t}^{T} r_{t'}.$$

Note what this is and is not. It is not the discounted value estimate of a critic; it is just an arithmetic fact computed from the logged data after the episode finished. During training, the return-to-go is known exactly for every step of every trajectory, because the episode is over and we can add up the rewards. So the model is trained to answer:

"in states like this, when the trajectory went on to collect a return of $\hat{R}_t$, what action came next?" That is still pure supervised learning, predict the recorded action, but the action is now *conditioned* on the eventual outcome.

The payoff comes at deployment. We no longer have a logged future to sum up, so instead we *specify* the return we want. We prompt the model with a high target return, say, the best return seen in the dataset, or a bit beyond, and let it generate the action that, in its training experience, preceded outcomes that good. Generate the action, execute it, observe the real reward, subtract it from the target to get the new return-to-go, and repeat. The desired return acts as a knob: ask for a high return and the model produces expert-like behavior even if it was trained on a pile of mediocre data, because it has learned to associate high-return prompts with the competent slices of that data. This is the mechanism that lets it beat behavior cloning on mixed-quality datasets.

The token sequence

Concretely, a trajectory is flattened into a single sequence with three token types per timestep, in a fixed order:

$$\tau = (\hat{R}_1,\, s_1,\, a_1,\, \hat{R}_2,\, s_2,\, a_2,\, \ldots,\, \hat{R}_T,\, s_T,\, a_T).$$

Each of the three is mapped to the model's embedding dimension by its own learned linear layer (or a small convolutional stem, if the state is an image), a timestep embedding is added so the model knows which step a token belongs to, and the whole thing goes into a GPT-style causal decoder of the kind §8.1 described. The causal mask is doing real work here: when the model predicts a_t, it is allowed to attend to every token up to and including s_t, all past returns, states, and actions, plus the current return-to-go and current state, but not to anything later. The training loss is applied only at the action positions: for continuous control, mean-squared error between predicted and recorded actions; for discrete actions, cross-entropy. Returns and states are fed in but not predicted.

The ordering $(\hat{R}, s, a)$ matters and is worth pausing on. Because the return token comes *before* the state and action at each step, the action prediction can attend to the return-to-go directly. The model literally reads "here is how well things are about to go, here is where you are" and then emits an action. Reverse the order and the conditioning signal would arrive too late to inform the action at that step.

A minimal forward pass

The architecture is almost entirely the §8.1 transformer plus three input heads. The only fiddly part is interleaving the three token streams so they line up in the sequence:

```
import torch, torch.nn as nn

class DecisionTransformer(nn.Module):
    def __init__(self, state_dim, act_dim, d=128, K=20):
        super().__init__()
        self.embed_R = nn.Linear(1, d)          # return-to-go
        self.embed_s = nn.Linear(state_dim, d)  # state
        self.embed_a = nn.Linear(act_dim, d)    # action
        self.embed_t = nn.Embedding(K, d)       # timestep
        self.transformer = CausalTransformer(d) # the §8.1 stack
        self.predict_a = nn.Linear(d, act_dim)  # action head

    def forward(self, R, s, a, t):              # each (batch, K, ·)
        te = self.embed_t(t)
        tokens = torch.stack(                    # interleave R,s,a
            [self.embed_R(R) + te,
             self.embed_s(s) + te,
             self.embed_a(a) + te], dim=2
        ).reshape(s.shape[0], 3 * s.shape[1], -1)
        h = self.transformer(tokens)             # causal attention
```

```
        h = h.reshape(s.shape[0], s.shape[1], 3, -1)
        return self.predict_a(h[:, :, 1])        # read off state slots
```

Everything that makes this an RL method is in how the data is prepared, computing returns-to-go, choosing a context window `K` of recent steps, not in the network. The network is a language model that happens to be reading robot trajectories.

Why dropping the Bellman backup is a big deal

It is worth being explicit about what just disappeared. Q-learning and the actor-critic methods of §7.4 estimate a value function by bootstrapping: the value of this state is defined in terms of the estimated value of the next state, which is itself being learned. That self-reference is the source of most of deep RL's instability, the moving targets, the need for target networks and careful learning-rate schedules, the divergence when function approximation, bootstrapping, and off-policy data combine (the "deadly triad" of §7.1). The Decision Transformer sidesteps all of it. Returns-to-go are computed once from fixed data; there is no bootstrapping because there is no value estimate chasing its own tail. The optimization is as stable as training a language model, which after a decade of engineering is very stable indeed. You trade a hard, unstable optimization for an easy, stable one, and let scale and supervised learning do the rest.

This reframing was not unique to one paper. The Trajectory Transformer (Janner et al., 2021), published essentially concurrently, takes the same "offline RL is sequence modeling" stance but uses the transformer as a *model* of the trajectory distribution and plans over it with beam search, the subject of §8.3. The two papers together marked the moment the field started taking "control as next-token prediction" seriously, and they are the conceptual ancestors of every VLA in Part 4 that decodes actions autoregressively.

What it cannot do, and the honest comparison

The Decision Transformer is not free magic, and the chapter's learning objective asks you to articulate the trade-offs against offline actor-critic methods like the CQL/SAC variants of §7.4. Three caveats matter.

First, it does not *stitch*. Value-based methods can, in principle, combine the good first half of one trajectory with the good second half of another, because the Bellman backup propagates value backward across trajectories that share a state. The Decision Transformer only recombines along sequences it can attend to within its context window; if no single logged trajectory ever connected two regions of the state space, prompting for a high return will not conjure the bridge. On benchmark suites built specifically to require stitching, value-based offline RL still wins.

Second, the return prompt is an approximation. Asking for a return far beyond anything in the data does not reliably extrapolate; performance typically peaks when you prompt for a return near the top of the training distribution and degrades if you ask for the impossible. The knob has a working range, and finding it is its own small tuning problem.

Third, it inherits the coverage limits of all offline learning. If the dataset never shows competent behavior in some region, no amount of return conditioning recovers it.

Against those costs sit real advantages: training stability, no critic to diverge, natural handling of sparse and delayed rewards (a return-to-go of zero until a final success is perfectly easy to sum and condition on), and an architecture identical to the sequence models the rest of the book relies on. On the standard D4RL offline benchmark (Fu et al., 2020), the suite the chapter's hands-on exercise uses, including the `hopper-medium` task, the Decision Transformer is competitive with strong offline actor-critic baselines across most locomotion datasets, and notably robust on the sparse-reward and mixed-quality settings where behavior cloning falls apart.

The deeper point, and the reason this section sits where it does, is architectural rather than numerical. Once control is next-token prediction over $(\hat{R}, s, a)$ tokens, the door is open to swap any of those tokens for something richer: replace the scalar return with a language instruction, replace the low-dimensional state with image patches, and you have walked the short distance from Decision Transformer to RT-1. We make that substitution explicit in §8.4, after the next section shows the other thing you can do with a trajectory model, not just generate from it, but search over it.

8.3 Trajectory Transformer and beam-search planning

Section 8.2 used a transformer one way: train it to predict the next action, then at deployment hand it a target return and read off whatever action it emits. The Trajectory Transformer (Janner et al., arXiv:2106.02039) uses the same raw ingredient, a causal transformer over robot trajectories, for a different job. Instead of asking the model "what action comes next?", it asks "what is the probability of this whole future?", and then *searches* for a high-value future the way a language model searches for a high-likelihood sentence. The Decision Transformer is a policy; the Trajectory Transformer is a model of the trajectory distribution that you plan against. The distinction is the whole point of this section, and it is the cleanest example in the book of the difference between a policy and a model.

Two ways to use the same network

It helps to hold the two papers side by side, because they were published within weeks of each other and are constantly confused.

Both flatten a trajectory into a token sequence and both train with ordinary next-token prediction. The Decision Transformer predicts only the action tokens and conditions on a return-to-go you supply at test time; the policy *is* the forward pass. The Trajectory Transformer predicts *everything*, next state, next reward, next return, next action, so the trained model is a joint distribution over how trajectories unfold. A joint distribution by itself does not tell you what to do. To get an action, you sample many candidate futures from the model and keep the one that scores best. That search step is where the planning happens, and it is what §5.2's value iteration did with a known dynamics model: look ahead over possible futures and pick the action that leads somewhere good. The Trajectory Transformer just learned the dynamics model from offline data instead of being handed it.

So the slogan "offline RL as one big sequence modeling problem" cashes out two different ways. In §8.2 it meant *amortize* the planning into a return-conditioned policy. Here it means *learn the model and plan explicitly at test time*. Same architecture, opposite ends of the classic model-free / model-based split.

Discretizing the continuous

The first concrete obstacle is that a `hopper` state is a vector of real numbers, not a word in a vocabulary, and beam search is an algorithm over discrete tokens. The Trajectory Transformer's answer is blunt: discretize every dimension. Each continuous coordinate of the state, each action dimension, the reward, and the return are quantized independently, the paper uses per-dimension binning, so each of the, say, eleven state dimensions of `hopper` becomes one token drawn from a fixed vocabulary of bins. A single timestep is then a short run of tokens,

$$(\underbrace{s_t^1, s_t^2, \ldots, s_t^N}_{\text{state dims}},\ \underbrace{a_t^1, \ldots, a_t^M}_{\text{action dims}},\ r_t,\ \hat{R}_t),$$

and a trajectory is these runs concatenated end to end. Training is plain cross-entropy next-token prediction over this vocabulary, exactly as for text.

To make the bin count concrete: the paper quantizes each dimension into roughly a hundred bins. For `hopper`, with eleven state dimensions, three action dimensions, plus a reward and a return token, one timestep costs sixteen tokens, all drawn from a shared vocabulary of about a hundred symbols. A trajectory of thirty steps is then almost five hundred tokens, comfortably inside a transformer's context window, but a useful reminder of why looking ahead a hundred steps is not free: the horizon you can plan over is the context window divided by sixteen.

The discretization has a cost and a benefit worth naming. The cost is resolution: a binned action is coarser than the raw float, and you spend sequence length, one token per dimension per step, which limits how far you can look ahead within a fixed context window. The benefit is that the model now expresses a full, multimodal distribution over each dimension instead of regressing to a single mean. If, from some state, two genuinely different actions are both reasonable, a regression head (the mean-squared-error action head of §8.2) splits the difference and outputs the average, which may be a bad action belonging to neither mode. A softmax over bins can put probability mass on both and let the search pick one. This is the same multimodality argument that will return, in continuous form, when we reach diffusion and flow-matching action heads in Chapter 10.

Beam search as planning

With the model in hand, control becomes a search over token sequences. Beam search is the standard decoding algorithm from machine translation: keep a "beam" of the B most promising partial sequences, extend each by one token, score the extensions, prune back to the best B, and repeat. In translation you score by accumulated log-likelihood, because you want the most probable sentence. The Trajectory Transformer's one modification is the scoring function. Likelihood alone would just reproduce the behavior policy; the most *probable* future is whatever the dataset did on average, which is the behavior-cloning failure mode again. Instead, sequences are scored by the cumulative reward (or predicted return-to-go) of the future they describe. The beam fills with futures the model believes are both plausible *and* high-value, and the first action of the best surviving beam is the action you execute.

The loop is the receding-horizon pattern from classical model-predictive control: plan a short horizon, take the first action, observe the real next state, and replan from there.

```
def plan(model, state, beam_width=64, horizon=15):
    beams = [(tokenize(state), 0.0)]          # (sequence, value)
    for _ in range(horizon):
        candidates = []
        for seq, value in beams:
            for nxt in model.sample_next_tokens(seq, k=beam_width):
                r_hat = decode_reward(nxt)     # model's predicted reward
                candidates.append((seq + nxt, value + r_hat))
        beams = top_k(candidates, beam_width)  # prune to best B by value
    best_seq, _ = max(beams, key=lambda x: x[1])
    return first_action(best_seq)              # execute, then replan
```

Two details make this honest rather than hand-wavy. First, the model predicts rewards and returns as tokens, so `decode_reward` is reading the network's own estimate of how good a step is; the search is only as good as the learned model, and a model that overestimates reward in unseen regions will plan toward fantasies. Second, scoring partial beams by the predicted return-to-go, not just summed rewards so far, lets the search value a state for what it leads to rather than only what it has collected, the same reason value functions exist in §5.2. The Trajectory Transformer essentially folds an approximate value estimate into the model's own predictions.

Why bother, given the Decision Transformer exists?

If §8.2's policy is simpler, one forward pass, no search, why carry the heavier machinery here? Three reasons, and they map onto the chapter's objective of articulating trade-offs.

The first is *stitching*, the exact weakness §8.2 admitted. Because the Trajectory Transformer searches over futures rather than emitting a single conditioned action, it can chain the good segment of one logged trajectory onto the good segment of another whenever its learned dynamics make the join plausible. On the sparse, long-horizon tasks where stitching matters most, the AntMaze navigation tasks in the D4RL benchmark (Fu et al., arXiv:2004.07219), where reaching the goal requires composing route fragments no single demonstration walked end-to-end, the planning approach is markedly stronger than a return-conditioned policy.

The second is that beam search gives you a knob the Decision Transformer lacks: spend more compute at test time and get a better plan. Widen the beam or lengthen the horizon and the action quality improves, up to the limits of the model. This is the same test-time-compute trade-off that later becomes central to large-model reasoning, appearing here, in 2021, for control.

The third is conceptual and is the reason this section sits between Decision Transformer and the tokenization discussion of §8.4. Once you accept that a transformer can *model* trajectories, planning, prediction, and even system identification all become decoding problems over the same network. The model that plans your next action can also roll out a hypothetical future for inspection, which is a short step from the world models of Chapter 9.

The costs, stated plainly

Beam search is not free. Every control step requires sampling and scoring hundreds of token continuations through a transformer, which is orders of magnitude slower than the Decision Transformer's single forward pass, a serious problem for anything approaching real-time control, and a recurring tension we will quantify when we reach latency budgets in Chapter 14. The per-dimension discretization caps spatial resolution and burns context length, so very high-dimensional observations (raw images, for instance) do not tokenize gracefully this way; this is one reason image-based VLAs in Part 4 lean on the Decision Transformer's amortized-policy style or on diffusion heads rather than on explicit beam search. And like every method in this chapter, it is bounded by the offline data: the learned model hallucinates outside its coverage, and a planner that trusts those hallucinations will confidently plan into them.

The takeaway is not that one approach beats the other but that the same sequence-modeling substrate supports both a fast amortized policy and a slow deliberate planner, and the right choice depends on whether you can afford to think at test time. With both the policy view and the model view of trajectory transformers on the table, the open question is what exactly we should be turning into tokens, scalars, states, actions, returns, or words, which is precisely what §8.4 takes up as it builds the bridge from these RL-flavored sequence models to the language-conditioned VLAs of Part 4.

8.4 What gets tokenized: states, actions, returns, language

The last three sections kept saying "tokenize the trajectory" as if it were one operation. It is not. A trajectory is a heterogeneous thing, a camera frame, a seven-dimensional joint command, a scalar reward, a scalar return, and, in the models of Part 4, a sentence of English, and every one of those quantities has to be turned into an entry in a token sequence before a transformer can touch it. The choices you make there are not plumbing. They decide what the model can represent, how long its sequences get, and whether the language pretraining of a vision-language model survives contact with robot actions at all. This section pulls the tokenization question out of the background and looks at it directly, because it is the hinge between the RL-flavored sequence models of this chapter and the VLAs of Part 4.

The core difficulty is that "token" means two incompatible things. To a transformer, a token is an integer index into an embedding table, a discrete symbol, one of a fixed vocabulary of, say, fifty thousand. But the quantities in a robot trajectory are mostly continuous. Reconciling those two facts is the whole game, and there are exactly two strategies: discretize the continuous thing into a symbol, or keep it continuous and project it into the embedding space with a learned layer. Every system in this book picks one of these per modality, and the interesting part is that they do not pick the same one for every modality.

Two strategies, modality by modality

Take the four quantities in a Decision Transformer trajectory, state, action, return-to-go, reward, and ask of each: symbol or vector?

The Decision Transformer (arXiv:2106.01345) keeps states, actions, and returns *continuous*. It does not bin them. Instead it learns a separate linear projection for each modality, one matrix that maps a state vector into the embedding space, another for actions, another for the scalar return, adds a timestep encoding, and feeds the resulting vectors to the transformer as if they were word embeddings. The transformer never sees an integer index for these; it sees vectors that happen to come from three different little networks. This is the projection strategy, and its virtue is that no resolution is thrown away: a joint angle of 0.4137 radians arrives intact.

The Trajectory Transformer (arXiv:2106.02039), as §8.3 described, does the opposite. It bins every dimension of the state and action and the reward and the return into a discrete vocabulary, roughly a hundred bins per dimension, and trains with plain cross-entropy over those symbols. This is the discretization strategy. Its virtue is the one §8.3 dwelt on: a softmax over bins is a full distribution, so the model can be honestly multimodal, and you can run beam search over the resulting symbols. Its cost is resolution and sequence length, one token per dimension per step.

So two papers, published weeks apart, on the same hardware benchmark, made opposite tokenization calls, and both worked. That is the lesson to internalize before Part 4: tokenization is a design axis, not a settled convention,

and the right choice depends on what you are going to *do* with the tokens. If you plan to search (§8.3), you want discrete symbols. If you plan to regress a single action and care about precision, you lean toward projection. Hold that tension; it explains the entire RT-1 → RT-2 → π0 progression.

Why returns are the strange token

The return-to-go deserves a moment of its own, because it is the token that has no analogue in language and the one beginners most often mishandle. In §8.2 the return-to-go was the conditioning signal: you prepend the performance you *want* and the model produces actions consistent with it. Tokenizing it is easy, it is a scalar, project it or bin it like any other, but its *semantics* are unusual. Unlike a state or an action, the return is not something the world hands you; it is a command you supply at test time, and it can be a lie. Asking for a return higher than anything in the dataset is exactly how you probe a Decision Transformer's extrapolation, and it usually fails gracefully into "the best behavior I saw," not magic. The point for tokenization is that the return token occupies a real slot in the vocabulary and the sequence, and it costs you context length just like everything else. When VLAs in Part 4 drop the return token entirely, RT-1 and successors are imitation learners with no return conditioning, they are buying back that slot and that conceptual complication, which is one quiet reason the foundation models look simpler than the offline-RL models they descend from.

Tokenizing actions for a language model

Now the bridge the chapter promised. RT-1 (arXiv:2212.06817) is the first system in this book to tokenize robot actions specifically so they can live in the same sequence as image and language tokens. Its scheme is the discretization strategy, applied deliberately: each dimension of the robot's action, three for end-effector position, three for rotation, one for the gripper, plus a mode switch, is uniformly discretized into 256 bins. An action becomes a short string of integers in the range 0–255, and the transformer predicts them autoregressively, exactly as it would predict the next word.

The number 256 is not arbitrary, and this is the idea that detonates in RT-2 (arXiv:2307.15818). A pretrained vision-language model already has a vocabulary of tens of thousands of tokens. RT-2's move is to *reuse* that vocabulary for actions: it takes 256 token IDs the language model already has, in the published version, the integer-string tokens, or rarely used ones reassigned to mean action bins, and declares them to be the action alphabet. No new embedding table, no new output head. The robot's action space is overloaded onto symbols the model learned during web pretraining. Producing an action is then *literally the same operation* as producing text: emit a token. This is why RT-2 can be fine-tuned from a web-scale VLM without architectural surgery, and why the web knowledge partly survives; the model is doing the only thing it ever knew how to do, predict the next token, and some of those tokens now happen to mean "move left." When the chapter's learning objectives say to connect sequence modeling to "the action-token decoding scheme used by RT-1/RT-2," this overloading is the scheme.

A small worked example makes the sequence concrete. A single RT-2 training example, flattened, looks like

```
<img patch tokens> "pick up the can"
  → 134 211 6 248 9 17 0
```

where the seven trailing integers are the binned action: position, rotation, gripper, and terminate. The loss is cross-entropy on those seven tokens. Nothing about the machinery distinguishes them from the caption tokens that the same model would emit on a web image. That uniformity is the entire trick, and its limits drive Part 4.

When discretizing actions stops being free

Uniform binning has a flaw that §8.3 only gestured at and that becomes acute for high-frequency control. Robot actions are correlated and smooth over time; a 50 Hz controller emits action vectors that barely change from step to step. Binning each dimension independently throws that structure away and spends a fixed token budget per step regardless of how much actually happened. Worse, fine manipulation needs fine resolution exactly where uniform bins are coarsest, near zero, where most action deltas live.

FAST (arXiv:2501.09747) is the clean fix and a good note to end on, because it shows tokenization is still an active research frontier rather than a solved preprocessing step. Instead of binning raw actions, FAST applies a discrete

cosine transform to short chunks of the action trajectory, the same compression idea used in JPEG, and tokenizes the frequency coefficients. Smooth motion compresses to a handful of tokens; the scheme spends sequence length in proportion to how much the action actually varies, and it gives high-frequency continuous-control VLAs an action vocabulary that is both compact and high-resolution. Concretely, where uniform binning might spend dozens of tokens encoding a near-static arm, the frequency representation collapses that stretch into a few coefficients and reserves resolution for the moments where the motion actually changes, the budget allocation a 50 Hz controller needs. The same paper reports that this compression both shortens training sequences and improves downstream control, a reminder that the right token format can buy accuracy and efficiency at once rather than trading them off. The takeaway is the section's thesis restated with a real artifact behind it: *how* you turn actions into tokens sets a ceiling on how well a transformer can control a robot, and moving that ceiling has been worth its own papers.

Two strategies, discretize into symbols or project as vectors, applied independently to states, actions, returns, and language, and overloaded onto a pretrained vocabulary when you want web knowledge to come along for free: that is the tokenization toolkit you carry into the rest of the book. With it in hand, the only thing left in this chapter is to make the bridge explicit, to trace exactly how a sequence model trained on tokens becomes a foundation action model, which is what §8.5 does next.

8.5 Bridge to foundation action models, and to the SSM alternative (RoboMamba)

This chapter started from an RL problem, offline trajectories, returns, the Bellman equation lurking in the background, and ended up with a machine that predicts the next token. The Decision Transformer and the Trajectory Transformer are both, structurally, language models that happen to read robot data. That structural fact is the bridge. Everything in Part 4 of this book is the same machine pointed at more data, more modalities, and a vision-language model's pretrained weights. This section makes the crossing explicit: it traces the four changes that turn a sequence model into a foundation action model, and then it does something the rest of Part 4 will not, which is to show that the transformer is not the only sequence model that can make the crossing. RoboMamba (arXiv:2406.04339) gets there with a state space model instead, and seeing how it differs sharpens what the transformer was actually buying you.

Four changes, and nothing else

Take a Decision Transformer and ask what you would have to change to turn it into RT-2 (arXiv:2307.15818). The list is short, and its shortness is the point.

First, drop the return. The Decision Transformer conditions on a return-to-go token because it is doing offline RL: you tell it how well to perform and it complies. The foundation action models are imitation learners (Chapter 6): they condition on a language instruction instead of a number, and they try to reproduce the demonstrated behavior, not to hit a commanded score. So the return token, the one §8.4 called the strange token with no analogue in language, simply goes away. In its place sits a sentence: "pick up the can." The conditioning signal changes from a scalar you invent at test time to a string a human types, and the model's job changes from "behave this well" to "do this thing."

Second, swap the data. A Decision Transformer trains on one task's worth of trajectories, D4RL hopper, say, a few million transitions from a single MuJoCo robot. RT-1 (arXiv:2212.06817) trains on roughly 130,000 demonstrations spanning 700-plus tasks on a real mobile manipulator, and its successors train on Open X-Embodiment, which pools data from dozens of labs and robot bodies (Chapter 12 takes the dataset apart). The architecture barely moves; the data moves by three orders of magnitude. This is the part beginners underestimate. The sequence-modeling recipe was sitting there in 2021; what made it a *foundation* model was pointing it at a corpus large and diverse enough that the next-token objective started producing behavior that generalized across tasks it had never seen jointly. A useful way to hold this: the Decision Transformer answered "can a sequence model do control?" and the foundation models answered "what happens to a sequence model at robot-data scale?" Those are different questions, and the second one has no clean theoretical answer; you find out by training the thing, which is why so much of Part 4 reads like engineering reports rather than theorems.

Third, initialize from a vision-language model rather than from scratch. The Decision Transformer's embedding tables are random at the start of training. RT-2's are not: it begins life as PaLI-X or PaLM-E, a model already

trained on web-scale image-text pairs, and robot data is poured in afterward. This is why §8.4 spent so long on the action-token overloading trick. Because actions are encoded as token IDs the vision-language model already owns, fine-tuning on robot data does not require bolting on a new output head; producing an action is the same operation as producing a word, so the web knowledge in those weights partly survives the transition. OpenVLA (arXiv:2406.09246) follows the identical pattern with an open Llama-2-based backbone and a SigLIP/DINOv2 vision stack, which is why it can be a 7-billion-parameter model that a single lab can actually fine-tune.

Fourth, and this one is optional, which is the seam Chapters 10 and 13 pry open, reconsider the action head. RT-1 and RT-2 keep the discrete action tokens from §8.4 all the way to the output: the model emits binned integers and you de-bin them into a motion. Octo (arXiv:2405.12213) and $\pi 0$ (arXiv:2410.24164) instead attach a continuous head, a diffusion or flow-matching decoder, onto the same transformer trunk, trading the clean "everything is a token" story for smoother, higher-frequency control. Hold that thread; it is the whole argument of Chapter 13.

That is the bridge. Drop the return, scale the data, start from a VLM, optionally change the head. No new core idea is required between the Decision Transformer and a foundation action model, which is both the good news (you already understand the architecture) and the uncomfortable news (most of the progress came from data and compute, not from a clever mechanism).

The cost the transformer never stopped charging

There is a tax buried in that architecture, and it is the reason the rest of this section exists. Attention compares every token with every other token, so its cost grows with the square of the sequence length. For a sentence that is fine. For a robot that streams images at 30 Hz, keeps a history of frames, and must emit an action within a few tens of milliseconds to stay in its control loop, quadratic attention is a real budget problem. A 7-billion-parameter VLA like OpenVLA produces actions at a handful of hertz on a good GPU, usable, but not comfortable for contact-rich control, and not cheap. Chapter 14's dual-system architectures exist partly to work around exactly this latency, splitting a slow deliberative model from a fast reactive one.

So a fair question, which the foundation-model literature mostly answers with "buy a bigger GPU," is whether the sequence model underneath has to be a transformer at all. The next-token recipe, read a sequence of tokens, predict the next one, does not actually require attention. It requires *some* sequence model. Attention is one choice, and a famously expensive one.

RoboMamba: the same recipe on a state space model

Mamba (Gu and Dao, 2023) is a state space model, an SSM: instead of attending over the whole history at every step, it maintains a fixed-size recurrent state and updates it as tokens stream in. The consequence that matters here is the cost curve. Mamba's inference is *linear* in sequence length, not quadratic, because each new token does a constant amount of work against the running state rather than re-examining every past token. The selective-state mechanism that makes Mamba competitive with attention on language is a section's worth of detail this book does not need; the load-bearing fact for us is the complexity class. A transformer carries its entire history forward explicitly and pays to look at all of it every step; an SSM compresses the history into a bounded state and pays a flat rate per step. That compression is a bet, it assumes the running state can summarize what matters, and for streaming sensor data, where the recent past dominates and the relevant context is bounded, it is often a good bet.

RoboMamba (arXiv:2406.04339) takes the four-change recipe above and runs it with Mamba where the transformer used to be. The construction is faithful to the pattern. It pairs a vision encoder with a Mamba backbone and co-trains so that visual tokens align with the language embedding space, which gives the model visual common sense and robot-relevant reasoning, the SSM analogue of starting from a VLM. Then, to get actions, it attaches a simple policy head that predicts an SE(3) end-effector pose, and fine-tunes only that head. The headline numbers are the argument for the whole approach: RoboMamba reaches manipulation competence by updating roughly 0.1% of its parameters, and it runs inference about three times faster than comparable transformer VLAs. The linear-cost backbone is doing visible work.

What should you take from this? Not that SSMs win, the foundation models that defined the field, and the ones this book spends Part 4 on, are transformers, and the largest, best-generalizing action models remain transformer-based as of this writing. The lesson is the more useful one for a student trying to read the field clearly: the *recipe*

is separable from the *architecture*. Tokenize the world, condition on language, initialize from a model that already understands images and text, fine-tune a small action head. That recipe ran on a transformer in RT-2 and on a state space model in RoboMamba, and both produced a working vision-language-action model. When you read Chapter 13 on $\pi 0$ or Chapter 14 on dual-system designs, keep asking which parts of the system are the recipe and which are the architecture, because conflating the two is how people end up believing that attention is load-bearing when it is often just incumbent.

It is worth being honest about why transformers still won the field despite the tax. Two reasons. The pretrained weights that make a foundation action model work, the web knowledge in PaLI-X, Llama-2, SigLIP, exist for transformers because the entire language and vision community standardized on them first; you cannot initialize from a vision-language model that nobody bothered to train. And attention's ability to reach any token directly, the very thing that costs you the quadratic factor, pays off when a task depends on a detail many steps back. RoboMamba's reasoning is competitive, but it leans on a co-trained vision encoder to supply the heavy multimodal grounding, and the largest, most general action models published to date are still transformers. The SSM result is a proof of possibility and an efficiency argument, not a changing of the guard, which is exactly why it belongs in a bridge section rather than a chapter of its own.

What the bridge does and does not carry

One caution before the chapter closes. The bridge carries the *mechanism*, next-token prediction over a tokenized trajectory, cleanly into Part 4. It does not carry the *guarantees*. The Decision Transformer sat inside an offline-RL framing where you could at least reason about returns and optimality; the foundation action models drop that scaffolding and become pure imitation learners, inheriting the compounding-error problem from §6.3 and offering no value function to fall back on. They are more capable and less analyzable at the same time. The smoothness fixes (diffusion and flow heads, Chapter 10), the data-scale arguments (Chapter 12), and the safety layers (Chapter 17) all exist in part to manage what gets lost when you trade the RL framing for the language-modeling one. The bridge is real, but it is one-way: you can walk a sequence model across into a foundation action model, and once there, the RL tools you started this chapter with mostly stay behind.

With the crossing mapped, the chapter has done its job: you can now see RT-1, RT-2, OpenVLA, and even the SSM-based RoboMamba as the same sequence-modeling idea from §8.1 scaled up and re-pointed. The summary in §8.6 collects the chapter's load-bearing ideas before Part 4 starts building on them in earnest.

8.6 Summary

Chapter 8 was the hinge of the book. Everything before it, classical planning, MDPs, reinforcement learning, imitation, described ways of producing actions that were invented for control and stayed in control. This chapter introduced a machine that was invented for something else entirely, next-token prediction over text, and showed that pointing it at robot data produces a controller. That single reframing is what connects the lineage of Part 2 to the foundation action models of Part 4, and the chapter's job was to make the reframing concrete enough that the rest of the book reads as engineering rather than magic. This summary collects the load-bearing ideas and marks where Parts 3 and 4 keep reaching back for them.

The four ideas worth carrying forward

Control can be cast as sequence modeling, and that reframing is the whole chapter. §8.1 and §8.2 made the case that a trajectory, states, actions, and whatever else you choose to record, is just a sequence of tokens, and that predicting the next token in that sequence is a way to act. The Decision Transformer (arXiv:2106.01345) is the cleanest demonstration: feed it a desired return, the recent history, and the current state, and it predicts the action a successful trajectory would have taken next. What makes this more than a notational trick is what it discards. There is no Bellman backup, no bootstrapping, no value function chasing its own target; the instability machinery that Chapter 7 spent five sections taming simply does not appear, because supervised next-token prediction has none of the moving parts that make temporal-difference learning fragile. You trade the theoretical scaffolding of RL for the empirical robustness of supervised learning, and for offline data that is often a good trade.

Conditioning is the control knob, and what you condition on defines the method. The Decision Transformer conditions on a return-to-go: you tell it how well to do, and it complies (§8.2). The Trajectory Transformer (arXiv:2106.02039) instead conditions on nothing in particular and uses beam search over its own predictions to plan (§8.3). The foundation models of Part 4 condition on a language instruction. Same architecture, same next-token objective, three different conditioning signals, and the conditioning signal is what determines whether you have an offline-RL method, a planner, or an imitation learner. Internalizing that the architecture is fixed and the conditioning is the design space is what lets you see RT-2 and a Decision Transformer as relatives rather than strangers.

Tokenization is a modeling decision with consequences, not a preprocessing chore. §8.4 was the technical heart of the chapter. Putting states, actions, returns, and language into one token stream forces a series of choices, discretize a continuous action into bins, or attach a continuous head; reuse a vision-language model's existing vocabulary, or add new symbols, and each choice propagates all the way to control quality. The action-token overloading trick, where RT-2 (arXiv:2307.15818) encodes actions as token IDs the vision-language model already owns, is the move that lets web knowledge survive fine-tuning, and it is only available because someone treated tokenization as a design problem. The flip side, which §8.4 was careful to state, is that naive uniform binning wastes resolution and caps your control frequency, the failure that FAST (arXiv:2501.09747) and the continuous heads of Chapter 10 exist to fix.

The recipe is separable from the architecture. §8.5 made the crossing into Part 4 explicit and then complicated it productively. Turning a Decision Transformer into a foundation action model takes exactly four changes, drop the return for a language instruction, scale the data by orders of magnitude, initialize from a vision-language model, and optionally swap the action head, and no new core idea. RoboMamba (arXiv:2406.04339) then showed that the same recipe runs on a state space model instead of a transformer, reaching manipulation competence while updating about 0.1% of its parameters and running several times faster. The point is not that state space models win; the largest, best-generalizing action models remain transformers. The point is that you should read every system in Part 4 by asking which parts are the recipe and which are the architecture, because conflating the two is how people end up believing attention is load-bearing when it is often just incumbent.

What you should be able to do now

Four concrete capabilities, in roughly the order the rest of the book will ask for them.

You should be able to *explain how a transformer produces an action, end to end, without hand-waving the attention step.* Not "it's a neural net that outputs actions," but the actual path: tokens in, attention as a soft lookup that lets each token gather information from the others, causal masking so the model only sees the past, and a prediction head that emits the next token, which, depending on what you tokenized, is an action. §8.1 gave you the two-page version specifically so you could read the architecture sections of Part 4 without stopping to relearn attention each time.

You should be able to *take a control problem and say what its token sequence is.* Given a task, you should be able to lay out what gets tokenized and in what order, which observations, whether a return or a language instruction does the conditioning, how the action is represented, and justify each choice against the trade-offs in §8.4. This is the skill that turns a vague "we'll use a transformer" into an actual design, and it is the same skill Chapter 11 exercises when it dissects the VLA recipe component by component.

You should be able to *place the Decision Transformer and the Trajectory Transformer relative to the offline RL of Chapter 5, and say what each gives up.* The Decision Transformer trades the Bellman backup for return-conditioned supervised learning, which buys stability and costs you the ability to stitch together better-than-demonstrated behavior the way dynamic programming can (§8.2). The Trajectory Transformer keeps a notion of planning by searching over its predictions, at the cost of inference time (§8.3). Naming what each method gained and surrendered, rather than ranking them, is the analytical habit the chapter was training.

You should be able to *read a foundation action model as a scaled-up sequence model and locate the four changes.* Shown RT-1, RT-2, OpenVLA, or RoboMamba, you should be able to point to where the return became a sentence, where the data scaled, where the vision-language initialization entered, and what the action head is. That decomposition is the throughline of Part 4, and §8.5 built it precisely so the later chapters could assume it.

Where the chapter has set up the rest of the book

Chapter 8 hands off in three directions, and all three are central to the remaining parts.

The most immediate is to the generative action heads of Chapter 10. §8.4 ended on the limits of discrete action tokens, wasted resolution, capped frequency, the awkwardness of binning a smooth motion, and that unfinished problem is exactly what diffusion and flow-matching heads solve. When Chapter 10 attaches a diffusion decoder to a transformer trunk, it is answering the question §8.4 left open: how do you keep the sequence-model backbone but stop forcing continuous control through a discrete bottleneck. $\pi 0$'s flow-matching objective in Chapter 13 (arXiv:2410.24164) is the most developed answer.

The deeper handoff is to the entirety of Part 4. §8.5's four-change recipe is the skeleton on which Chapters 11 through 14 hang their detail. Chapter 11 walks the recipe from CLIP to RT-1; Chapter 12 takes apart the data-scaling step with Open X-Embodiment; Chapters 13 and 14 push on the action head and on splitting one model into two. A reader who holds the recipe in mind will recognize each of those chapters as elaborating one change rather than introducing a new paradigm, which is the correct way to read them, because the paradigm shift already happened, here, in Chapter 8.

The third handoff is to the latency and efficiency discussion that runs through Chapter 14 and Chapter 16. §8.1 and §8.5 were blunt about the quadratic cost of attention and what it means for a robot that must close a control loop in tens of milliseconds. RoboMamba's linear-cost backbone was the first answer; Chapter 14's dual-system architectures, which split a slow deliberative model from a fast reactive one, are the second. The student who took the cost discussion seriously will not be surprised when deployment chapters spend as much effort on inference budget as on accuracy.

What the chapter has not covered

Two omissions are worth naming. The chapter treated the sequence model almost entirely as an imitation or offline-RL device and said little about *online* sequence-model control, using a transformer policy inside a live RL loop, learning from fresh interaction. That is a deliberate framing choice, not an oversight: the foundation action models the book builds toward are imitation learners, and the online-RL fine-tuning of a large sequence policy belongs with the fine-tuning machinery of Chapter 16, where the sample-budget arguments of Chapter 7 do the real work. Read this chapter's silence on online learning as a deferral.

The chapter also stayed deliberately shallow on the internals of the sequence models themselves. §8.1 gave attention two pages, and §8.5 gave the selective-state mechanism of Mamba a single sentence, because this is a book about action models, not about architecture research. A reader who wants the full derivation of multi-head attention, positional encodings, or the state space recurrence will need a dedicated transformer reference; what this chapter supplied is the working understanding required to read the architecture as a component of a controller. That is the right depth for the book's purpose, but it is worth being honest that it is a floor, not a ceiling.

Chapter 8's contribution to the book's overall argument is to perform the single conceptual move the whole second half depends on: recasting action generation as next-token prediction, and showing that the move is small in mechanism and enormous in consequence. The three findings to carry forward are that control-as-sequence-modeling trades RL's theoretical scaffolding for supervised robustness, that conditioning, not architecture, is what distinguishes an offline-RL method from a planner from an imitation learner, and that the foundation-model recipe is separable from the transformer that usually carries it. Part 3 now picks up the generative and world-model machinery that, layered onto this sequence-modeling foundation, produces the smooth, scalable action models of Part 4.

§8.x closes the chapter with a hands-on exercise, training a small Decision Transformer on an offline dataset and probing how its behavior changes as you sweep the target return, and the full reading list for the chapter.

8.x Hands-on exercise + chapter references

Chapter 8 rests on one claim that is easy to accept on paper and only becomes real once you have trained the model yourself: a transformer with no value function, no Bellman backup, and no notion of optimality can still produce

competent actions, purely by predicting the next token of a trajectory the way GPT predicts the next word of a sentence. The drills below make that claim something you have watched happen rather than something you have been told. The headline exercise trains a small Decision Transformer on an offline dataset and then sweeps the target return at test time, so you can see the conditioning knob §8.2 described actually steer behavior. The remaining drills probe the boundaries of the reframing: where return conditioning stops working, what tokenization costs you, and how to read a foundation action model as the same machine scaled up.

The exercises take two to three hours combined. The model is deliberately tiny, a few hundred thousand parameters, two or three attention layers, so a full training run finishes in minutes on a CPU and you can afford to rerun it while you change one thing at a time. After the dependency install nothing here needs internet.

```
pip install torch
```

You need one offline dataset of `(state, action, reward)` trajectories. The cleanest choice is a D4RL task (Fu et al., arXiv:2004.07219) such as `hopper-medium`, which is the dataset the Decision Transformer paper (arXiv:2106.01345) itself reports on; if installing D4RL's MuJoCo dependency is more friction than you want, log a few thousand episodes from a CartPole policy of mixed quality and use that. The lesson does not depend on the task, only on the data being offline and of varied return.

Exercise 8.x.1 — A Decision Transformer, and the return knob

Build `dt.py` following §8.1 and §8.2. The architecture is the §8.1 transformer with nothing exotic: embed each of the three token types, return-to-go, state, action, into the same width, add a positional (here, timestep) signal, stack two or three causal attention blocks, and put a head on the action positions that predicts the next action. Train with the ordinary supervised loss from §3.4: cross-entropy if your actions are discrete, mean-squared error if they are continuous. There is no target network, no replay-buffer staleness, none of the Chapter 7 machinery; that absence is the point of the chapter, and you should notice how much shorter this training loop is than the SAC loop from Exercise 7.x.2.

Precompute the returns-to-go for every timestep, the sum of rewards from that step to the end of its episode, because that is the quantity the model conditions on. Train until the action-prediction loss plateaus.

Now the actual experiment. At evaluation time you supply the first return-to-go token yourself; it is a request, not a measurement. Roll out the policy in the live environment across a sweep of target returns, say five values from well below the dataset's average return to somewhat above its maximum, and record the *achieved* return at each. Plot requested return on the x-axis against achieved return on the y-axis.

The expected picture, and §8.2's central demonstration, is a band where the two track each other: ask for more, get more. Read the two ends carefully, because they are where the method's nature shows. At the low end the policy will obligingly underperform, proof that you are steering behavior, not just maximizing it. At the high end the curve flattens: ask for a return larger than anything the data contains and the model cannot deliver it, because it only ever learned to imitate trajectories that existed. That ceiling is the honest limit §8.2 named, a return-conditioned imitator cannot invent better-than-demonstrated behavior the way the dynamic programming of §5.2 can stitch one together. Record where your curve stops climbing; that knee is the edge of the dataset.

Wall clock: about forty-five minutes including the sweep.

Exercise 8.x.2 — Where stitching breaks, and why the planner exists

This drill makes §8.3's policy-versus-model distinction concrete by finding a case the Decision Transformer gets wrong. Construct, or pick from D4RL, a dataset whose good behavior is *fragmented*: no single logged trajectory is high-return end to end, but a high-return path exists if you splice the good first half of one episode onto the good second half of another. The classic illustration is a dataset of trajectories that each do one subtask well and the other badly.

Train your Decision Transformer on it and ask for a high return. It will typically fail to splice, because return-conditioned imitation reproduces whole trajectories and never learned the join. This is the concrete face of the "no stitching" limitation §8.2 and §8.3 kept flagging. If you have the appetite, contrast it with a value-based offline method,

even a small CQL or the IQL baseline D4RL ships, which can stitch because it propagates value through the Bellman backup the Decision Transformer threw away. You do not have to implement the planner; the Trajectory Transformer (arXiv:2106.02039) recovers stitching by searching over its own predictions instead, at the cost of inference time §8.3 measured. The deliverable is a short written answer: state precisely what information the Bellman backup carries that return-conditioned supervised learning does not, and why beam search over a trajectory model is a third way to recover it.

Wall clock: about thirty minutes if you reuse 8.x.1's code and skip the value-based baseline; an hour if you run the comparison.

Exercise 8.x.3 — Tokenize an action, and pay the resolution bill

§8.4 argued that tokenization is a modeling decision with consequences, not a preprocessing chore. Feel the consequence. Take a continuous control task and discretize each action dimension into uniform bins, start with 256 bins, the RT-1-style choice, then retrain the Decision Transformer to predict bin indices with cross-entropy. Sweep the bin count down: 256, 64, 16, 8. Plot achieved return against bin count.

You will watch control quality fall off as the bins coarsen, and the fall-off is not linear; there is a cliff below which the discretization can no longer represent the smooth corrections the task needs, and the policy starts to chatter or stall. That cliff is exactly the wasted-resolution failure §8.4 warned about and the reason FAST (arXiv:2501.09747) and the continuous diffusion and flow-matching heads of Chapter 10 exist. For the written part, estimate the effective control frequency your tokenization implies: if each action costs one token and your model emits tokens at some rate, how fast can the loop close? That back-of-envelope number is the §8.1 cost argument made personal, and it is the constraint Chapter 14's dual-system split is designed to dodge.

Wall clock: about forty minutes including the sweep.

Exercise 8.x.4 — Read a foundation action model and find the four changes

Pick one Part 4 system from the reading list, RT-1 (arXiv:2212.06817), RT-2 (arXiv:2307.15818), or OpenVLA (arXiv:2406.09246) are the cleanest, and read it with §8.5's recipe in hand. Write a short audit that locates, by figure or paragraph, each of the four changes that turn a Decision Transformer into a foundation action model:

1. **Where did the return become a language instruction?** Find the conditioning input and confirm it is a sentence, not a scalar.
2. **Where did the data scale, and by how much?** Name the dataset and its order of magnitude relative to the few-thousand-episode buffer of Exercise 8.x.1.
3. **Where did the vision-language initialization enter?** Identify the pretrained backbone and what it was trained on before it ever saw a robot.
4. **What is the action head, and how are actions tokenized?** Discrete bins, overloaded vocabulary IDs, or a continuous head, and what §8.4 trade-off that choice accepts.

Then answer one synthesis question: which parts of this system are the recipe and which are the architecture? The §8.6 warning was that conflating the two is how people end up believing attention is load-bearing when it is often just incumbent, and RoboMamba (arXiv:2406.04339), which runs the same recipe on a state space model, is the evidence. The point of the audit is that once you can decompose a VLA into these four moves, every chapter of Part 4 reads as elaborating one of them rather than introducing a new paradigm.

Wall clock: about thirty minutes for the read plus the written audit.

Chapter 8 reading list

The works below are cited in §8.1–§8.6, grouped by purpose. Full bibliographic entries for everything cited in the book live in Appendix E.2; this is the chapter-local subset.

The transformer itself

- Vaswani, A., et al. (2017). "Attention Is All You Need." *NeurIPS 2017*. The scaled dot-product attention §8.1 builds the whole chapter on.
- Radford, A., et al. (2018). "Improving Language Understanding by Generative Pre-Training." OpenAI tech report. GPT; the causal decoder §8.1 says a control policy almost always is.

Control as sequence modeling

- Chen, L., et al. (2021). "Decision Transformer: Reinforcement Learning via Sequence Modeling." arXiv:2106.01345. §8.2's main result and Exercise 8.x.1's algorithm; the return-conditioned imitator whose ceiling Exercise 8.x.1 measures.
- Janner, M., et al. (2021). "Offline Reinforcement Learning as One Big Sequence Modeling Problem" (Trajectory Transformer). arXiv:2106.02039. §8.3's planner; the model-not-policy framing and the beam search that recovers the stitching Exercise 8.x.2 exposes.
- Fu, J., et al. (2020). "D4RL: Datasets for Deep Data-Driven Reinforcement Learning." arXiv:2004.07219. The offline-RL benchmark the exercises draw their datasets from.

Tokenization and its limits

- Brohan, A., et al. (2022). "RT-1: Robotics Transformer for Real-World Control at Scale." arXiv:2212.06817. The discrete action-binning scheme §8.4 dissects and Exercise 8.x.3 stress-tests.
- Pertsch, K., et al. (2025). "FAST: Efficient Action Tokenization for Vision-Language-Action Models." arXiv:2501.09747. The fix for the wasted-resolution problem Exercise 8.x.3 makes you feel.

The bridge to foundation action models

- Brohan, A., et al. (2023). "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control." arXiv:2307.15818. §8.4's action-token overloading and §8.5's four-change recipe; Exercise 8.x.4's target.
- Kim, M. J., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246. The open 7B VLA §8.5 points to and Chapter 12 takes apart.
- Liu, J., et al. (2024). "RoboMamba: Efficient Vision-Language-Action Model for Robotic Reasoning and Manipulation." arXiv:2406.04339. §8.5's state space model alternative; the evidence that the recipe is separable from the transformer.
- Gu, A., & Dao, T. (2023). "Mamba: Linear-Time Sequence Modeling with Selective State Spaces." The linear-cost backbone underneath RoboMamba and §8.1's answer to the quadratic-attention bill.

Chapter summary

Chapter 8 performed the single conceptual move the second half of the book depends on: it recast action generation as next-token prediction and showed that the move is small in mechanism and large in consequence. You can now explain how a transformer produces an action end to end, tokens in, attention as a content-addressed average, causal masking, a prediction head on the action positions, without hand-waving the attention step. You can take a control problem and say what its token sequence is, justifying each tokenization choice against the resolution-versus-frequency trade-off you measured by hand. You can place the Decision Transformer and the Trajectory Transformer against the offline RL of Chapter 5 and name what each gives up: the policy that imitates returns but cannot stitch, the model you plan against but pay inference time for. And you can read a foundation action model as a scaled-up sequence model, locating the four changes, language for return, data at scale, a vision-language initialization, an action head, that §8.5 showed are the whole difference. Part 3 now picks up the generative and world-model machinery that, layered onto this sequence-modeling foundation, produces the smooth, scalable action models of Part 4, beginning, in Chapter 9, with world models and model-based learning.

Chapter 9 — World models and model-based learning

9.1 What is a world model, really

Chapter 8 left the policy doing all the work. A Decision Transformer reads a trajectory and predicts the next action; nothing in it ever asks what that action *does*. The model has no opinion about the future beyond the action it is about to emit. That is fine when you have enough demonstrations to imitate, and it is the design every VLA in Part 4 inherits. But it throws away a capability that classical control took for granted: the ability to answer "what happens if I do this?" before committing to it.

A world model is the machine that answers that question. Give it the current state and a candidate action, and it predicts the next state, and usually the reward, and sometimes the raw observation that the robot's cameras would return. It is, in one phrase, a learned simulator of the environment, trained from the agent's own experience rather than written by hand. Everything else in this chapter is a variation on that idea: how to learn the simulator (§9.2), how to use it to choose actions (§9.3), how to scale it to video (§9.4), and whether it should replace the policy-only VLA entirely (§9.5).

The function you are trying to learn

Recall the transition function from the MDP of §5.1. The environment moves according to

$$s_{t+1} \sim P(s_{t+1} \mid s_t, a_t), \qquad r_t = R(s_t, a_t).$$

In a real robot you do not have P or R in closed form. A world model is a parametric approximation of them, call it $\hat{P}_\theta$ and $\hat{R}_\theta$, fit by watching transitions (s_t, a_t, s_{t+1}, r_t) roll past and minimizing prediction error. That is the entire definition. A world model is supervised learning on the dynamics, where the labels come for free because the next state is simply the thing that happened next.

This connects directly to §4.3, where we wrote the forward dynamics of a manipulator as $\ddot{q} = M(q)^{-1}(\tau - C(q,\dot{q})\dot{q} - g(q))$ and called it a model of the robot. That equation *is* a world model, an analytic, hand-derived one, valid for the rigid body and useless the moment you point a camera at a pile of laundry. The learned world models in this chapter are the same object built by a different route: instead of deriving the dynamics from physics, you regress them from data. The trade is the usual one. The analytic model is exact within its assumptions and blind outside them; the learned model is approximate everywhere and, with enough data, approximate in places physics gives you nothing.

A model predicts; a policy acts

It is worth being pedantic about the distinction, because the rest of Part 3 and Part 4 hinges on it. A *policy* is a function from state to action, $\pi(a \mid s)$, "what should I do?" A *world model* is a function from state and action to next state, $\hat{P}(s' \mid s, a)$, "what would happen if I did that?" They point in opposite directions. You can hold one without the other. A chess engine that evaluates positions but cannot move is missing a policy; a behavior-cloned arm that grasps cups but cannot tell you what the cup will do once grasped is missing a world model.

The reason to want the second function is that it buys you something the first cannot: the ability to try actions without paying for them. A policy commits in the real world, where mistakes cost broken grippers and reset time. A world model lets you commit in imagination, roll the consequences forward, and only then act. The three uses of a world model are exactly the three ways to spend that imagined experience.

Three things a world model is for

The first use is **planning**. Given a model, you can search over action sequences, simulate each one, and pick the one whose predicted trajectory scores best, model-predictive control, but with a learned model instead of a hand-derived one. Section 9.3 develops this; the short version is that planning turns a one-step predictor into a decision-maker without ever training a policy.

The second use is **learning a policy inside the model**, training in imagination. This is the oldest idea in the chapter and it has a name: Dyna, from Sutton (1991). The agent collects a little real experience, fits a model to it, and then generates large quantities of synthetic experience by rolling the model forward, treating those synthetic transitions as if they were real and feeding them to an ordinary RL update. Real data is expensive on a robot; model rollouts are nearly free. Dreamer (§9.2) is Dyna scaled up with a deep latent model and a neural policy, and it is the reason the technique matters again.

The third use is **prediction as an end in itself**, sometimes called pure world-model evaluation. Here you never plan and never train a policy in the model; you simply ask how good the model is at forecasting, on the bet that a system that can predict the world has learned something worth reusing. The video-prediction models of §9.4, and LeCun's (2022) argument that prediction is the core of intelligence, live here. A model that can imagine the next two seconds of a kitchen has, in some sense, understood kitchens, whether or not it ever picks anything up.

The canonical example: a controller trained inside a dream

The cleanest illustration is Ha and Schmidhuber's *World Models* (2018), which is worth walking through because every later system in this chapter is a refinement of its three pieces.

The task is a 2-D car-racing game with pixel observations. The system has three components. A **vision model**, a variational autoencoder, compresses each 64×64 frame into a small latent vector z_t, call it roughly 30 numbers instead of roughly 12,000 pixels. A **memory model**, a recurrent network with a mixture-density output head, predicts the next latent: $\hat{P}(z_{t+1} \mid z_t, a_t, h_t)$, where h_t is the RNN's hidden state carrying the history. This is the world model proper; it lives entirely in latent space and never touches a pixel. Finally a tiny **controller**, a single linear layer mapping (z_t, h_t) to an action, is the policy.

The striking result was not the racing score. It was that the controller could be trained *entirely inside the memory model's hallucinated rollouts*, the agent learned to drive in its own dream, then transferred to the real game with little loss. That is the Dyna loop made vivid: the world model became a free simulator, and the policy never needed the expensive environment during its own training.

Two design choices from this example recur for the rest of the chapter. First, **predict in latent space, not pixel space.** Forecasting raw images is wasteful, most of the bits in a frame are irrelevant to what the agent should do, and a model that spends capacity rendering the exact texture of the asphalt has spent it badly. Compress first, predict in the compressed space. Section 9.2's RSSM is a more capable version of exactly this two-stage idea. Second, **the model carries memory.** The world is not Markov in its observations, a single frame does not tell you the car's velocity, so the model maintains a recurrent state that makes the prediction Markov in the *latent*. We return to why that matters in §9.2.

What a world model is not

Three clarifications, because the term gets stretched.

A world model is not necessarily generative in the sense of producing watchable video. The car-racing memory model never renders a frame; it predicts the next 30-number latent and that is enough to plan with. Pixel-perfect video generation, as in Genie (§9.4), is one option on a spectrum, and often an expensive one. What the model must

predict is whatever the downstream use needs: a reward for planning, a latent for policy training, a full frame only if a human or a perception module has to read it.

A world model is not a policy with extra steps. You can derive a policy from a model by planning, but the model itself expresses no preferences. This is precisely the dividing line in the §9.5 debate: the VLAs of Part 4 are policies that learned to act by imitation and hold no explicit model of consequences, while the world-model camp argues that learning consequences first is the more data-efficient and more general bet. Neither side disputes the definitions; they disagree about which function is worth learning.

And a world model is not the same as the simulator you set up in Appendix D. A physics simulator like MuJoCo is a hand-built world model with privileged access to the true state, it knows every joint angle and contact force because you told it the equations. A learned world model has to infer all of that from observations, which is harder and also the entire point: it can model things, like cloth or granular media or a human's next move, that you cannot conveniently write down in a simulator's configuration file.

With the object defined, the obvious question is how to learn it well enough to plan with. The dominant answer for the last several years has been the recurrent state-space model behind Dreamer, and that is where we turn next.

9.2 Latent dynamics: RSSM and Dreamer

Section 9.1 ended on a claim: to plan or to train a policy in imagination, you first need a model good enough to imagine with. Ha and Schmidhuber's car-racing controller showed the shape of the answer, compress pixels to a latent, predict the next latent, act on it, but its memory model was a mixture-density RNN trained separately from the vision model, and it worked cleanly on one toy game. The line of work that turned that sketch into a method used across dozens of tasks is the *recurrent state-space model*, or RSSM, and the family of agents built on it: PlaNet, then Dreamer through its three versions. This section is about what the RSSM is and why its one unusual design choice, splitting the latent state in two, is what makes the whole thing learnable.

Why one latent variable is not enough

Start from the obvious design and watch it fail. You want a recurrent model that carries a latent state s_t forward: given s_t and action a_t, produce s_{t+1}, and from s_t reconstruct the observation o_t and predict the reward. If you make s_t purely deterministic, an ordinary RNN or GRU hidden state, the model cannot represent uncertainty. A robot that has not yet looked inside a drawer does not know whether it holds a stapler or a snake, and a deterministic state forces the model to commit to one guess. Prediction error then punishes it for being decisive about something it could not have known.

If instead you make s_t purely stochastic, sample it fresh from a learned distribution at every step, you get the opposite failure. The model can express uncertainty, but it struggles to *remember*. Any information it wants to carry across ten steps has to survive being resampled ten times, and the noise washes it out. Long-horizon prediction, which is the entire point of a world model you plan with, falls apart.

The RSSM's answer, from PlaNet (Hafner et al. 2019), is to keep both. The latent state is a pair: a **deterministic** part h_t, carried by a GRU, and a **stochastic** part z_t, sampled from a distribution that the network predicts. The deterministic path is a reliable wire down which information flows unchanged across many steps; the stochastic part rides on top of it and captures what the model cannot know for certain. This split is the single idea that makes latent dynamics work, and every Dreamer variant keeps it.

The two distributions: prior and posterior

The RSSM has to do two jobs that pull in different directions, and it handles them with two distributions over the stochastic state.

The **prior** predicts the next stochastic state from the past alone: $\hat{z}_t \sim p_\theta(z_t \mid h_t)$, where $h_t = f_\theta(h_{t-1}, z_{t-1}, a_{t-1})$ is the GRU rolling the recurrent state forward. This is the imagination path. When you dream, roll the model forward without looking at the world, you sample from the prior, because there is no new observation to condition on.

The **posterior** corrects that prediction using the observation that actually arrived: $z_t \sim q_\theta(z_t \mid h_t, o_t)$. This is the perception path, used during training and whenever the agent is grounded in a real frame. The posterior sees the drawer's contents; the prior had to guess.

Training pushes these two together. The model reconstructs the observation and predicts the reward from the state (h_t, z_t), and, the part that matters, it minimizes the KL divergence between the posterior and the prior, $\mathrm{KL}(q_\theta(z_t \mid h_t, o_t) \,\|\, p_\theta(z_t \mid h_t))$. That KL term is the dynamics-learning signal. It says: whatever the observation told you, the model should have been able to predict it from the past. Drive that KL to zero and the prior alone is enough to forecast, which is exactly the condition you need to dream without looking. The whole objective is the variational lower bound (the ELBO from §3.2's probability toolkit), applied to a sequence: reconstruction accuracy minus the KL, summed over the trajectory.

A short, illustrative version of one RSSM step, ignoring batching and the distribution parameterization details:

```
def rssm_step(h_prev, z_prev, a_prev, obs=None):
    # Deterministic recurrence carries memory forward.
    h = gru(h_prev, concat(z_prev, a_prev))

    # Prior: predict next stochastic state from the past alone.
    prior = mlp_prior(h)                # imagination path

    if obs is not None:
        # Posterior: correct the prior using the real observation.
        feat = encoder(obs)
        post = mlp_post(concat(h, feat)) # perception path
        z = post.sample()
        kl = kl_divergence(post, prior)  # dynamics-learning signal
    else:
        z = prior.sample()               # dreaming: no observation
        kl = None
    return h, z, kl
```

The `obs is None` branch is not an afterthought. It is the mode the model runs in when it imagines, and the fact that the same recurrence produces both grounded and imagined states, differing only in whether z comes from the posterior or the prior, is what lets a policy trained in imagination transfer back to the real environment.

PlaNet: plan with the model, skip the policy

PlaNet (Hafner et al. 2019) used the RSSM the way §9.1 called the first use of a world model: pure planning, no learned policy. At each step it ran a sampling-based planner, the cross-entropy method, entirely inside the latent space. It would sample a few hundred candidate action sequences, roll each one forward through the RSSM prior to predict its rewards, keep the best-scoring fraction, refit a distribution to them, and repeat. The first action of the winning sequence gets executed; then the whole search runs again from the new state. This is model-predictive control (§9.3 develops it) with a learned latent model standing in for the physics. On continuous-control tasks from pixels, a cheetah running, a cartpole swinging up, PlaNet matched model-free agents while using a fraction of the environment interaction, because the expensive search happened in imagination.

The catch is cost. Re-planning hundreds of rollouts at every single control step is slow, and the plan is only as good as the search budget. That is the opening Dreamer walked through.

Dreamer: learn the policy inside the dream

Dreamer (Hafner et al. 2020) kept PlaNet's RSSM unchanged and replaced the online planner with a policy learned *from* imagined rollouts, the Dyna idea from §9.1, now with a deep latent model and an actor-critic. Training runs three interleaved loops. First, fit the RSSM to a replay buffer of real experience, exactly as above. Second, imagine: starting from states the model has seen, roll the prior forward for a fixed horizon (fifteen or so steps) to produce a batch of dreamed trajectories, and label each imagined state with a predicted reward and a learned value. Third,

update an **actor** to maximize those imagined returns and a **critic** to estimate them, an ordinary actor-critic (§7.2), except every transition it learns from is synthetic.

The move that makes Dreamer more than Dyna-with-a-neural-net is *how* the actor gets its gradient. Because the RSSM is fully differentiable, Dreamer backpropagates the imagined return through the learned dynamics and straight into the actor's parameters. The policy does not have to discover which actions were good by trial and correlation, the way a policy-gradient method does (§7.2's variance problem); it gets an analytic signal for how a nudge to the action changes the predicted future reward. Real environment interaction is now needed only to keep the world model honest, to collect the experience the RSSM trains on, and not to train the policy at all. On the same continuous-control suite, Dreamer beat PlaNet's scores while dropping the per-step planning cost, because at deployment it just runs the learned actor: one forward pass, no search.

DreamerV2 and V3: discrete latents and hands-off robustness

Two refinements matter for the reader who wants to know why Dreamer is the default the TOC's hands-on exercise reaches for.

DreamerV2 (Hafner et al. 2021) changed the stochastic state from a Gaussian to a set of *categorical* variables, a vector of, say, 32 one-hot codes with 32 classes each, trained through the straight-through gradient estimator. Categorical latents turned out to model the sharp, multimodal uncertainty of visually rich environments far better than a smooth Gaussian, and DreamerV2 became the first agent to reach human-level Atari scores purely from a world model, matching methods that learned directly from the game.

DreamerV3 (Hafner et al. 2023) is the version worth remembering, because its contribution is not a new architecture but *robustness*. A single configuration, one fixed set of hyperparameters, trained working agents across more than 150 tasks spanning continuous control, Atari, and 3-D navigation, with no per-task tuning. It got there through a stack of unglamorous engineering fixes: **symlog** compression of rewards and observations so the same network handles wildly different magnitude scales, **KL balancing** and **free bits** to stop the dynamics KL from either collapsing or dominating, and careful normalization of returns. The headline demonstration was collecting a diamond in Minecraft from scratch, a task with a long, sparse chain of prerequisites that had resisted every prior method, without being shown a human doing it. For a practitioner the point is simpler: DreamerV3 is the world-model agent you can drop onto a new pixel-input task and expect to train without a tuning campaign, which is why it anchors this chapter's exercise.

What the split bought us

Step back and the through-line is one decision made well. Splitting the latent into a deterministic carrier and a stochastic rider let a single model both remember across long horizons and represent what it does not know, and expressing that split as a prior and a posterior turned dynamics learning into a KL term you can optimize by gradient descent. Everything downstream, from PlaNet's planner to DreamerV3's diamond, is built on states produced by that same recurrence. The remaining question is the one PlaNet answered crudely and Dreamer sidestepped by learning a policy: given a trustworthy latent model, how do you actually search it for good actions? That is planning in latent space, and it is next.

9.3 Planning in latent space

Section 9.2 ended with a trustworthy latent model and a pointed question: given a model that can roll the future forward, how do you search it for a good action? Dreamer answered by not searching at all, it trained a policy inside imagination and, at deployment, ran one forward pass. PlaNet answered the other way, with no policy and a fresh search at every control step. This section is about that second answer, because the search machinery is worth understanding on its own. It is the bridge between the learned dynamics of this chapter and the model-predictive control of §4.4, and it is the reason a world model can act well the first time it is asked, before any policy has been trained.

The receding-horizon loop

Planning with a model means solving a small optimization problem, over and over. At the current latent state s_t, you look for the sequence of actions $a_{t:t+H}$ that maximizes predicted return over a horizon of H steps:

$$a^{\star}_{t:t+H} = \arg\max_{a_{t:t+H}} \mathbb{E}\left[\sum_{k=0}^{H-1} \gamma^k \, \hat{R}_\theta(s_{t+k}, a_{t+k})\right], \qquad s_{t+k+1} \sim \hat{P}_\theta(\cdot \mid s_{t+k}, a_{t+k}).$$

Both $\hat{P}_\theta$ and $\hat{R}_\theta$ are the learned model from §9.2, the RSSM prior rolling latents forward and the reward head reading them off. You solve for the whole sequence, then execute only the *first* action $a^{\star}_t$. The world advances one step, you observe the new frame, encode it to a fresh latent through the posterior, and solve the problem again from there. This is model-predictive control (MPC): plan a horizon, commit one step, replan. The name "receding horizon" describes the effect, the planning window slides forward with you, always looking H steps ahead and never further.

Two features of this loop matter before we look at how the arg max gets solved. First, replanning at every step is what makes MPC robust to a mediocre model. A single fifteen-step forecast will drift; but you never trust step fifteen, because by the time you get there you have replanned fourteen times, each time correcting on a real observation. The horizon buys foresight, the replanning buys correction, and the combination tolerates a model that would be useless if you followed its full prediction blindly. Second, the whole loop lives in latent space. Nothing here renders a pixel. The planner scores candidate futures by the rewards the model predicts for latent states, exactly as §9.1 promised; the model must predict whatever the search needs, and here it needs a reward, not a picture.

Sampling-based planners: shoot, weight, repeat

The optimization is nasty. The objective is non-convex, the dynamics are a deep network, and the action sequence is high-dimensional. The methods that work best in practice do not compute gradients at all; they sample.

The crudest version is **random shooting**. Draw a few hundred action sequences at random, roll each through the model, sum the predicted rewards, and keep the single best sequence. It is trivial to implement and embarrassingly parallel, every rollout is independent, so a GPU evaluates the whole batch at once. It is also wasteful, because most random sequences are garbage and the good region of action space never gets sampled densely.

The **cross-entropy method** (CEM), which PlaNet used, fixes that by iterating. Sample a batch of sequences from a Gaussian, roll them all out, keep the top-scoring fraction, the "elites," typically the best ten percent, and refit the Gaussian's mean and variance to just those elites. Sample again from the tightened distribution and repeat a handful of times. The distribution walks toward the high-reward region and narrows around it. PlaNet ran something like a thousand candidate sequences, ten elites, a few refinement iterations, at every control step. The cost is real: hundreds to thousands of model rollouts per action. That cost is exactly what Dreamer eliminated by amortizing the search into a policy, and it is why online planning is reserved for settings where a slow, deliberate decision is worth paying for.

MPPI (model-predictive path integral control; Williams et al. 2017) is the variant you meet most often on real hardware. Instead of a hard cutoff between elites and the rest, it keeps every sampled sequence and weights it by the exponential of its return, $w_i \propto \exp(\eta \, \hat{G}_i)$, then sets the next mean to the reward-weighted average of all samples. A good sequence pulls the mean strongly, a bad one barely at all, and nothing is thrown away. MPPI tends to produce smoother action sequences than CEM's hard selection, which matters for a physical robot whose motors dislike jerky commands, and it underlies a good deal of real-time model-based control on drones and manipulators. Chua et al.'s PETS (2018) is the other reference point here: it paired CEM planning with an *ensemble* of probabilistic dynamics models, and showed that model-based control could match model-free RL on continuous benchmarks with a fraction of the data, the same efficiency argument §9.1 made for world models, demonstrated with explicit planning rather than a learned policy.

The horizon problem, and the value bootstrap that fixes it

A short horizon is cheap but myopic. Plan fifteen steps ahead and the planner is blind to any reward that arrives on step sixteen; a robot that must cross a room to reach a reward will never see the point of the first step if the room

is twenty steps wide. A long horizon sees the reward but is expensive and, worse, compounds model error, each predicted step feeds the next, and small mistakes snowball into a fantasy by the time the horizon runs out.

The fix is one of the more elegant ideas in model-based control: cap the rollout at a short horizon and staple a **learned value function** onto the end. Instead of summing rewards to the end of the episode, you sum H steps of predicted reward and then add $\gamma^H \hat{V}(s_{t+H})$, the value function's estimate of everything that happens after the horizon. The value acts as a learned summary of the far future, so a five-step plan can still account for a reward a hundred steps away, because the value at step five already knows it is coming. This is where planning and learning stop being alternatives and start collaborating: you plan over the near term where the model is reliable, and you trust a learned value for the long term where rollouts would drift.

TD-MPC (Hansen et al. 2022) is the clean instantiation. It learns a latent dynamics model, a reward model, and a value function together, then at each step runs MPPI-style planning over a short latent horizon with the value bootstrapping the tail. Crucially, its model is trained only to predict rewards and values, not to reconstruct observations, which lets the latent space discard everything irrelevant to control. TD-MPC and its successor TD-MPC2 (Hansen et al. 2024) are, as of this writing, among the strongest methods on continuous-control benchmarks, and TD-MPC2 in particular showed a single model handling dozens of tasks across different embodiments, the model-based echo of the generalist ambition that drives the VLAs of Part 4.

Gradient-based planning, and why it is the road less taken

The RSSM is differentiable end to end, so an obvious alternative to sampling is to compute $\partial \hat{G} / \partial a_{t:t+H}$ by backpropagating the predicted return through the rollout, then do gradient ascent on the action sequence directly. This is exactly the mechanism Dreamer used to train its actor (§9.2). For *planning* at test time, though, it is used less than you might expect. Backpropagating through many steps of a recurrent model runs into the same exploding and vanishing gradients that plague any long unrolled network, and the loss surface over raw actions is riddled with poor local optima that a gradient walk falls into and a population of samples escapes. The practical rule of thumb: gradients are excellent for training a policy offline, where you average over many trajectories and can afford to be careful, and sampling is more robust for planning online, where you need one good sequence right now from one state. Some systems split the difference, seed a sampler with a gradient step, or vice versa, but the workhorses in latent planning remain CEM and MPPI.

Tree search over a learned model: MuZero

Everything so far assumed a continuous action space and a shooting-style search. When actions are discrete and the payoff for deep lookahead is large, board games, some strategic tasks, the planner of choice is Monte Carlo tree search, and the landmark system is **MuZero** (Schrittwieser et al. 2020). MuZero learns a latent model in the spirit of this chapter, then plans by building a search tree over latent states: from the current latent, it expands actions, predicts the resulting latent and its value with the model, and uses those predictions to guide which branches to explore deeper. What made it notable is what its model does *not* predict. It never reconstructs the board or the screen; it is trained only to make its reward, value, and policy predictions match reality after each imagined step. The latent is whatever internal state makes planning accurate, nothing more. That is the same lesson TD-MPC learned in the continuous world, a world model for planning should model consequences that matter to the decision, not the appearance of the scene, and it is the design that §9.4 will push in a very different direction by asking a model to predict appearance in full.

The failure mode every planner shares

One warning ties the section together. A planner is an optimizer pointed at a learned model, and an optimizer will find whatever the model rewards, including the model's mistakes. If the dynamics network wrongly predicts that driving the gripper into the table yields high reward, the planner will gleefully propose exactly that, because it is optimizing the model, not the world. This is model exploitation, and it is the model-based analogue of the reward-hacking problem from §5.4. The mitigations are the recurring ones: keep horizons short so errors have less room to compound, bootstrap with a value learned from real returns, and quantify the model's uncertainty so the planner can be penalized for wandering into states the model has never seen, the ensemble in PETS exists for precisely this

reason. A planner is only ever as good as the model is honest, and honesty degrades the further from the data you push it.

Latent planning, then, is a powerful way to turn a one-step predictor into a decision-maker without training a policy, but everything it can do is bounded by how far into the future the model stays trustworthy, and in the visually rich worlds a robot actually inhabits, that horizon is short. The response has been to build models that predict the world in far greater fidelity, all the way to raw video, and that is where we turn next.

9.4 Video-prediction world models (Genie, V-JEPA)

The models in §9.2 and §9.3 all shared a frugal instinct. The RSSM reconstructed observations only because reconstruction was a convenient training signal; MuZero and TD-MPC dropped even that and predicted nothing but reward, value, and policy. The guiding rule was: model only what the decision needs. This section is about the line of work that broke that rule on purpose. Instead of compressing the world down to the few numbers a planner reads off, these models predict the future in its full sensory detail, the next frame of video, pixel by pixel, and bet that a model which can imagine the world in that fidelity has learned something more transferable than any reward-shaped latent. The bet is not obviously right, and the two systems this section centers on, Genie and V-JEPA, disagree sharply about how to make it. That disagreement is the point.

Why predict pixels at all

The case for reconstruction-free latents was airtight for a fixed task with a known reward. It stops being airtight the moment you want one model to serve many tasks you have not specified yet, and it collapses entirely when the data you want to learn from has no reward and no action labels at all.

Consider the data. The internet holds hundreds of thousands of hours of video, people cooking, driving, assembling furniture, playing games, and essentially none of it comes annotated with the actions that produced it or the rewards that mattered. A DreamerV3-style model cannot touch this, because its whole training loop assumes it collected the data itself and therefore knows the actions and rewards. A model that learns purely by predicting the next frame has no such requirement. The supervision is the video itself: show it frames 1 through t, ask it to predict frame $t{+}1$, and the pixels grade the answer. This is the same self-supervised bargain that made language models scale (§8.1), transplanted to observation streams, and it is why "video prediction" and "world model" have become nearly synonymous in the foundation-model era.

The second argument is generality. A reward-specific latent is sharp but narrow, TD-MPC2's state knows exactly what its tasks need and nothing else. A model that must predict the full next frame is forced to represent whatever governs how scenes evolve: that unsupported objects fall, that occluded objects persist, that a pushed cup slides and does not teleport. Those regularities are task-agnostic. If you can learn them once from passive video, you have a substrate that many downstream policies can share, which is exactly the promise Part 4's foundation models chase from the action side.

Genie: controllable worlds from action-free video

Genie (Bruce et al. 2024) is the cleanest demonstration that the bargain pays off. It was trained on over 200,000 hours of publicly available 2-D platformer gameplay video, with no action labels, no rewards, and no knowledge of which button produced which effect. From that alone it learned a *generative interactive environment*: give it a single image, a photograph, a sketch, a frame from a game it never saw, and it turns that image into something you can play, frame by frame, responding to a control input at every step.

The trick that makes this work despite the missing action labels is a **latent action model**. Genie has three learned parts, all trained together on raw frames. A **video tokenizer** compresses each frame to a grid of discrete tokens, in the spirit of §8.4's "what gets tokenized" question, so the rest of the model works over a manageable sequence rather than raw pixels. A **latent action model** looks at a pair of consecutive frames and infers a small discrete code that explains the transition between them, what "action" must have occurred to turn this frame into the next. Crucially it is trained only from the frames, so it invents its own action vocabulary; nobody tells it "this code means jump."

A **dynamics model**, an autoregressive transformer, then predicts the next frame's tokens given the past tokens and one of these latent action codes.

At training time the latent action is inferred from the real next frame, the way §9.2's posterior read the true observation. At play time you withhold the future and instead *supply* the latent action yourself, pick one of the small set of learned codes, and the dynamics model rolls the world forward under your control. The astonishing result is that these self-discovered codes are consistent and interpretable: the same code reliably moves the avatar left across wildly different generated worlds, another makes it jump, and a human can learn to "play" a generated environment within seconds even though the control scheme was never designed. Genie learned a controllable model of how worlds change from watching, without ever being told what control means.

For a robotics reader the implication is direct and is why Genie sits in this chapter rather than a graphics one. Robot teleoperation data is expensive and scarce; passive video of the world is abundant. A world model that can be trained from action-free video and later steered by a small learned action interface is a template for pretraining on the cheap data and grounding on the expensive data, a template later humanoid efforts, and Genie's own successors aimed at 3-D and robotic domains, build on directly.

The objection: pixels are the wrong thing to predict

Genie predicts pixels, and pixels are, in a real sense, a terrible thing to spend a model's capacity on. Most of the bits in a video frame are detail that no decision depends on and no model can predict anyway: the exact texture of gravel, the precise flicker of a shadow, the arrangement of leaves on a tree the robot will never touch. A next-frame objective pours enormous modeling effort into getting those unpredictable details plausible, and a model graded on pixel accuracy is punished for admitting, correctly, that it cannot know them. This is the same complaint §9.3 made about long rollouts drifting into fantasy, sharpened into a critique of the objective itself.

Yann LeCun's position paper (LeCun 2022) turned this complaint into a design principle. The argument: an intelligent agent should predict the *consequences* of actions at an abstract level, not render the sensory future in full. Prediction should happen in a representation space that has already thrown away the unpredictable noise, so the model spends its effort on structure that matters and is not penalized for failing to hallucinate irrelevant detail. This is precisely the lesson MuZero and TD-MPC learned for reward-driven control in §9.3, model consequences, not appearance, promoted to a claim about how any world model should be built. LeCun's name for the resulting architecture is the *Joint Embedding Predictive Architecture*, or JEPA.

V-JEPA: predict the representation, not the frame

V-JEPA (Bardes et al. 2024) is that principle made concrete for video. The setup looks like masked prediction, but with a decisive twist. Take a video, encode it, and mask out a large spatiotemporal chunk, a block of patches across several frames. A generative model like a masked autoencoder would try to reconstruct the missing *pixels*. V-JEPA instead predicts the missing region's *representation*: it runs a target encoder over the full, unmasked video to produce feature vectors for the masked region, then trains a predictor to match those features from the visible context alone. The loss is computed entirely in latent space. No pixel is ever reconstructed.

The mechanism that keeps this from collapsing to the trivial solution (map everything to a constant and predict the constant) is an asymmetry borrowed from self-supervised image learning: the target encoder is not trained by gradient descent but is an exponential moving average of the context encoder, updated slowly, so it provides a stable, non-trivial target the predictor must actually work to hit. What comes out is a frozen video encoder whose features transfer well to downstream recognition and motion-understanding tasks without any fine-tuning of the backbone, strong evidence that predicting in representation space captures the structure of how scenes evolve while spending nothing on rendering.

The contrast with Genie is the whole argument of this section in one line. Genie predicts frames and gets an interactive, playable, visibly impressive world you can steer, at the cost of modeling every pixel. V-JEPA predicts features and gets a compact, transferable understanding of dynamics, at the cost of not being able to show you the future it imagines. One is generative and controllable; the other is abstract and efficient. Which you want depends on whether the downstream job needs to *look* at imagined futures (planning by rolling out visible states, or generating training scenarios) or merely needs a representation that has absorbed the world's dynamics.

Where this meets action

Neither Genie nor V-JEPA is a policy; both are world models in the §9.1 sense, waiting to be put to one of the three uses. Two connections to action are worth naming now and returning to later.

The first is planning by video. If a model can predict future frames conditioned on candidate actions, a planner can, in principle, score those candidates by imagining their visible outcomes and checking whether a goal frame is reached, a pixel-space version of §9.3's shooting search. Early robot systems along these lines conditioned video prediction on a goal image and selected actions whose predicted rollouts best approached it. The obstacle is cost and drift: generating high-fidelity video for hundreds of candidate rollouts at control rate is far more expensive than rolling an RSSM latent, and the pixel objective's blurring compounds over the horizon exactly as §9.3 warned.

The second is pretraining. A representation trained to predict video, whether Genie's tokens or V-JEPA's features, is a candidate visual backbone for a downstream policy, one that has learned about physical dynamics from data no teleoperation rig could ever collect. Whether that substrate actually beats a policy trained end-to-end on action-labeled data is unsettled, and it is the crux of the architecture debate §9.5 takes up head-on: the world-model camp holds that understanding the world must come first and control follows, while the VLA camp of Part 4 bets that pouring action-labeled data through one network learns the relevant dynamics implicitly, no separate world model required.

For now, hold onto the split this section drew. Video-prediction world models predict the observation itself, which unlocks learning from abundant action-free video and produces models you can imagine with in full sensory detail, a real gain in generality over the reward-shaped latents of §9.3. But predicting pixels is expensive and partly wasted, and the JEPA response, predict abstract representations instead, trades away the visible, controllable rollout for efficiency and transfer. Genie and V-JEPA are the two poles of that trade, and the question of which pole the future of embodied intelligence should sit closer to is exactly the debate we turn to next.

9.5 World models vs. VLAs: the architecture debate

The last four sections built up a machine that learns how the world changes and then acts by consulting that machine. Part 4 of this book is about a machine that skips the middle step: it maps observations and a language instruction straight to actions, and never builds an explicit model of dynamics at all. Both machines work. Both have produced systems that control real robots. And their proponents disagree, sometimes heatedly, about which one is a detour and which one is the road. This section lays out the disagreement plainly, because you will spend the rest of the book inside the VLA camp and it is worth knowing what the other camp thinks you are getting wrong.

The dispute is not really about neural network layers. It is about where understanding of physics is supposed to live. One side says a competent agent must first learn a model of how the world behaves, and that control is a comparatively easy search on top of that model. The other side says that understanding is a means, not an end, and that if you have enough data pairing observations with correct actions, a single network will absorb whatever physics it needs implicitly, on the way to predicting the action. Everything else, pixels versus features, planning versus reflex, video pretraining versus teleoperation, is downstream of that split.

The world-model bet

State the strong version, the one Yann LeCun's position paper (LeCun 2022) makes without hedging. A system that only ever learns stimulus-to-response mappings is brittle by construction. It has no way to answer a question it was not trained on, because answering novel questions requires simulating consequences, "if I push here, what happens?", and simulation requires a model. Intelligence, on this view, *is* a predictive world model plus a cheap planner that queries it. The action policy is almost an afterthought; the hard-won asset is the model, and once you have a good one, steering it toward any goal is a search problem you already know how to solve (§9.3).

Three arguments prop this bet up, and each has real weight.

The first is data. Correct-action data is the scarcest thing in robotics, every trajectory costs a human on a teleoperation rig (§6.1). Video of the world is nearly free and effectively infinite. A world model can drink from the free source: Genie learned controllable dynamics from 200,000 hours of unlabeled gameplay (§9.4), and V-JEPA learned

transferable video features with no action labels at all. If most of what a robot needs to know about physics can be learned from passive observation, then the architecture that can *use* passive observation has an enormous head start over one that can only learn from action-labeled demonstrations.

The second is generalization. A model of dynamics is task-agnostic by nature. Gravity, contact, occlusion, and momentum do not change when you switch from stacking cups to folding towels. If those regularities are captured once, in a reusable model, then a new task is a new goal handed to the same planner, no retraining of the physics. A stimulus-response policy, the argument goes, has to relearn the relevant slice of physics inside every new behavior it acquires, which is why such policies are famously data-hungry and famously bad at anything outside their training distribution.

The third is verifiability. A planner that rolls out an explicit model can, at least in principle, be inspected. You can ask what future it predicted, check whether that future was physically plausible, and reject a plan whose imagined rollout is nonsense. A policy that emits an action from an opaque forward pass offers no such handle, a point that returns with force in Chapter 17, where we try to make deployed systems safe. Model-based systems wear their reasoning on the outside.

The VLA counter-bet

Now the other side, which is where the field's momentum and most of its recent results actually sit. The VLA bet is deflationary: an explicit world model is a nice idea that keeps losing to the crude approach of predicting actions directly, and it loses for reasons that are not accidental.

Start with the observation that dooms pixel-space planning in practice. §9.3 and §9.4 already showed the mechanism, model rollouts drift, and a next-frame objective spends most of its capacity rendering detail no decision depends on. Push that further. To act by planning, you must roll your world model forward tens or hundreds of times per decision, at control rate, and every rollout inherits the model's errors compounded over the horizon. The VLA camp's response is: why pay that tax at all? A policy trained by imitation does the equivalent of the whole plan-and-select loop in one forward pass, with no rollout to drift, and it learns to do so from exactly the data, expert trajectories, that already encode good behavior.

Then there is the awkward empirical fact. When you scale a vision-language model that was pretrained on internet images and text and then fine-tune it to output actions, you get a system that generalizes to new objects and new phrasings of a task strikingly well, RT-2 (arXiv:2307.15818) was the demonstration that a VLM's semantic knowledge transfers to control without any explicit dynamics model in sight. OpenVLA (arXiv:2406.09246) and $\pi 0$ (arXiv:2410.24164) doubled down on the recipe and did not add a world model; they added data, better action heads, and bigger backbones. The generalization the world-model camp promised as the payoff of understanding-first turned out to be available, at least in part, straight from language-and-vision pretraining poured through an action objective. Physics, on this reading, gets learned implicitly and well enough, as a side effect of predicting what an expert did.

The counter-bet, sharpened to one sentence: understanding the world is instrumental, and if the instrument you actually need is "output the right action," then training directly for that objective is more efficient than training for the harder, wasteful proxy of "predict the entire future." The world-model camp is solving a superset of the problem, and paying for the extra generality with data, compute, and drift it does not need.

Where the honest uncertainty is

It would be tidy to declare a winner. The tidy version is wrong, and pretending otherwise would misinform you about a live research question.

The VLA camp is winning on today's benchmarks, but today's benchmarks mostly measure short-horizon manipulation from large demonstration sets, precisely the regime where reflexive imitation shines and where long-horizon reasoning, the world model's home turf, barely gets tested. The tasks that would actually separate the two bets, long, novel, multi-step problems that demand planning through states no demonstration covered, are the ones the field is least able to evaluate, a gap Chapter 15 examines directly. Absence of a world-model advantage on cup-stacking is not evidence it will stay absent on assembly.

The two approaches are also converging in ways that blur the fight. DreamerV3 (Hafner et al. 2023) is a world model that trains its policy inside imagined rollouts, a world model with a reflex policy bolted on. The dual-system VLAs of Chapter 14 pair a slow deliberative module with a fast reactive one, which is structurally close to "consult a model, then act." And a growing line of work uses video-prediction models (§9.4) purely as pretraining for the visual backbone of an otherwise ordinary VLA, the world model as ingredient rather than as the whole architecture. The cleanest statement of the two bets is a useful map, but the territory is filling in with hybrids that take something from each.

The deepest open question is the one neither camp can currently settle: does predicting actions from enough data really teach a network the physics it needs, or does it teach a shortcut that happens to work on the training distribution and fails silently off it? If the former, the world model is redundant and the VLA camp is simply right. If the latter, VLAs are accumulating a debt that comes due exactly when we ask them to do something truly new, and the understanding-first camp will have been early rather than wrong. Nobody has the experiment that decides this cleanly, which is why the debate is a debate and not a settled fact.

What to carry forward

For the rest of this book you will follow the VLA road, and you should now understand that this is a choice with a live alternative, not a foregone conclusion. The reasons the VLA path dominates the coming chapters are honest ones: it uses the data we can actually get, it sidesteps the drift that cripples model rollouts, and it inherits generalization from language-vision pretraining for free. The reasons to keep an eye on the world-model camp are equally honest: passive video is a data source VLAs struggle to exploit, planning offers a verifiability that opaque policies lack, and the tasks most likely to expose the difference are the ones we have barely started to test.

Hold both. When Part 4 shows you RT-2, OpenVLA, and $\pi 0$ mapping pixels and language to actions with no explicit model of the world, remember that a serious research tradition thinks this works despite the missing model, not because it is unnecessary, and that the two traditions are already trading parts. We turn next to a summary of the chapter, and then leave world models behind for the action-generation machinery, diffusion and flow, that modern VLAs actually use.

9.6 Summary

Chapter 9 was a detour with a purpose. Every architecture the book builds toward in Part 4 is a policy, a function that reads an observation and a language instruction and emits an action, with no explicit opinion about what that action will do. This chapter spent five sections on the road not taken: models that learn how the world changes and then act by consulting that model. The point was not to talk you out of the VLA path you are about to follow, but to make sure you follow it knowing there is a serious alternative, understanding what it offers and what it costs, and able to recognize the pieces of it that keep reappearing inside VLA systems anyway. This summary collects the load-bearing ideas and marks where the rest of the book reaches back for them.

The four ideas worth carrying forward

A world model is supervised learning on the dynamics, and its value is that it lets you try actions without paying for them. §9.1 gave the definition plainly: a world model approximates the transition function $\hat{P}_\theta(s' \mid s, a)$ and usually the reward $\hat{R}_\theta$, fit by watching transitions roll past, with the labels free because the next state is simply what happened next. A policy answers "what should I do?"; a world model answers "what would happen if I did that?" They point in opposite directions, and the second question is worth asking because it can be asked in imagination, where mistakes cost nothing. That single property generates the three uses the whole chapter organized around, planning over the model, training a policy inside it (the Dyna idea, from Sutton 1991), and prediction as an end in itself, and Ha and Schmidhuber's car-racing controller (2018), trained entirely inside its own hallucinated rollouts, was the vivid proof that the third loop closes.

The RSSM works because it splits the latent in two, and expresses dynamics learning as a KL term. §9.2 was the technical core. A purely deterministic latent cannot represent uncertainty; a purely stochastic one cannot remember across a horizon. The recurrent state-space model keeps both, a deterministic GRU path carrying memory, a stochastic

part riding on top, and this one design choice is what made latent dynamics learnable at scale. The mechanism that trains it is the prior/posterior pair: the posterior corrects a prediction using the observation that arrived, the prior must learn to predict without it, and driving the KL between them toward zero is exactly the condition that lets the model dream without looking. Dreamer then backpropagated imagined returns through the differentiable dynamics into an actor, so the policy gets an analytic signal for how an action changes the predicted future, no environment interaction needed to train it, only to keep the model honest. DreamerV3 (arXiv, Hafner et al. 2023) is the version to remember, because its contribution was robustness: one fixed configuration across 150-plus tasks, up to and including a diamond in Minecraft from scratch.

Planning turns a one-step predictor into a decision-maker, and every planner is only as good as the model is honest. §9.3 developed model-predictive control in latent space, plan a horizon, execute one action, replan, and made the case that the replanning, not the forecast, is what tolerates a mediocre model: you never trust step fifteen because you replan fourteen times before you get there. The workhorses are sampling-based, not gradient-based: CEM (which PlaNet used), MPPI (Williams et al. 2017, the one you meet on real hardware), and PETS's ensemble (Chua et al. 2018). The horizon problem, short is myopic, long compounds error, is solved by stapling a learned value function onto the end of a short rollout, the collaboration between planning and learning that TD-MPC (Hansen et al. 2022) instantiates cleanly and that MuZero (Schrittwieser et al. 2020) mirrors with tree search over a model trained to predict only reward, value, and policy, never appearance. The warning that ties the section together is model exploitation: a planner optimizes the model, so it will happily propose whatever the model's *errors* reward, the model-based cousin of the reward hacking from §5.4.

Video-prediction world models break the frugality rule on purpose, and the pixel-versus-representation split is the live disagreement. §9.4 covered the models that predict the full sensory future rather than a reward-shaped latent, and bet that a model which can imagine the world in that fidelity has learned something more transferable. The motive is data: the internet's hundreds of thousands of hours of action-free, reward-free video are unreachable by a DreamerV3-style loop but are exactly what a next-frame objective can drink from. Genie (Bruce et al. 2024) is the demonstration, a controllable, playable world learned from 200,000 hours of unlabeled gameplay via a self-invented latent action vocabulary. V-JEPA (Bardes et al. 2024) is the counter-position made concrete: predict the missing region's *representation*, not its pixels, following LeCun's (2022) argument that a model should forecast consequences in an abstract space that has already discarded unpredictable detail. Genie is generative and steerable at the cost of modeling every pixel; V-JEPA is compact and transferable at the cost of never being able to show you the future it imagines. Which pole you want depends on whether the downstream job needs to *look* at imagined futures or merely needs a representation that absorbed the world's dynamics.

What you should be able to do now

Four concrete capabilities, in roughly the order the rest of the book will ask for them.

You should be able to *state precisely what a world model is and what it is not.* Not "a model of the world," but the actual object: a learned approximation of the transition and reward functions, distinct from a policy (which expresses preferences a model does not), distinct from a hand-built simulator (which has privileged access to true state), and not necessarily generative in the sense of producing watchable video. §9.1 drew those three boundaries specifically so you can read later claims about "world models" without being misled by the term's elasticity.

You should be able to *explain why the RSSM splits its latent state, and what the prior, posterior, and KL term each do.* Given the Dreamer training loop, you should be able to say which distribution is the imagination path, which is the perception path, why the KL between them is the dynamics-learning signal, and how a differentiable model lets Dreamer give its actor an analytic gradient instead of a policy-gradient estimate. This is the skill that lets §9.5's "world model with a reflex policy bolted on" read as a precise description rather than a slogan.

You should be able to *set up a latent planning loop and name its failure modes.* Given a learned model, you should be able to describe the receding-horizon MPC loop, pick between CEM and MPPI and say why, explain the value bootstrap that rescues a short horizon, and, most importantly, anticipate model exploitation and list the mitigations (short horizons, value bootstrapping, uncertainty penalties). §9.3 built this so that the model-predictive control still running inside many deployed robots (§4.4) reads as continuous with the learned-model version.

You should be able to *place any world model on the pixel-to-representation axis and say what the placement costs.*

Shown a new system, you should be able to ask whether it predicts observations, reward-shaped latents, or abstract features, and reason about the resulting trade in data requirements, generality, transfer, and rollout cost. That axis is the one §9.4 drew between Genie and V-JEPA, and it is the analytical habit the section was training.

Where the chapter has set up the rest of the book

Chapter 9 hands off in three directions, and the first is the most immediate. §9.5's architecture debate is the frame for the entire remainder of the book. You are about to spend Parts 4 and 5 inside the VLA camp, and the chapter's job was to make that a choice you understand rather than a default you inherited, VLAs use the data we can actually get, sidestep the drift that cripples model rollouts, and inherit generalization from language-vision pretraining, while the world-model camp keeps the better claim on passive video, verifiability, and long-horizon reasoning. Hold both when RT-2 (arXiv:2307.15818), OpenVLA (arXiv:2406.09246), and $\pi 0$ (arXiv:2410.24164) appear in Part 4 with no explicit model of the world in sight.

The second handoff is to Chapter 14's dual-system architectures. §9.5 noted that a slow deliberative module paired with a fast reactive one is structurally close to "consult a model, then act," and DreamerV3's own policy-in-imagination is a world model with a reflex policy attached. The convergence the debate section flagged is not rhetorical; Chapter 14 is where the hybrid actually gets built, and a reader who saw the two camps trading parts here will read it as synthesis rather than novelty.

The third handoff is quieter but real: to the video-pretraining thread that resurfaces in Chapter 15's data discussion and Chapter 18's open problems. §9.4 argued that a representation trained to predict video is a candidate visual backbone for an otherwise ordinary VLA, the world model as ingredient rather than as the whole architecture, and whether that substrate beats end-to-end action training is one of the unsettled questions Chapter 18 returns to directly.

What the chapter has not covered

Two omissions are worth naming. The chapter treated world models almost entirely at the level of principle and small-scale demonstration, and said little about the engineering of *deploying* a model-based controller on a real robot at control rate, the latency budgets, the fallback behavior when a plan fails, the monitoring of model uncertainty online. That is deliberate: those concerns belong with the deployment and safety machinery of Chapters 14 and 17, where the model-based system is one case among several, and where the verifiability argument §9.5 made abstractly gets its concrete test.

The chapter also stayed shallow on the generative machinery itself. It said that Genie tokenizes frames and that video-prediction models produce pixels, but it deferred the actual mechanism, how you train a model to generate high-fidelity output step by step, to Chapter 10, where diffusion and flow matching are developed properly. This was the right split: §9.4 needed you to understand *what* a video-prediction model is for and *why* the pixel objective is contentious, not *how* the pixels get generated. That how is next, and it is also the machinery the smooth-control VLAs of Chapters 10 and 13 use for their action heads, which is why Chapter 10 does double duty.

Chapter 9's contribution to the book's overall argument is to define the alternative to the policy-only VLA rigorously enough that the VLA choice reads as a bet with stakes rather than the only option on the table. The three findings to carry forward are that a world model buys the ability to act in imagination and that this generates its every use; that the RSSM's deterministic/stochastic split and its KL-based dynamics objective are what made learned latent models practical; and that the field's central unresolved question, whether predicting actions from enough data really teaches a network the physics it needs, or a shortcut that fails silently off distribution, is exactly the question the world-model camp exists to press. Part 3 now turns to the generative action machinery, diffusion and flow, that the VLAs of Part 4 actually use to turn a chosen intention into smooth motion.

§9.x closes the chapter with a hands-on exercise, training a DreamerV3 agent on a pixel-input control task and probing how imagination horizon and the KL balance affect what it learns, and the full reading list for the chapter.

9.x Hands-on exercise + chapter references

Chapter 9 makes one claim that stays abstract until you watch it fail: a learned world model is only as useful as its rollouts are accurate, and its rollouts degrade the further into the future you ask them to reach. §9.1 defined the model, §9.2 built it, §9.3 searched it, and §9.5 argued about whether the whole apparatus is the future of robotics or a detour. None of that lands until you have trained a model, rolled it forward, and put its prediction next to what actually happened. The headline exercise does exactly that, train a compact Dreamer-style agent on a pixel task and lay its imagined rollout beside the ground-truth frames, which is the canonical Chapter 9 drill and the fastest way to see both the promise and the drift in one picture. The remaining drills probe the edges: how horizon length trades against accuracy, what the RSSM's stochastic branch is actually for, and where the world-model bet of §9.5 stops being a slogan and becomes a measurement.

Budget about three hours. Most of that is wall-clock training time you spend doing something else; the code you write is short. The one real cost is the first exercise's training run, so start it early and read while it runs.

```
pip install torch dm-control
```

You need one pixel-input control task with a low-dimensional action. DeepMind Control's `cartpole-swingup` trains fastest and is the gentlest place to watch a latent rollout track reality; `walker-walk` is more interesting and more punishing if you have the patience. Everything after the install runs offline.

Exercise 9.x.1 — Train a world model, then watch it imagine

Build a small RSSM agent following §9.2. You do not need to reproduce DreamerV3's full stack; a scaled-down version is enough to see the phenomena. The pieces, all named in §9.2: a convolutional encoder from pixels to a feature vector, a recurrent core that carries the deterministic hidden state h_t forward, a stochastic state z_t with a prior (predicted from h_t alone) and a posterior (corrected by the current observation), a decoder that reconstructs the frame, and reward and continue heads. Train it on replay data, either DreamerV3's own actor collecting experience, or, if you want to skip the actor entirely for now, a buffer of random-policy episodes. Optimize the §9.2 loss: reconstruction plus reward prediction plus the KL between prior and posterior. Let the reconstruction loss plateau; on `cartpole-swingup` that is roughly a hundred thousand environment steps, well under an hour on a single GPU and a couple of hours on CPU.

Now the actual experiment, and the one the TOC names. Pick a held-out episode the model never trained on. Feed the model the first few frames so it can build up a posterior state, this is the "warm-up" or context window, then cut off the observations and let it run on the *prior* alone, imagining forward from its own predictions with the recorded actions as input. Decode each imagined latent back to a frame. Lay the imagined filmstrip directly above the ground-truth filmstrip, frame for frame, and look.

The first several frames will be nearly indistinguishable: the pole is where it should be, the cart is where it should be. Then the two strips begin to diverge, and the way they diverge is the lesson. Small dynamical errors compound, a pole angle off by a degree becomes off by ten, then the imagined pole is falling while the real one is upright. Mark the frame where you can first tell the two strips apart without squinting. That frame is your model's honest planning horizon, and it is almost always shorter than you hoped. This is the drift §9.3 warned about and the reason §9.3's planners re-solve every control step instead of trusting one long rollout.

Wall clock: training aside, about thirty minutes for the rollout and the side-by-side plot.

Exercise 9.x.2 — Find the horizon where planning stops helping

This drill turns the drift you just watched into a number that changes a design decision. Take the trained model from 9.x.1 and use it to plan, the §9.3 way: at the current latent state, sample action sequences of length H, score each by predicted return under the model, and execute the first action of the best sequence, the cross-entropy method or MPPI, either is fine. Sweep the planning horizon H across a range, say 1, 5, 15, 30, 50 steps, and record the achieved return in the *real* environment at each setting. Plot horizon against achieved return.

The curve is not monotonically increasing, and that is the point. Short horizons plan myopically and leave return on the table. But past some horizon the achieved return stops climbing and then falls, because the plan is now

scoring itself against imagined frames the model got wrong; you are optimizing an action sequence to look good in a hallucination. The peak of that curve is the horizon where your model's accuracy and your planner's foresight are in balance, and it should sit near the divergence frame you marked in 9.x.1. For the written part, connect the two numbers explicitly and state the consequence: this is why §9.3's receding-horizon loop plans short and often, and why TD-MPC (Hansen et al., 2022) bootstraps a learned value at the end of a short rollout instead of rolling out to the true horizon at all.

Wall clock: about forty-five minutes reusing 9.x.1's model.

Exercise 9.x.3 — Delete the stochastic state and break the drawer

§9.2 argued that splitting the latent into a deterministic h_t and a stochastic z_t is the one design choice that makes the RSSM learnable, and the argument is easy to nod along to and hard to feel. Feel it. Take your 9.x.1 model and ablate the stochastic branch: force z_t to a deterministic function of h_t, so the model can no longer represent "I don't know yet." Retrain and re-run the imagination rollout from 9.x.2.

On a fully observed task like `cartpole-swingup` the ablated model may barely suffer, which is itself worth noting, determinism costs nothing when nothing is hidden. To make the cost appear, introduce partial observability: mask a patch of the frame, or switch to a task where an outcome is genuinely uncertain until revealed (the stapler-or-snake drawer of §9.2 is the mental image). Now the deterministic model is forced to commit to one guess about the hidden variable and is punished by reconstruction loss for guessing wrong, while the full RSSM can spread probability across outcomes and pay less. Report the reconstruction loss of both models on the partially observed task. The gap is the concrete value of the stochastic state, and it is exactly the failure §9.2 predicted rather than one you had to be told about.

Wall clock: about forty minutes if you reuse the training code and only change the latent.

Exercise 9.x.4 — Weigh the world-model bet against a policy

§9.5 laid out the disagreement, learn dynamics and plan on top, or map observations straight to actions and let physics be absorbed implicitly, and insisted you would spend the rest of the book in the second camp. Before you do, price the first camp honestly. Take your trained world model and, without any further environment interaction, train a policy purely inside its imagination: roll the model forward from replayed start states, let an actor-critic learn on the imagined trajectories, and deploy the resulting policy on the real task. This is Dreamer's actual training loop, and the thing to measure is sample efficiency: how many *real* environment steps did the whole pipeline consume to reach a given return, counting only the steps used to fit the model?

Compare that number against a model-free baseline from Chapter 7, the SAC agent from Exercise 7.x.2 is the clean comparison, run to the same return on the same task. The world-model pipeline should reach competence in dramatically fewer real steps, because it manufactures cheap experience in imagination; that efficiency is the §9.5 world-model bet stated as a measurement. Then write the counter-bet in one paragraph: name what the pipeline cost you in return for that efficiency, the model to build and debug, the drift that caps the horizon, the reconstruction objective that spends capacity on pixels a policy would have ignored. That paragraph is the §9.5 debate in your own words, grounded in a number you produced, and it is the frame you should carry into Part 4, where every system drops the world model and bets the other way.

Wall clock: about forty-five minutes if 9.x.1's model is already trained; longer if you train the SAC baseline from scratch.

Chapter 9 reading list

The works below are cited in §9.1–§9.6, grouped by purpose. Full bibliographic entries for everything cited in the book live in Appendix E.2; this is the chapter-local subset.

The idea of a learned simulator

- Sutton, R. S. (1991). "Dyna, an Integrated Architecture for Learning, Planning, and Reacting." *SIGART Bulletin* 2(4). The original argument §9.1 rests on: a learned model lets you interleave real experience with planning against imagined experience.
- Ha, D., & Schmidhuber, J. (2018). "World Models." *NeurIPS 2018*. The compress-pixels-then-predict-latents sketch §9.1 and §9.2 build from; the car-racing controller trained entirely inside its own dream.

Latent dynamics and Dreamer

- Hafner, D., et al. (2019). "Learning Latent Dynamics for Planning from Pixels" (PlaNet). *ICML 2019*. The RSSM §9.2 dissects and the cross-entropy-method planner §9.3 and Exercise 9.x.2 use.
- Hafner, D., et al. (2020). "Dream to Control: Learning Behaviors by Latent Imagination" (Dreamer). *ICLR 2020*. The imagination-trained actor-critic Exercise 9.x.4 reproduces.
- Hafner, D., et al. (2023). "Mastering Diverse Domains through World Models" (DreamerV3). arXiv:2301.04104. The agent Exercise 9.x.1 scales down; §9.2's and §9.5's evidence that one recipe works across many domains.

Planning against a learned model

- Chua, K., et al. (2018). "Deep Reinforcement Learning in a Handful of Trials using Probabilistic Dynamics Models" (PETS). *NeurIPS 2018*. §9.3's ensemble-of-models planner and its sample-efficiency argument.
- Williams, G., et al. (2017). "Information-Theoretic MPC for Model-Based Reinforcement Learning" (MPPI). *ICRA 2017*. The sampling-based controller §9.3 describes and Exercise 9.x.2 sweeps the horizon of.
- Hansen, N., et al. (2022). "Temporal Difference Learning for Model Predictive Control" (TD-MPC). *ICML 2022*. §9.3's short-rollout-plus-learned-value method; the fix for the horizon cliff Exercise 9.x.2 measures.
- Schrittwieser, J., et al. (2020). "Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model" (MuZero). *Nature* 588. §9.3 and §9.4's reconstruction-free model that predicts only reward, value, and policy.

Video-prediction world models

- Bruce, J., et al. (2024). "Genie: Generative Interactive Environments." *ICML 2024*. §9.4's pixel-predicting, action-inferring world model learned from unlabeled video; one pole of the §9.5 debate.
- Bardes, A., et al. (2024). "V-JEPA: Revisiting Feature Prediction for Learning Visual Representations from Video." *TMLR / Meta AI*. §9.4's predict-in-representation-space alternative to Genie's pixel reconstruction.

The architecture debate

- LeCun, Y. (2022). "A Path Towards Autonomous Machine Intelligence." OpenReview position paper. The §9.5 world-model bet in its strongest form: understanding of physics must be learned before control is easy.
- Brohan, A., et al. (2023). "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control." arXiv:2307.15818. The §9.5 counter-bet incarnate, no explicit dynamics model, physics absorbed implicitly; Part 4's opening move.
- Kim, M. J., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246. The open policy-only system §9.5 contrasts with the world-model path.

Chapter summary

Chapter 9 was the road not taken, walked far enough to know what is on it. You can now say precisely what a world model is, a learned approximation of the transition and reward functions, fit by supervised learning on transitions the agent has already seen, whose value is that it lets you try an action in imagination before paying for it in reality. You can build the RSSM that makes this work on pixels, and explain why its split into a deterministic memory and a stochastic "what I don't yet know" state is the choice that keeps it learnable, a claim Exercise 9.x.3 turned from assertion into a reconstruction gap you measured. You can plan against such a model with a receding-horizon search, and you know from Exercise 9.x.2 why that search must be short and frequent: rollout accuracy decays with horizon, and past a point the planner optimizes against its own hallucinations. And you can state the architecture bet of §9.5 in both directions, learn dynamics first, or map observations to actions and let physics take care of itself, and back the

trade-off with the sample-efficiency number Exercise 9.x.4 had you produce. That bet is the hinge of the book. Every system in Part 4 takes the second side of it, and Chapter 10 begins there, with the generative machinery, diffusion and flow models, that lets a policy emit smooth, multimodal actions without ever modeling the world it acts in.

Chapter 10 — Diffusion and flow models for action generation

10.1 A 10-minute introduction to diffusion models

Every action head you have met so far commits to one number. The Gaussian policy of §7.2 outputs a mean and a variance and samples once; the discrete action-token classifier you will meet in Chapter 11 picks the argmax bin. Both assume the right action is a single point, or a single blob around a point. That assumption breaks the moment a task has more than one good answer. Reaching for a mug on a cluttered table, you can swing left around the laptop or right around the coffee pot; both work, and the average of the two drives your hand straight into the laptop. A policy trained to regress the mean of the demonstrations learns exactly that average, and exactly that collision.

Diffusion models are the tool the field reached for to fix this. They were invented for images, the flashy text-to-image systems of the early 2020s are diffusion models, but the property that made them good at images is the property robotics needed: they represent a full distribution over outputs, multi-peaked and all, instead of collapsing it to a mean. Before we can talk about diffusing *actions* in §10.2, you need the mechanism itself. This section is the ten-minute version.

The idea: destroy structure, then learn to rebuild it

Suppose you have a dataset of samples x_0 drawn from some complicated distribution $q(x_0)$ you cannot write down, natural images, or, for us, expert action trajectories. You want a model that can produce new samples from that same distribution. The direct approach, writing a formula for q and sampling from it, is hopeless; the distribution lives in a space with thousands of dimensions and has structure no closed form captures.

Diffusion sidesteps the problem with a trick that sounds like it should not work. Take a real sample and gradually wreck it, adding a little Gaussian noise at a time, until after many steps nothing is left but pure noise. That direction is trivial, adding noise requires no learning. Then train a network to undo one step of the wreckage: given a noisy sample, predict what was added. If the network can reliably strip away a little noise, you can start from pure noise and run it many times, and what falls out the far end is a fresh sample from q. You have turned "sample from an intractable distribution" into "denoise, repeatedly," which is just supervised regression.

The forward, structure-destroying process is fixed and defined for you. Denoising Diffusion Probabilistic Models (Ho, Jain & Abbeel, 2020), the paper that made the recipe practical, defines it as a chain of T steps, each adding Gaussian noise with a small variance β_t:

$$q(x_t \mid x_{t-1}) = \mathcal{N}\big(x_t;\ \sqrt{1-\beta_t}\, x_{t-1},\ \beta_t I\big)\,.$$

A useful accident of Gaussians is that you never have to run this chain step by step. Composing t Gaussian steps gives another Gaussian, so you can jump straight to any noise level in one shot. Writing $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$,

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I).$$

Read that equation as a dial. At $t = 0$ you have the clean sample. As $t \to T$, $\bar{\alpha}_t \to 0$ and the sample dissolves into standard Gaussian noise, all trace of x_0 gone. The whole schedule is chosen so that x_T is indistinguishable from noise you could draw yourself, which is what lets you start sampling from nothing.

Training: predict the noise

The reverse process is the part you learn. In principle you want a network $p_\theta(x_{t-1} \mid x_t)$ that inverts one forward step, and the honest derivation runs through a variational bound on the data log-likelihood, the same machinery behind a VAE. That derivation is worth reading once in the original paper; here is where it lands, which is all you need to implement it.

Because the forward equation tells you exactly which noise ϵ was mixed into x_0 to produce x_t, you can train the network to recover that noise. Sample a clean example, pick a random timestep t, noise the example to level t, and ask the network, call it $\epsilon_\theta(x_t, t)$, to guess the ϵ that was added. The loss is the plainest thing imaginable, mean squared error between the true noise and the predicted noise:

$$\mathcal{L} = \mathbb{E}_{x_0,\, t,\, \epsilon}\left[\left\| \epsilon - \epsilon_\theta(x_t, t) \right\|^2 \right].$$

This is the payoff. All the probabilistic scaffolding collapses to a regression problem of the sort you wrote a training loop for in §3.3. The network takes a noisy input and a timestep, outputs a same-shaped noise estimate, and you minimize squared error. There is no adversary to balance as in a GAN, no reconstruction-versus-KL tension as in a VAE (§3.2 for the KL term). Diffusion training is stable for the same reason it is boring, and that stability is a large part of why it took over.

A note on what the network predicts. Estimating the added noise ϵ is equivalent, up to a rescaling, to estimating the gradient of the log-density $\nabla_x \log q(x_t)$, the *score*. Song et al. (2021) arrived at the same family of models from that direction, framing the whole thing as a stochastic differential equation and calling it score-based generative modeling. Noise prediction and score matching are two dialects for one idea; you will see both names, and it is worth knowing they refer to the same object.

Sampling: run the denoiser backward

To generate, start from $x_T \sim \mathcal{N}(0, I)$ and walk down the ladder. At each step, use ϵ_θ to estimate the noise in the current x_t, subtract a scaled version of it to get a cleaner x_{t-1}, and add back a touch of fresh noise to stay on the distribution the network was trained on. One DDPM sampling step looks like:

```
# eps_theta: trained noise-prediction network
# alpha[t], alpha_bar[t], beta[t]: from the fixed schedule
def ddpm_step(x_t, t):
    eps = eps_theta(x_t, t)
    mean = (x_t - beta[t] / (1 - alpha_bar[t]).sqrt() * eps) / alpha[t].sqrt()
    if t > 0:
        return mean + beta[t].sqrt() * torch.randn_like(x_t)
    return mean  # last step is deterministic
```

Repeat from $t = T$ down to $t = 0$ and the final x_0 is your sample.

The catch is right there in the loop: you call the network once per step, and the original DDPM used $T = 1000$. A thousand forward passes to produce one sample is fine for generating a wallpaper offline; it is a disaster for a robot that needs a new action every 30 milliseconds. Most of the engineering since 2020 has been about buying back those steps, DDIM-style deterministic samplers that skip most of the ladder, distillation into a handful of steps, and eventually the flow-matching reformulation of §10.3 that changes the objective so that fewer steps are needed by construction. Hold onto this tension between sample quality and inference latency; it is the axis §10.4 organizes the whole chapter around.

Conditioning: generating the *right* sample

So far the model generates unconditioned samples, some plausible x_0, with no say over which one. A robot needs the opposite: given what the camera sees right now, produce an action that fits *this* situation, not a random draw from every action in the dataset. The fix is to feed the condition into the denoiser. Let o be the current observation (an image embedding, a proprioceptive state, a language instruction, or all three) and train $\epsilon_\theta(x_t, t, o)$ to predict noise given that context. The loss is unchanged; you just widen the network's input. At sampling time you hold o fixed and denoise as before, and the walk down the ladder is now steered toward actions consistent with the observation.

The same conditioning hook is what lets you dial *how strongly* the model obeys the condition. Classifier-free guidance, the standard trick borrowed from text-to-image work, trains the network to run both with and without o and then extrapolates between the two predictions at sampling time, sharpening obedience to the instruction. It matters less for manipulation than it does for image generation, but it is the reason a language-conditioned diffusion policy can be pushed to follow the prompt more literally when you need it to. We flag it here and return to it when it earns its keep.

Why any of this helps a robot

Return to the mug and the laptop. A diffusion model does not learn the average of the two ways around the obstacle; it learns the distribution that has probability mass on the left path and on the right path and almost none in the collision zone between them. Sample it once and you get a committed left-swing or a committed right-swing, never the averaged smear. That is the multimodality property, and it is the single reason diffusion displaced mean-squared-error regression as the default action head for imitation learning.

There is a second reason, quieter but just as important for control. Nothing in the recipe cares whether x_0 is an image, a single action, or a chunk of sixteen consecutive actions stacked into one vector. If you define x_0 to be a short trajectory, the model learns to generate whole coherent trajectories at once, the smooth, temporally consistent motion that Diffusion Policy (Chi et al., 2023) gets and that the ACT architecture reaches by a related route. We will make that concrete in §10.2, where the abstract x_0 finally becomes a sequence of gripper poses.

None of this comes free. You have already seen the cost: a diffusion head trades one cheap forward pass for many, and a robot's control loop has no patience. Whether the multimodality is worth the latency depends on the task, and answering that question well is a skill this chapter is trying to give you.

With the mechanism in hand, corrupt with a fixed noise schedule, train a network to predict the noise, sample by denoising from pure noise, we can stop talking about images and start diffusing actions.

10.2 Diffusion Policy and ACT

At the end of §10.1 the sample x_0 was still an abstraction, some vector the denoiser learns to produce. This section fixes what that vector is. In both systems below, x_0 is a short stretch of future robot actions: a sequence of end-effector poses or joint targets, maybe sixteen of them, stacked into one object the model generates in a single shot. That one design decision, predict a chunk of the future, not the next step, is what these two papers share, and it turns out to matter more than the generative machinery bolted on top of it.

Diffusion Policy (Chi et al., 2023) and ACT (Zhao et al., 2023) landed within weeks of each other in 2023 and became the two reference points for modern imitation learning on real hardware. They reach similar places by different roads: one diffuses the action chunk, the other draws it from a conditional VAE. Reading them side by side is the fastest way to see which parts of the recipe are essential and which are interchangeable.

Diffusion Policy: the head from §10.1, pointed at actions

Take the conditional denoiser from the end of §10.1 and make three concrete commitments. First, the thing being generated is not one action but a horizon of them, Diffusion Policy predicts a block of T_p future actions at once, typically around sixteen. Second, the condition o is the recent observation history: the last two camera frames, encoded by a vision backbone, plus proprioception. Third, the denoiser is a network sized for sequences, the paper

offers both a 1-D convolutional U-Net over the time axis and a transformer variant, and reports the U-Net as the more forgiving default.

Training is exactly the loss from §10.1, no additions. Sample a chunk of expert actions from the demonstration set, noise it to a random level t, and regress the noise given the observation:

$$\mathcal{L} = \mathbb{E}_{\mathbf{a}_0,\, t,\, \epsilon}\left[\, \left\| \, \epsilon - \epsilon_\theta(\mathbf{a}_t,\ t,\ o) \, \right\|^2 \,\right],$$

where $\mathbf{a}_0$ is now the clean action chunk and $\mathbf{a}_t$ its noised version. Nothing here knows or cares that the target is a trajectory rather than a picture. That is the point of §10.1's second observation paying off: the recipe is indifferent to what x_0 means.

Two smaller choices in the paper are worth flagging because they travel with the method. The observation conditions the denoiser through FiLM, feature-wise linear modulation, where the encoded observation produces per-channel scale and shift parameters that reweight the denoiser's activations, rather than being concatenated to the input. And the action space matters: Diffusion Policy reports that predicting end-effector *positions* works better than predicting velocities, because position targets keep the generated chunk anchored to where the gripper should be rather than accumulating drift through integration. Neither point is exotic, but both are the kind of detail that decides whether a reimplementation trains at all.

The interesting engineering is at deployment, and it goes by **receding-horizon control**. The policy predicts T_p actions but you do not execute all of them. You run the first T_a, say the first eight of sixteen, then throw the rest away, take a fresh observation, and generate a new chunk. Predicting a long horizon keeps the motion committed and smooth; executing only a prefix keeps the robot reactive to whatever changed while it was moving. The gap between T_p and T_a is a knob: predict far, act briefly, replan often.

Why the chunk fixes what §10.1 promised

Two failure modes from earlier chapters dissolve here, and it is worth naming both.

The first is multimodality, the mug-and-laptop problem from §10.1. On the PushT benchmark, a 2-D task where the robot must nudge a T-shaped block into a target pose using a circular pusher, there are often two equally good ways to approach the block, from one side or the other. A network that regresses the mean action steers straight down the middle and stalls against the block's flat edge. Diffusion Policy, because it samples from a distribution with mass on both approaches, commits to one side and completes the push. PushT is deliberately small and deliberately bimodal; it exists to make this exact difference visible, and Diffusion Policy's jump in success rate on it over mean-regression baselines is the headline result of the paper.

The second is compounding error, the reason §6.3 had to introduce DAgger. A single-step policy that drifts slightly off the demonstrated states keeps drifting, because each small error moves it toward states it never trained on. Predicting a coherent chunk of actions at once does not eliminate the problem, but it dampens it: the model reasons over a window rather than reacting frame by frame, so short-term noise averages out inside the chunk instead of accumulating across steps. Diffusion Policy buys some of DAgger's robustness without DAgger's online data collection, which is a large part of why practitioners reached for it.

The cost is the one §10.1 flagged. Each action chunk requires running the denoiser many times, and the original DDPM sampler wanted a hundred-plus steps. Diffusion Policy leans on DDIM-style samplers to cut that to roughly ten network calls per chunk, and receding-horizon execution means you only pay that cost once every T_a steps rather than every step. Even so, the latency question does not go away, and §10.4 is where we account for it honestly.

ACT: same chunk, a different generator

ACT, Action Chunking with Transformers, arrives at the chunk from the other direction. Zhao et al. (2023) built it as the learning half of ALOHA, a low-cost bimanual teleoperation rig assembled from two pairs of leader-follower arms. The hardware is the point of the paper as much as the algorithm: it made fine-grained two-handed tasks, threading a zip tie, slotting a battery into a slot with millimeters of clearance, opening a translucent condiment cup, collectable by an untrained human operator and learnable from a few tens of demonstrations.

ACT keeps the action chunk but replaces the diffusion head with a **conditional variational autoencoder** wrapped around a transformer. The structure is an encoder-decoder transformer, the same shape as §8.1, that maps the current observation to a chunk of k future actions in one forward pass, no iterative denoising. The "variational" part is a small latent variable z: at training time an encoder compresses the demonstrated action chunk into z, and the decoder reconstructs the chunk from z plus the observation. The training loss is the standard CVAE objective, a reconstruction term plus the KL penalty on z you met in §3.2:

$$\mathcal{L}_{\text{ACT}} = \underbrace{\|\mathbf{a} - \hat{\mathbf{a}}\|_1}_{\text{reconstruction}} + \beta \underbrace{D_{\text{KL}}(q(z \mid \mathbf{a}, o) \,\|\, \mathcal{N}(0, I))}_{\text{latent regularizer}}.$$

At deployment the encoder is discarded. You sample z from the prior, in practice ACT often just sets $z = 0$, its mean, feed it with the observation to the decoder, and read off a chunk of actions in a single pass. Where Diffusion Policy handles multimodality by sampling a noisy denoising trajectory, ACT parks it in z: different draws of the latent correspond to different coherent ways of doing the task. In practice ACT uses the latent lightly and gets most of its mileage from chunking itself.

ACT's counterpart to receding-horizon control is **temporal ensembling**. Rather than execute a prefix and discard the rest, ACT queries the policy at every timestep, producing overlapping chunks, and averages the multiple predictions that now exist for each moment in time (weighted so recent predictions count more). The overlapping votes smooth the executed trajectory and suppress the jitter a single chunk boundary can introduce. It is a different answer to the same question Diffusion Policy answers with receding horizons: how do you turn a stream of overlapping chunk predictions into one smooth stream of commands.

Reading them side by side

Strip away the branding and the two systems line up cleanly.

Both predict a chunk of future actions rather than a single next action, and both treat that as the load-bearing idea. The chunk is what gives them smooth motion and partial immunity to compounding error, and it is the design choice that survived into every VLA action head that came after; you will see it again in Octo's diffusion head (§12.3) and in $\pi 0$'s flow-matching head (Chapter 13).

Where they differ is the generator strapped to the chunk. Diffusion Policy denoises, paying for expressiveness with a handful of sequential network calls. ACT decodes a CVAE in one pass, paying for that speed with a generator that captures multimodality less sharply, a Gaussian latent is a blunter instrument than a full denoising chain. Neither choice is universally right, which is precisely the trade-off §10.4 lays out: how much multimodality does your task actually need, and how many milliseconds can your control loop spare to get it.

A concrete way to feel the difference: on a task with one clearly correct motion and tight timing, inserting a peg where there is only one sane approach angle, ACT's single-pass speed is close to free and its weaker multimodality costs nothing, because there was only one mode to capture. On PushT, where two approaches genuinely compete, Diffusion Policy's sharper distribution earns its extra compute. Matching the head to the task is the skill; the two papers are the anchor examples at opposite ends of the spectrum.

One historical note worth carrying forward. ACT is not, strictly, a diffusion model, it belongs in this chapter because it solves the same problem with the same chunking insight, and because the field consistently benchmarks the two against each other. When someone says "diffusion versus ACT," they mean "iterative generative head versus single-pass generative head," and the real subject of the comparison is the chunk they have in common.

Both systems still generate their chunk from scratch each time, whether by ten denoising steps or one CVAE pass. The next section asks whether the ten steps can be collapsed toward one without giving up the distribution, which is the question flow matching was built to answer.

10.3 Flow matching and rectified flow for action

§10.2 left both action heads generating their chunk from scratch every time, ten denoising steps for Diffusion Policy, one CVAE pass for ACT. The ten steps are the expensive part, and §10.1 already named the reason: the DDPM sampler walks down a long ladder, one network call per rung, because the reverse process it learned is a wandering, curved path from noise back to data. Flow matching asks a sharper question. What if you trained the path to be straight? A straight path from noise to a sample can, in the limit, be traversed in a single step, because you already know the direction, it never changes. This section is how that idea works and why π0 built a foundation model on it.

From a noise schedule to a velocity field

Start by throwing out the noise schedule. In §10.1 the forward process was a fixed chain of Gaussian corruptions indexed by a discrete step $t \in \{1, \dots, T\}$, and the whole apparatus of β_t and $\bar{\alpha}_t$ existed to describe how much of the original sample survived at each rung. Flow matching replaces that with something simpler to picture: a continuous path in time, $t \in [0, 1]$, that carries a point from pure noise at $t = 0$ to a data sample at $t = 1$.

The cleanest such path is a straight line. Draw a noise sample $x_0 \sim \mathcal{N}(0, I)$ and a data sample x_1 from your dataset, and define the point at time t as the linear interpolation between them:

$$x_t = (1 - t)\, x_0 + t\, x_1.$$

At $t = 0$ you are sitting on the noise sample; at $t = 1$ on the data sample; in between you slide along the segment connecting them. Now ask the obvious calculus question, how fast is the point moving, and in what direction? Differentiate:

$$\frac{dx_t}{dt} = x_1 - x_0.$$

The velocity is constant. Along this particular straight segment the point moves in the same direction at the same speed the whole way. That constant vector, $x_1 - x_0$, is the target the network learns to predict, and its constancy is the entire source of flow matching's speed advantage. Where diffusion learned to nudge a sample one noisy notch toward the data manifold, flow matching learns the *velocity of a flow* that transports the whole noise distribution onto the whole data distribution.

Training: regress the velocity

The training loop is, if anything, blunter than diffusion's. Pick a data sample x_1, draw a noise sample x_0, pick a random time $t \in [0, 1]$, form the interpolated point x_t, and train a network $v_\theta(x_t, t)$ to output the velocity that carried it there:

$$\mathcal{L} = \mathbb{E}_{x_1, x_0, t}\left[\left\| v_\theta(x_t,\ t) \ -\ (x_1 - x_0) \right\|^2 \right].$$

This is the conditional flow matching objective of Lipman et al. (2023), specialized to straight-line paths, the same specialization Liu et al. (2023) arrived at independently and named rectified flow. Compare it line for line with the diffusion loss from §10.1. Both are mean squared error. Both feed the network a corrupted point and a time index. The only change is the regression target: diffusion predicts the *noise* ϵ that was added, flow matching predicts the *velocity* $x_1 - x_0$ that points from noise toward data. That is a small edit to the code and a large change in what the model represents.

```
# v_theta: velocity network, same shape in and out as the sample
def flow_matching_loss(x1, v_theta):
    x0 = torch.randn_like(x1)                # noise endpoint
    t  = torch.rand(x1.shape[0], 1)          # random time in [0, 1]
    xt = (1 - t) * x0 + t * x1               # point on the segment
```

```
    target = x1 - x0                          # constant velocity
    return ((v_theta(xt, t) - target) ** 2).mean()
```

Two things are worth pausing on. First, there is no schedule to tune, no β_t curve, no choice of variance-preserving versus variance-exploding parameterization. The interpolation is linear and that is the end of it, which removes a whole category of the fiddly hyperparameters diffusion practitioners argue about. Second, the loss regresses a *conditional* velocity, the exact vector for this one (x_0, x_1) pair, but what the network converges to is the *marginal* velocity averaged over every pair that could have produced x_t. That averaging is the subtle part, and it is exactly where rectified flow earns its name, so hold the thought.

Sampling: integrate an ODE

Generation runs the flow forward in time. Start at a fresh noise sample $x_0 \sim \mathcal{N}(0, I)$ and integrate the ordinary differential equation $\frac{dx}{dt} = v_\theta(x, t)$ from $t = 0$ to $t = 1$. The crudest integrator, forward Euler, is a loop that reads almost like the DDPM step from §10.1 with the stochastic term deleted:

```
def sample(v_theta, shape, n_steps):
    x = torch.randn(shape)          # start at noise, t = 0
    dt = 1.0 / n_steps
    for i in range(n_steps):
        t = i * dt
        x = x + v_theta(x, t) * dt  # step along the velocity
    return x                        # arrives at data, t = 1
```

The number of steps `n_steps` is now an honest dial you turn at inference, not a property baked into a training schedule. Turn it up and the Euler integration hugs the true trajectory more closely; turn it down and you take fewer, larger strides and pay for it in accuracy. This is the knob §10.4 will weigh against control-loop latency.

Here is the catch, and it is the reason "just use a straight line" is not the whole story. The individual training segments are straight, but the *marginal* field the network learns is generally not, because many different straight segments cross the same point x_t heading in different directions, and the network can only output their average there. Follow that averaged field with Euler and your actual trajectory bends. A curved trajectory needs many small steps to integrate accurately, you are back to diffusion's problem, just in nicer notation. Naive flow matching with a linear path buys you a cleaner objective, but not, by itself, one-step sampling.

Rectified flow: straighten the path

Rectified flow (Liu et al., 2023) is the procedure that removes the curvature, and the trick is almost impudent. Train a first flow model the usual way. Then generate a batch of samples with it, and keep the *pairing*: each generated x_1 came from a specific starting noise x_0. Now retrain a fresh flow model on these matched pairs instead of on random noise-data pairings. Because the pairs came from the model's own transport, the straight segments between them cross each other far less, so the marginal field the second model learns is much closer to actually straight. Repeat once more if you want it straighter. Each pass is called a *reflow*.

The payoff is that a well-rectified flow can be integrated in a handful of Euler steps, sometimes even one, with little loss in sample quality, because a straight trajectory is exactly the case where one big Euler step lands in the right place. This is the concrete sense in which flow matching "needs fewer steps by construction," the promise §10.1 made when it first pointed forward to this section. Diffusion gets to few-step sampling by distilling a slow teacher into a fast student after the fact; rectified flow builds the straightness into the training objective and its reflow procedure. Both end up fast; flow matching gets there with less machinery.

It is worth being precise about what reflow costs, because §10.4 will hold it against the alternatives. Each reflow pass means generating a dataset from the current model and training another one, real compute, paid once, offline. In exchange you move expense out of the inference loop, where a robot cannot afford it, and into training, where you can. That is usually the trade you want on hardware.

Flow matching for action

Everything so far has been generic, the sample x_1 could be an image. Point it at robot actions the same way §10.2 pointed diffusion at them: let x_1 be a chunk of future actions, a stack of, say, fifty joint or end-effector targets, and condition the velocity network on the current observation. The training loss gains a conditioning argument and nothing else changes:

$$\mathcal{L} = \mathbb{E}\left[\, \left\| \, v_\theta(x_t,\ t,\ o) \ -\ (x_1 - x_0) \, \right\|^2 \,\right],$$

with o the encoded cameras, proprioception, and language instruction. At deployment you draw noise the shape of an action chunk, integrate the observation-conditioned ODE for a few steps, and read off a chunk of continuous actions, no discretization, no binning.

The reference implementation is **π0** (Black et al., 2024, arXiv:2410.24164), the first foundation-scale VLA to use a flow-matching action head, and the model Chapter 13 dissects in full. The one-sentence version: π0 takes a pretrained vision-language model, attaches a flow-matching head that generates action chunks at high frequency, and trains the whole thing on a large cross-embodiment demonstration mixture. The choice of flow matching over diffusion is not incidental. π0 targets smooth, dexterous control at up to 50 Hz, folding laundry, bussing a table, and at that rate a fifty-step diffusion sampler per chunk is untenable. A flow head that produces a good chunk in around ten integration steps, on a continuous action space that never has to be quantized into tokens, is what makes the control frequency reachable. The contrast with the discrete action-token approach of RT-2, which we reach in Chapter 12, is exactly the continuous-versus-discrete axis this chapter keeps circling; π0 is the continuous pole, and flow matching is how it holds that pole cheaply.

Two caveats keep the picture honest. Flow matching does not invent a new kind of multimodality, a flow, like a diffusion model, represents a full distribution over action chunks and samples committed modes from it, so the mug-and-laptop argument of §10.1 carries over unchanged; what flow matching changes is the *cost* of drawing that sample, not the expressiveness of the draw. And the straightening is not free lunch: push toward genuinely one-step generation on a hard, multimodal action distribution and quality does eventually degrade, which is why deployed systems like π0 sit at a few steps rather than one. The right number of steps is a task-dependent choice, and choosing it well is precisely the trade-off §10.4 takes up next.

10.4 Trade-offs: latency, multimodality, smoothness

The last three sections handed you three action heads and, each time, punted the hard question to this one. ACT decodes a chunk in one pass but represents multimodality with a blunt Gaussian latent. Diffusion Policy samples a sharp distribution but pays for it in denoising steps. Flow matching straightens the sampling path to buy back most of those steps but still cannot conjure a mode the training data never showed it. Every one of those sentences is a trade. This section names the three axes the trades live on, latency, multimodality, smoothness, shows why you cannot maximize all three at once, and turns that into a way to pick a head for a task instead of picking one by habit.

The reason this deserves its own section rather than a table is that the three axes are coupled. Turn the latency knob and multimodality moves. Ask for more multimodality and smoothness gets harder to guarantee. Nobody ships the corner of the cube where all three are perfect, because that corner does not exist. What you ship is a defensible point in the interior, chosen for the task in front of you.

Latency: count the network calls

Start with the axis a robot feels most directly. A control loop has a budget, the interval between when a fresh observation arrives and when an action must be on the wire. On the LIBERO-style setups from Chapter 2 that budget is tens of milliseconds; on the 50 Hz dexterous tasks π0 targets it is 20 milliseconds, full stop. Blow the budget and the robot stutters, or worse, acts on stale sensing.

The dominant term in a generative head's latency is the number of function evaluations, how many times you run the network to produce one output. The field abbreviates it NFE, and it is the honest currency for comparing these heads:

- **ACT: 1 NFE.** One forward pass through the CVAE decoder yields the whole chunk. This is the floor; you cannot do fewer than one.
- **Diffusion Policy: roughly 10 NFEs.** DDPM wanted a hundred-plus steps; DDIM-style samplers (§10.2) cut it to about ten without much quality loss. Each step is a full denoiser call.
- **Flow matching: 1 to 10 NFEs.** Naive flow matching lands near diffusion because its marginal field is curved (§10.3). Rectified flow pushes toward a handful of steps, and a well-reflowed model can approach one, $\pi 0$ sits at a small number rather than one because the last step of straightening costs quality on hard action distributions.

Two multipliers turn NFE into wall-clock time, and both cut in your favor. The first is per-call cost: a call to $\pi 0$'s billion-parameter backbone is not remotely the same price as a call to Diffusion Policy's compact U-Net, so NFE only compares heads of similar size, across sizes you multiply by the per-call latency. The second is the one §10.2 already banked: **receding-horizon execution amortizes the generation cost over a whole chunk**. If you generate sixteen actions and execute eight before replanning, you pay the head's latency once every eight control ticks, not once per tick. A ten-NFE head that would be hopeless at 50 Hz per step becomes affordable when its cost is spread across an eight-step prefix. Chunking is not only a smoothness trick; it is the reason a multi-step sampler can meet a real-time budget at all.

Multimodality: how sharply can the head disagree with itself

The second axis is expressiveness, how faithfully the head represents the fact that a task often has several correct actions. §10.1 made the case with the mug-and-laptop image and §10.2 made it measurable on PushT, the bimodal block-pushing benchmark where a mean-regression policy stalls against the block's flat edge because the average of "go left" and "go right" is "drive straight into it." Here we rank the heads by how sharply they hold competing modes apart.

At the bottom is plain regression, an L2 or L1 head that predicts one action. It has no multimodality at all; it collapses every mode to their mean, and on a bimodal task that mean is often the one action that is wrong. This is the baseline the whole chapter exists to beat, and it is worth remembering it is still the right choice when the task genuinely has one mode.

In the middle sits ACT's CVAE. A Gaussian latent z can carry more than one mode, but a unimodal prior smears nearby modes together, and, as §10.2 noted, ACT often runs with z near its mean and leans on chunking for most of its performance. Call it multimodality on paper that is used lightly in practice.

At the top are diffusion and flow matching. Both represent a full distribution over action chunks and sample a single committed mode from it, so both clear PushT cleanly. The catch is the coupling to the first axis: **the sharpness of that distribution is what you spend NFEs to recover.** Collapse a diffusion sampler to one or two steps, or push rectified flow toward genuine one-step generation, and the sampled distribution starts to blur back toward its mean, you are trading multimodality for latency directly, and past a task-dependent point the head stops committing to a mode and starts averaging again. This is why "just use one step" is a claim to check against your task's mode structure, not a free upgrade.

Smoothness: the axis that lives between chunks

The third axis is the one demos undersell and hardware punishes: whether the executed trajectory is smooth or jerks. Jerk is not a cosmetic complaint. It shakes cameras, overshoots contact-rich targets, and on a real arm it is the difference between seating a battery and crushing it.

Chunking is the main lever, and §10.2 already explained the mechanism: predicting a coherent block of future actions keeps the motion committed over a window, so short-term noise averages out inside the chunk instead of accumulating across single-step reactions. That is also the smoothness half of chunking's answer to compounding error (§6.3). But chunking creates its own seam, the boundary where one chunk ends and the next begins, which is exactly where a discontinuity can appear if the new chunk disagrees with where the old one left the arm. The two systems handle the seam differently, and the difference is instructive:

- **Receding horizon (Diffusion Policy)** executes a prefix and replans, accepting a potential discontinuity at each replan boundary and keeping it small by predicting far and acting briefly.

- **Temporal ensembling (ACT)** queries every timestep, producing overlapping chunks, and averages the several predictions that now exist for each moment. The overlap smooths the seam by construction, at the cost of running the head every tick, which, since ACT is 1 NFE, it can afford. A ten-NFE head cannot run every tick, which is why Diffusion Policy reaches for receding horizon instead. The smoothness strategy and the latency budget are, again, the same decision.

One more smoothness lever is upstream of the generator entirely: the action space. §10.2 noted Diffusion Policy predicts end-effector positions rather than velocities, because absolute position targets stay anchored while integrated velocities drift. Continuous heads help here too, π0's flow head emits continuous action chunks with no quantization, whereas a discrete action-token head (RT-2, reached in Chapter 12) must round every action to a bin, and bin edges are small discontinuities baked into the representation before the controller ever sees them. Smoothness is partly won or lost before you choose a sampler.

Choosing a head: match the corner to the task

Put the three axes together and the choice stops being about which paper is newest. Ask three questions about the task, in order:

Is it multimodal? If there is one sane way to do the motion, insert this peg at the one approach angle the geometry allows, you need no multimodality, and a regression head or ACT with a near-mean latent is not just adequate, it is the right call: fewer moving parts, one NFE, nothing to tune. Spending a ten-step diffusion sampler on a unimodal task is paying for a distribution with one point in it.

How tight is the control loop? If the loop is slow enough that ten NFEs fit inside the budget after receding-horizon amortization, Diffusion Policy's sharp distribution is free to use. If the loop is tight and the model is large, the 50 Hz, billion-parameter regime of dexterous manipulation, you cannot afford ten calls to a big backbone, and flow matching's few-step sampling is what makes the frequency reachable at all. This is precisely why π0 chose a flow head over a diffusion head (§10.3): not fashion, arithmetic.

How much does smoothness matter, and where are the seams? Contact-rich and high-frequency tasks want the tightest seam handling, temporal ensembling if you can afford per-tick inference, a continuous action space regardless. Coarser pick-and-place tolerates receding-horizon seams fine.

Two worked cases make the framework concrete. Peg insertion with a fixed approach: unimodal, moderately tight, moderate smoothness, ACT is close to ideal, and its single pass costs nothing you needed. PushT: genuinely bimodal, loop timing forgiving, Diffusion Policy earns its extra compute because the multimodality is the whole task. Folding laundry at 50 Hz on a foundation-scale model: multimodal, brutally tight loop, smoothness critical, flow matching on a continuous action space, which is the π0 corner. Same three axes, three different points in the cube, each defensible for its task and indefensible for the others.

The honest summary is that there is no dominant head, only heads matched well or badly to tasks. The three axes are coupled tightly enough that improving one usually spends another, so "best" is a property of the pairing, not of the algorithm. What travels across all of them is the chunk, every head here predicts a block of future actions, and that one shared decision is doing more work than the generator bolted on top of it. Section 10.5 takes this framework to the models that actually ship and asks which corner each one chose: how RT-2, OpenVLA, π0, and Helix each answered these three questions, and what their answers reveal about where foundation action models are heading.

10.5 Action-head choices in modern VLAs

The previous section gave you three axes and promised the shipping models would show up here to be measured against them. Now we collect on that. Four systems, RT-2, OpenVLA, π0, and Helix, each bolted a different action head onto a vision-language backbone, and each choice is a legible answer to the three questions from §10.4: how multimodal is the head, how many network calls does it cost, and what does it do about smoothness. Read in order, the four also trace an arc: the field started by reusing the machinery it already had, discovered where that machinery broke, and then built heads specifically for control. This section is that arc, told through the head each model chose.

We are looking at the heads only. The backbones, training data, and scaling stories belong to Part 4, RT-2 and OpenVLA return in Chapter 12, $\pi 0$ in Chapter 13, Helix in Chapter 14. Here the question is narrower and sharper: given a VLM that can already see and read, how do you get an action out of it, and what does that decision cost?

RT-2: actions as words

RT-2's move was the one a language modeler would make first. If the backbone is a VLM that emits text tokens, then make the action a string of text tokens. RT-2 discretizes each continuous action dimension into 256 bins and assigns each bin an integer, then reuses tokens from the model's existing vocabulary to spell out those integers (arXiv:2307.15818). An action becomes a short sentence, seven numbers for a 6-DoF arm plus a gripper flag, and the model predicts it the same way it predicts any other tokens, one at a time, autoregressively.

The appeal is that nothing about the architecture changes. The same transformer that learned "a photo of a cat" from web data now emits "1 128 91 ..." for a reach, and because the action head *is* the language head, RT-2 can co-train on web vision-language data and robot data in one mixture. That co-training is where its celebrated generalization comes from, and we take it apart in Chapter 12.

Score it on the three axes. Multimodality: real but coarse. An autoregressive classifier over bins can, in principle, place probability on two separated bins and sample one, so RT-2 is not mode-averaging the way an L2 head does. But 256 bins per dimension is a blunt grid, and every action is snapped to a bin edge before it leaves the model, which is a quantization error baked into the representation (§10.4). Latency: this is where the choice hurts. Producing an action means decoding several tokens in sequence, and RT-2 runs at roughly 1–3 Hz, fine for slow pick-and-place, hopeless for anything dexterous. Smoothness: the bin grid is a lattice of small discontinuities, and RT-2 predicts a single next action rather than a chunk, so there is no within-chunk averaging to lean on. RT-2's head is the "reuse what we have" corner: maximal architectural simplicity, paid for in speed and in the granularity of motion.

OpenVLA: the same head, made open and studied

OpenVLA inherited RT-2's answer almost wholesale. It is a 7B-parameter model, a Llama 2 language backbone fed by two vision encoders, and its action head is again discrete token classification, 256 bins per dimension, decoded autoregressively (arXiv:2406.09246). You already met this head in §2.3, where the inference loop's last step turned predicted token IDs back into a 7-vector. OpenVLA's contribution was not a new head but an open, reproducible one: weights, code, and a fine-tuning recipe the rest of us can actually run.

Because it is open, OpenVLA is also where the discrete head's limits got measured in daylight. The bin grid caps precision; autoregressive decoding caps frequency at roughly the same single-digit Hz as RT-2; and fine-tuning to a new robot means the action distribution shifts under a tokenizer that was frozen to someone else's data. None of these are fatal, OpenVLA is a strong, widely used baseline precisely because the discrete head is simple and trains stably, but together they framed the open question the next two models answer: if the discrete-token head is what is throttling frequency and smoothness, what replaces it?

There were two answers, and they split.

FAST: keep the tokens, fix the tokenizer

The first answer kept the autoregressive head and blamed the tokenizer. Naive binning wastes tokens: consecutive timesteps in a smooth trajectory are nearly identical, so a per-timestep, per-dimension grid spends most of its tokens re-encoding "almost the same as last step." FAST (Frequency-space Action Sequence Tokenization, arXiv:2501.09747) instead runs the action chunk through a discrete cosine transform, the same frequency-space compression idea behind JPEG, and tokenizes the resulting coefficients. Smooth trajectories compress to a handful of low-frequency coefficients, so the same token budget now buys a longer, higher-frequency chunk.

FAST matters here because it shows the RT-2/OpenVLA lineage was not a dead end; the head was fine, the encoding was wasteful. A model with a FAST tokenizer can express high-frequency, contact-rich motion that a naive-binning model of the same size cannot, without abandoning the autoregressive machinery that makes co-training with language easy. It is the "repair the head we have" branch, and it is the branch $\pi 0$ reaches for when it needs a fast-to-train autoregressive variant. But the other branch threw the discrete head out entirely.

π0: a flow-matching action expert

π0 keeps the VLM backbone, a PaliGemma-class vision-language model, but attaches a separate **action expert**, a smaller (~300M-parameter) transformer that generates continuous action chunks by flow matching (§10.3), no discretization anywhere (arXiv:2410.24164). The backbone reads the image and instruction; the action expert takes that context and integrates a flow field from noise to a chunk of, typically, 50 continuous actions. This is the head built *for* control rather than inherited from language.

On the three axes it is the near-mirror image of RT-2. Multimodality: high and sharp, a flow model represents the full action distribution and samples a committed mode, so π0 handles the bimodal situations that make a mean-regression head stall (§10.2). Latency: this is the whole point. Flow matching's few-step sampling (§10.3), spread across a 50-action chunk by receding-horizon execution (§10.4), is what lets π0 run its control loop at up to 50 Hz, an order of magnitude past RT-2, on a larger model. Smoothness: continuous outputs mean no bin lattice, and chunk prediction gives within-chunk coherence, so the motion is smooth enough for folding and other high-frequency, contact-rich tasks. π0 is the "build the head the task needs" corner, and it pays for it with more moving parts, a second network, a flow objective, sampling to tune, than a single token head.

Notice that π0 did not have to choose *only* flow matching. The same architecture ships in an autoregressive-plus-FAST variant for faster training and a flow-matching variant for the smoothest, fastest inference. That the same backbone hosts both heads is the cleanest evidence for §10.4's thesis: the head is a swappable choice matched to a deployment target, not a fixed property of the model.

Helix: two heads at two clock rates

Helix, Figure's humanoid VLA, refuses to pick one point on the latency axis and instead occupies two at once. It is a dual-system design: a slow vision-language system (call it S2) reasons about the scene and the instruction at roughly 7–9 Hz, and a fast visuomotor system (S1) produces continuous motor commands at around 200 Hz, with S2's latent understanding fed to S1 as conditioning (figure.ai/news/helix). The full architecture is a Chapter 14 topic; what belongs here is the action-head logic.

Helix's answer to the three axes is to stop treating them as one decision. The tight-loop, smoothness-critical part of the problem, keeping a humanoid's many joints coordinated in real time, goes to a small, fast continuous head running far above any single big-model frequency. The slow, multimodal, semantic part, deciding *what* to do, which of several sane behaviors the instruction calls for, goes to the big VLM, which can afford to be slow because it does not run every motor tick. RT-2 put both jobs in one autoregressive head and got a rate that served neither well; Helix splits the jobs so each runs at the rate it needs. It is the strongest illustration that "which action head" can be the wrong question, sometimes the answer is *two heads*, at two clocks.

What the four choices add up to

Line the four up and the trend is not subtle. RT-2 reused the language head and discovered its ceiling. OpenVLA made that head open and let the community measure the ceiling precisely. FAST showed the ceiling was in the tokenizer, not the head, and lifted it. π0 replaced the head with a continuous flow-matching expert and broke into the high-frequency, smooth-control regime the discrete head could not reach. Helix stopped insisting on one head at all. The through-line is the field moving from *inherited* action heads toward heads *designed* for the physics of control, and, at the frontier, toward decoupling the semantic decision from the motor decision so each can use the head that suits it.

Three practical takeaways survive out of this. First, discrete-token heads (RT-2, OpenVLA) are simple, train stably, and co-train naturally with language, at the cost of frequency and precision, a good default when your loop is slow and your motions are coarse. Second, continuous flow or diffusion heads (π0) are what you reach for when smoothness and control frequency are load-bearing, and you accept more architectural complexity to get them. Third, when no single frequency serves both the thinking and the moving, a dual-system split (Helix) lets you have both. Section 10.6 gathers the chapter, diffusion, flow matching, the three axes, and these four choices, into the short list of things you should now be able to reason about without looking anything up.

10.6 Summary

Chapter 10 answered a question the earlier chapters kept deferring. A policy has to turn a chosen intention into numbers, joint velocities, gripper commands, end-effector deltas, and until now the book had been vague about how. The naive answer, regress to the mean action, fails in a specific and common way: when two good actions bracket a bad one, their average is the bad one. This chapter was about the generative machinery that fixes that failure, where it came from (image synthesis), how it was adapted to control, and which of the shipping VLAs use which variant. The through-line is that action generation is a design space with real axes, not a solved detail, and the point of the chapter was to make you fluent in those axes before Part 4 starts naming models that live at different corners of them.

The four ideas worth carrying forward

Diffusion generates by learning to reverse noise, and that indirection is exactly what lets it represent more than one right answer. §10.1 built the mechanism from the ground up: corrupt data with Gaussian noise over many steps, then train a network to predict the noise that was added, and sampling becomes a walk backward down that ladder from pure noise to a clean sample (Ho et al. 2020). The reason this matters for control is the multimodality argument, which the whole chapter leaned on afterward. A network trained with an L2 loss to output *the* action learns the conditional mean, and the mean of two valid trajectories can be an invalid one, reach left or reach right averages to reach into the obstacle. A diffusion model never regresses to that mean; it learns the full conditional distribution and samples a committed sample from it, so the two ways around the obstacle stay two ways around the obstacle. That single property is why generative heads displaced regression heads for contact-rich manipulation.

Diffusion Policy and ACT turned image generators into action generators by predicting chunks, not single steps. §10.2 was where the abstraction became robotics. Diffusion Policy (Chi et al. 2023) conditions the denoiser on observations and denoises a short *sequence* of future actions at once; ACT (Zhao et al. 2023) reaches the same place through a different door, a conditional VAE that predicts an action chunk and averages overlapping predictions at execution time through temporal ensembling. The shared move, action chunking, is the load-bearing idea. Predicting a horizon of actions and executing part of it before replanning smooths the output, cuts the decision frequency the model has to sustain, and blunts the compounding error from §6.3, because a chunk commits to a short plan instead of reacting one jittery step at a time. Remember ACT and Diffusion Policy as the pair that proved a generative head is not a luxury for a research demo but the default for fine bimanual work on cheap hardware.

Flow matching is the same idea made straighter and faster, and that speed is what made it viable at control rate. §10.3 introduced the alternative that the frontier VLAs actually ship. Instead of a long stochastic denoising chain, flow matching (Lipman et al. 2023) trains a network to predict a velocity field that transports noise to data along a path, and sampling integrates that field as an ODE. Rectified flow (Liu et al. 2023) then straightens the paths so the integration needs very few steps, sometimes one. The consequence is the number of function evaluations per action drops from diffusion's tens to a handful, which is the difference between a head you can only run on slow pick-and-place and a head you can run inside a real-time loop. Flow matching is the machinery under $\pi 0$ (arXiv:2410.24164), and it is why the smooth-control VLAs of Chapter 13 are possible at all.

Every action head is a choice on three axes, and no single point wins. §10.4 gave the analytical frame the rest of the chapter measured against: multimodality (can the head represent more than one valid action?), latency (how many network calls does one action cost, and can the loop keep control rate?), and smoothness (does the output avoid jitter and discontinuity?). These trade against each other. A discrete autoregressive head is simple and co-trains with language but decodes slowly and snaps motion to a grid. A diffusion head is maximally expressive but pays in sampling steps. A flow-matching head recovers most of the expressiveness at a fraction of the latency but adds a second network and an objective to tune. The chapter's insistence is that you cannot rank these in the abstract, you rank them against a deployment target, a control frequency, and a task's demand for precision.

What you should be able to do now

Four concrete capabilities, in roughly the order Part 4 will ask for them.

You should be able to *explain why a generative action head beats a regression head, in one specific failure mode.* Not

"diffusion is more powerful," but the actual mechanism: an L2 head learns the conditional mean, the mean of a multimodal action distribution can be a non-action, and a generative head avoids this by modeling and sampling the distribution instead of collapsing it. §10.1 and §10.2 built this so that when a later model justifies its diffusion or flow head by pointing at bimodal behavior, you read it as a precise claim rather than a fashion.

You should be able to *describe action chunking and say what three problems it addresses at once.* Given a policy that predicts a horizon of actions and executes part before replanning, you should be able to name the payoffs, smoother output, lower required decision frequency, and reduced compounding error, and connect the last of these back to the DAgger discussion of §6.3. This is the idea that recurs in almost every modern head, so it is worth being able to state without hedging.

You should be able to *place diffusion and flow matching on a steps-versus-quality spectrum and pick between them for a latency budget.* Shown a control loop that must run at, say, 30 Hz, you should be able to reason that a tens-of-steps diffusion sampler is likely too slow, that rectified flow's few-step integration is the reason $\pi 0$ clears the bar, and that the number of function evaluations is a dial you can turn rather than a fixed cost. §10.3 and §10.4 were training exactly this judgment.

You should be able to *look at any VLA and classify its action head on the three axes.* Given a new system, you should be able to ask whether its head is discrete-token, diffusion, flow-matching, or split across clock rates, and predict the resulting trade in frequency, precision, multimodality, and architectural complexity, the same reading §10.5 ran on RT-2, OpenVLA, $\pi 0$, and Helix. That classification is the analytical habit the chapter exists to install, and it is the one you will use most in Part 4.

Where the chapter has set up the rest of the book

Chapter 10 hands off forward more directly than any chapter since Part 1, because the machinery it built is machinery the foundation models use by name. The first and largest handoff is to Chapter 13, which is $\pi 0$ in full. §10.3 and §10.5 gave you the action expert and the flow-matching objective as ideas; Chapter 13 puts the whole model together and argues that continuous flow-matching heads are what let a VLA fold laundry rather than shuffle blocks. A reader who understood flow matching here reads Chapter 13 as the payoff, not as a new concept.

The second handoff is to Chapter 12, where RT-2, OpenVLA, and Octo return with their backbones and data stories attached. §10.5 deliberately looked at their heads only, discrete tokens for RT-2 and OpenVLA, a diffusion head for Octo, and left the scaling and co-training arguments for later. Chapter 12 supplies those arguments, and the head-level reading from this chapter is what keeps the discussion honest: it is easy to credit a model's generalization to its scale and forget that its action head is capping its control frequency the whole time.

The third handoff is to Chapter 14 and the dual-system architectures. Helix in §10.5 was the preview: two heads at two clock rates, a slow semantic system and a fast motor one. Chapter 14 builds that hybrid properly, and §10.4's insight that latency, multimodality, and smoothness need not be served by one decision is the frame that makes the dual-system split read as an answer to a real tension rather than as extra machinery for its own sake. The thread that ran from §9.5's world-model-versus-VLA debate through this chapter's heads lands in Chapter 14 as an architecture you can actually deploy.

What the chapter has not covered

Two omissions are worth naming so they do not read as gaps later. The chapter developed diffusion and flow matching as generative tools for *action*, but said little about their other robotics use, generating *observations*, the video-prediction world models of §9.4 that were explicitly deferred to here for their generative mechanism. The mechanism is now in hand, and Chapter 15's data discussion and Chapter 18's open problems are where the same denoising and flow machinery reappears on the perception side. The split was deliberate: action generation is the load-bearing case for this book, and it was worth covering cleanly before branching.

The chapter also stayed light on training economics. It said flow matching is cheaper to sample than diffusion and that FAST makes autoregressive heads train faster, but it did not weigh the cost of *training* each head on the scale of data a foundation model sees, or the cost of fine-tuning one to a new robot. Those numbers belong with the practical

fine-tuning machinery of Chapter 16, where picking a base model and its head is a budget decision as much as a capability one, and where the abstract axes of §10.4 get priced.

Chapter 10's contribution to the book's overall argument is to remove the last piece of hand-waving between "the model decided what to do" and "the robot moved." The three findings to carry forward are that a generative head exists to represent multimodal action distributions that a regression head collapses; that action chunking, shared across Diffusion Policy, ACT, and every flow-matching VLA, buys smoothness and error tolerance at once; and that flow matching's few-step sampling is the specific advance that let generative heads run inside real-time control loops. With the head understood, Part 4 can finally name the foundation models and argue about how they scale, confident that you can see past the backbone to the action head doing the last, physical mile of work.

§10.x closes the chapter with a hands-on exercise, implementing a small diffusion action head and a flow-matching one on the same chunked-prediction task, and measuring how sampling steps trade against control rate, followed by the full reading list for the chapter.

10.x Hands-on exercise + chapter references

Chapter 10 spent five sections insisting that the generator strapped to an action chunk is a design choice, not a law of nature, that diffusion and flow matching produce the same kind of output and differ mainly in how many network calls they cost you to get it. That claim is easy to accept on paper and easy to forget the moment you have a policy that works. The way to make it stick is to build one policy, swap the head, and hold the two side by side on identical data. The headline exercise does exactly that, and it is the drill the TOC names: train a Diffusion Policy on PushT, replace the diffusion head with a flow-matching head, and compare what changes and what does not. The supporting drills probe the edges the chapter argued about, whether the multimodality of §10.1 is real or a story, what the sampling-step count actually buys you, and how the receding-horizon window of §10.2 trades smoothness against reactivity.

Budget about three hours. Most of it is two training runs on a small task that you can start and walk away from; the code you write on top is short. Start the first run early and read while it trains.

```
pip install diffusion_policy   # or: pip install lerobot, which ships both heads
```

PushT is the task from §10.2: a circular pusher must nudge a T-shaped block into a target pose in a 2-D plane. It exists precisely to make bimodal behavior visible, the block can be approached from either side, and it trains in minutes, not hours, on a single GPU. Everything below runs offline against the demonstration set that ships with the benchmark.

Exercise 10.x.1 — Train Diffusion Policy, then swap in a flow-matching head

Start from a working Diffusion Policy on PushT. Use the reference implementation rather than writing the denoiser from scratch; the point of this exercise is the comparison, not a reimplementation of §10.1. Train it on the PushT demonstrations with the defaults, a 1-D convolutional U-Net over the time axis, an action chunk of sixteen steps, the last two observations as the condition, DDIM sampling at roughly ten steps, and confirm you can reproduce the benchmark's headline: a success rate well above what a plain mean-regression policy reaches on the same data. That gap is §10.1's multimodality argument cashed out, and you want it in hand before you touch anything.

Now the swap the TOC asks for. Keep every part of the system fixed, the vision encoder, the U-Net backbone, the chunk size, the receding-horizon execution, and change only the training objective and the sampler. Where diffusion trains ϵ_θ to predict the noise added at a random level t, the flow-matching head of §10.3 trains a velocity field $v_\theta(\mathbf{a}_t, t, o)$ to point from a noise sample straight toward the clean action chunk along the linear interpolation between them:

```
# a0: clean action chunk;  a1 ~ N(0, I): noise
# t ~ Uniform(0, 1)
a_t   = (1 - t) * a0 + t * a1          # linear path from data to noise
target = a1 - a0                        # constant velocity along that path
loss  = ((v_theta(a_t, t, obs) - target) ** 2).mean()
```

Sampling is now integrating an ODE from noise back to data, a handful of Euler steps, sometimes as few as one or two after the path has been straightened, rather than the ten-rung denoising ladder. Retrain on the same PushT demonstrations and evaluate the same way.

The result you are looking for is deliberately anticlimactic. Success rate should land close to the diffusion policy's, within noise, if you tuned both fairly, because both heads represent the same multimodal distribution over the same chunk. What moves is inference cost: the flow head reaches comparable quality at far fewer function evaluations. Record both numbers, success rate and network calls per chunk, for both heads, and write one sentence: the generator was interchangeable, the compute was not. That sentence is the whole thesis of §10.4, and you now have it as two rows in a table you produced.

Wall clock: about ninety minutes, most of it the two training runs.

Exercise 10.x.2 — Sweep the sampling steps and find where quality saturates

This drill turns "the compute was not interchangeable" into a curve. Take both trained heads from 10.x.1 and vary the number of function evaluations (NFE) used at sampling time, for the diffusion head, the number of DDIM steps; for the flow head, the number of Euler integration steps. Sweep each across a range, say 1, 2, 4, 8, 16, 32, and record PushT success rate at every setting. Plot NFE against success for both heads on one axis.

The two curves tell the §10.4 story in one picture. The diffusion head's success collapses at very low step counts and climbs as you add steps, plateauing somewhere around eight to ten, below that, the denoiser has too few rungs to land on the data manifold and the actions come out garbled. The flow head's curve is shifted left: it reaches its plateau at a handful of steps, and if you applied reflow (§10.3) it may hold up even at one or two. Mark the NFE where each curve flattens. That knee is the head's honest latency cost, and the gap between the two knees is exactly why π0 (Black et al., 2024, arXiv:2410.24164) chose a flow-matching head for real-time control rather than a diffusion one. For the written part, multiply each knee by your measured per-call latency and state the control frequency each head can actually sustain. One of them may miss a 10 Hz budget; that is the point.

Wall clock: about forty minutes, reusing both trained models.

Exercise 10.x.3 — Kill the multimodality and watch PushT stall

§10.1 claimed that averaging two good actions produces a bad one, and PushT was built to expose it. Make the failure appear on purpose. Train a third policy on the same data with the same backbone and chunk, but replace the generative head with a plain regression head, one that outputs the action chunk directly under an L2 loss, no noise, no latent, no velocity field. This is the mean-regression baseline §10.1 warned about, and PushT will punish it.

Run it and watch where it fails. On approaches where the demonstrations split between going around the block's left and right, the regressed action is the average of the two, which drives the pusher straight into the flat edge of the T and stalls. Log the states where the episode stalls and confirm they cluster at exactly these decision points, not at random. Then compare success rates across all three heads, regression, diffusion, flow, on the identical dataset. The regression head should sit well below the other two, and the two generative heads should sit together. That ordering is the chapter's central claim rendered as three numbers: the generator has to be able to represent more than one mode, but which multimodal generator you pick is close to a wash on quality.

Wall clock: about thirty minutes; the regression head trains fastest of the three.

Exercise 10.x.4 — Tune the receding-horizon window

The last drill isolates the deployment knob from §10.2 that has nothing to do with the generative head at all. Diffusion Policy predicts a chunk of T_p actions and executes only the first T_a before replanning. Take your flow head from 10.x.1 (it is the cheapest to sample, so this sweep is fast) and vary T_a while holding the predicted horizon T_p fixed at sixteen. Try executing the full chunk ($T_a = 16$, replan rarely), a prefix ($T_a = 8$), and nearly every step ($T_a = 1$, replan constantly). Record success rate and, if your harness exposes it, a smoothness measure such as mean squared jerk of the executed trajectory.

The trade-off should separate cleanly. Executing the whole chunk gives the smoothest motion, the policy commits to a plan and follows it, but the robot is blind to anything that changes mid-chunk, so success dips on any perturbation. Replanning every step keeps it maximally reactive but reintroduces the jitter at chunk boundaries that §10.2 said temporal ensembling exists to suppress. Somewhere in between is the setting that keeps PushT both smooth and reactive. Name it, and note that you found it without retraining anything: this is a pure inference-time knob, orthogonal to the head choice you spent the first three exercises studying. Keeping those two axes separate, what generator, and how you schedule its output, is the mental model to carry into Part 4.

Wall clock: about twenty-five minutes, sweeping one trained model.

Chapter 10 reading list

The works below are cited in §10.1–§10.6, grouped by purpose. Full bibliographic entries for everything cited in the book live in Appendix E.2; this is the chapter-local subset.

The diffusion mechanism

- Ho, J., Jain, A., & Abbeel, P. (2020). "Denoising Diffusion Probabilistic Models." *NeurIPS 2020*. The corrupt-then-denoise recipe §10.1 builds from and the DDPM sampler Exercise 10.x.2 sweeps.
- Song, Y., et al. (2021). "Score-Based Generative Modeling through Stochastic Differential Equations." *ICLR 2021*. The score-matching, SDE view §10.1 names as the second dialect for the same object.

Diffusion and CVAE action heads

- Chi, C., et al. (2023). "Diffusion Policy: Visuomotor Policy Learning via Action Diffusion." *RSS 2023*. The system Exercises 10.x.1–10.x.4 train; the action chunk, receding-horizon execution, and the PushT result at the center of this chapter.
- Zhao, T. Z., Kumar, V., Levine, S., & Finn, C. (2023). "Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware" (ACT). *RSS 2023*. §10.2's single-pass CVAE alternative and the temporal-ensembling counterpart to receding horizons.

Flow matching and rectified flow

- Lipman, Y., et al. (2023). "Flow Matching for Generative Modeling." *ICLR 2023*. The velocity-field objective §10.3 derives and Exercise 10.x.1 swaps in.
- Liu, X., Gong, C., & Liu, Q. (2023). "Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow." *ICLR 2023*. The reflow procedure §10.3 uses to straighten the sampling path down to one or two steps, the effect Exercise 10.x.2 measures.

Action heads in shipping VLAs

- Brohan, A., et al. (2023). "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control." arXiv:2307.15818. §10.5's discrete action-token classifier, the pole opposite continuous diffusion.
- Kim, M. J., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246. §10.5's open discrete-token policy; the baseline Part 4 returns to repeatedly.
- Octo Model Team (2024). "Octo: An Open-Source Generalist Robot Policy." arXiv:2405.12213. §10.5's generalist policy with a diffusion action head, developed in Chapter 12.
- Black, K., et al. (2024). "π0: A Vision-Language-Action Flow Model for General Robot Control." arXiv:2410.24164. The flow-matching head §10.3 and §10.5 build toward and Exercise 10.x.2 explains the latency motivation for; Chapter 13 in full.
- Pertsch, K., et al. (2025). "FAST: Efficient Action Tokenization for Vision-Language-Action Models." arXiv:2501.09747. §10.5's frequency-domain action tokenizer, the discrete-token camp's answer to the smoothness problem.

Chapter summary

Chapter 10 answered the question every earlier chapter deferred: once a policy has decided what to do, how does it emit the numbers, and how does it emit them when more than one answer is right? You can now explain the diffusion mechanism without hand-waving, corrupt a sample along a fixed noise schedule, train a network to predict the noise, and generate by denoising from pure noise, and say why that machinery represents a full multimodal distribution where mean-regression collapses to a single averaged, often colliding, action. You can build the two reference action heads of 2023, Diffusion Policy and ACT, and explain the one idea they share, the action chunk, and the two knobs they use to turn overlapping chunks into smooth commands: receding-horizon execution and temporal ensembling, the latter of which Exercise 10.x.4 had you tune by hand. You can describe flow matching in plain terms, learn a velocity field that carries noise to data along a straight path, and say precisely where it beats diffusion for control: not on quality, which Exercise 10.x.1 showed is a wash, but on sampling steps, the axis Exercise 10.x.2 measured and the reason $\pi 0$ runs in real time. And you can decide which head a task actually needs by weighing the three quantities this chapter kept returning to, latency, multimodality, and smoothness, against what the robot can afford. That last skill is the one Part 4 leans on. Starting in Chapter 11, we stop studying action heads in isolation and read the modern VLA literature paper by paper, beginning where the recipe began: CLIP, language conditioning, and the discrete action tokens of RT-1.

Chapter 11 — The VLA recipe: from CLIP to RT-1

11.1 CLIP and the multimodal pretraining moment

The four chapters ahead of you are about the same trick applied at larger and larger scale, so it is worth naming the trick before we get lost in model names. A vision-language-action model is a policy that takes a camera image and a sentence and emits an action, and it works because someone else already spent millions of dollars teaching a network that a picture of a mug and the word "mug" belong together. The policy inherits that alignment for free and only has to learn the last step, from aligned perception to motor commands. Every model in Part 4 is a variation on where that inherited alignment comes from and how the action step is bolted on.

This section is about where the alignment came from. In January 2021 two groups, one at OpenAI and one at Google, published the same idea within weeks of each other: stop labeling images and let the caption be the label. The OpenAI version is called CLIP (Radford et al., 2021), the Google version ALIGN (Jia et al., 2021), and CLIP is the one the robotics field ended up building on, so it gets top billing here. The reason it mattered for us has almost nothing to do with the task CLIP was built for and everything to do with a side effect: it produced an image encoder whose features already speak the same language as text. Chapter 6 taught you that imitation is the dominant training signal in robotics; this chapter is the story of what happens when you point that signal at a perception stack that was pretrained to understand language. But first, the perception stack.

The problem CLIP was actually solving

Before 2021, the standard way to get a good image encoder was to train it on ImageNet: a million photographs, each stamped with exactly one of a thousand class labels. The resulting network was excellent at the thousand things it had seen and useless at the thousand-and-first. If your robot needed to recognize a "spatula" and spatula was not one of the thousand classes, you collected a spatula dataset and fine-tuned. The label set was a fence, and everything outside it was invisible.

The fence was not a modeling limitation; it was a data-collection decision. Someone had to sit down and decide the thousand categories, then pay annotators to sort images into them. That does not scale, and worse, it throws away most of what an image's caption already tells you. A web photo captioned "our golden retriever puppy chewing a blue slipper on the porch" contains a dozen concepts and their relationships, and the ImageNet pipeline reduces all of it to the single token "dog," if that.

CLIP's move was to treat the caption as the supervision and skip the fixed label set entirely. Radford and colleagues scraped roughly 400 million image-text pairs off the public web, no human relabeling, just whatever caption happened to sit next to each image, and trained on the raw pairs. ALIGN pushed the same idea to over a billion pairs and made a virtue of the noise, showing that at that scale you do not even need to clean the captions. The lesson both papers drove home is one this book returns to constantly: a mediocre signal at enormous scale beats a clean signal at small scale, and the web is the only place the enormous scale lives.

Contrastive learning in one matrix

Here is the mechanism, because it is simpler than its reputation. You have two encoders. One is an image encoder f (a vision transformer or a ResNet) that turns a photo into a vector; the other is a text encoder g (a transformer) that turns a caption into a vector. Both project into the same d-dimensional space, and both outputs are normalized to unit length so that a dot product between them is just a cosine similarity.

Take a batch of N image-caption pairs that actually go together. Encode all N images and all N captions, and form the $N \times N$ matrix of similarities $S_{ij} = f(\text{image}_i) \cdot g(\text{caption}_j)$. The N entries on the diagonal are the real pairs; the $N^2 - N$ off-diagonal entries are mismatches, image i against some other image's caption. The training objective is a sentence long: make the diagonal large and everything else small. Concretely, treat each row as a classification problem over N options where the correct answer is the diagonal entry, apply the cross-entropy loss you already know from §3.2, and do the same down each column. That is it. No new loss function, no adversary, no reconstruction term. It is the InfoNCE contrastive loss, and in PyTorch the core is about six lines:

```
# image_emb, text_emb: (N, d), each row L2-normalized
logits = image_emb @ text_emb.T          # (N, N) similarity matrix
logits = logits * temperature.exp()      # learned scalar sharpens the softmax
labels = torch.arange(N, device=logits.device)   # correct match is the diagonal
loss_i = F.cross_entropy(logits, labels)          # each image picks its caption
loss_t = F.cross_entropy(logits.T, labels)        # each caption picks its image
loss = (loss_i + loss_t) / 2
```

The batch supplies its own negatives, which is the quiet efficiency of the scheme: with $N = 32{,}768$, every pair is contrasted against tens of thousands of wrong answers at once, no negative mining required. Scale the batch up and each gradient step gets sharper on its own.

The side effect that mattered: zero-shot

CLIP was pitched as a zero-shot image classifier, and the demo is worth walking through because it shows what the shared space buys you. Suppose you want to classify a photo as cat, dog, or spatula, and you never trained on any of those labels. Write each label into a sentence, "a photo of a cat," "a photo of a dog," "a photo of a spatula," and run all three through the text encoder. Run the photo through the image encoder. Whichever caption vector sits closest to the image vector is your prediction. The class list is now just text you type at inference time, so adding "spatula" costs you a sentence instead of a dataset. The fence is gone.

That is a neat classification result, but the deeper fact is the shape of the space it lives in. CLIP did not learn "cat" and "dog" as opaque category IDs. It learned a geometry in which visual features and word meanings are laid out together, so that directions in image space correspond to directions in language space. A model that can find the mug in a picture because you asked for "mug" is one small step from a model that can pick up the mug because you asked it to; the perception half of that sentence is already solved, and solved in a way that generalizes to objects and phrasings nobody enumerated in advance.

Why a robotics book cares

Return to the trick from the top of the section. A policy trained by pure imitation (Chapter 6) learns to map pixels to actions, and its understanding of the world is bounded by the demonstrations it saw. Show it a thousand demos of stacking red blocks and it stacks red blocks; ask for the green one and you are outside the fence again. The demonstrations are expensive, a human teleoperates every one, so the fence is low and you cannot buy your way past it the way CLIP bought its way past ImageNet. Robot data does not exist on the web at 400-million-pair scale.

Language-conditioned pretraining is the escape. If your policy's vision front-end is a CLIP-style encoder, it arrives already knowing that mugs, cups, and "the thing you drink from" point in similar directions, and that knowledge was paid for with web images, not teleop hours. The policy's own scarce demonstrations then only need to teach the mapping from *aligned, language-aware features* to motor commands, which is a far smaller thing to learn than perception-plus-control from scratch. This is exactly the claim in the chapter's third learning objective, that language

conditioning unlocked generalization imitation alone could not reach, and §11.2 makes it concrete with BC-Z and RT-1, the first policies to cash the check.

The bridge is not hypothetical. PaLM-E (Driess et al., 2023, arXiv:2303.03378) took the idea to its logical end by injecting robot sensor data directly into a large language model's token stream, so that "multimodal sentences" interleaving words, images, and states could be reasoned over by one network; it is the clearest early demonstration that a language-pretrained backbone transfers to embodied tasks. RT-1 (Brohan et al., 2022, arXiv:2212.06817), which anchors the rest of this chapter, made the more modest and more copied choice: use a pretrained language embedding to condition a comparatively small transformer policy, and spend the saved capacity on real-robot data. We will pull RT-1 apart in §11.2 and §11.4.

What CLIP did not solve

Two cautions, so you do not overclaim. First, alignment is not grounding in action. CLIP knows that an image of an open drawer and the words "open drawer" are near each other; it has no idea how to move a gripper to produce that outcome, and nothing in its training touched a robot. The action half of vision-language-action is new work that Chapters 11 through 14 are about, and it is the hard half. Second, CLIP's world is a single still image and a short caption. It has no notion of time, force, or the difference between a mug that is empty and one that is full to the brim. A control policy lives and dies on exactly those distinctions, and inheriting a static-image encoder means inheriting its blind spots along with its strengths. Keep both limits in mind; several design decisions in the models ahead are attempts to work around them.

With the perception stack pretrained and its features aligned to language, the recipe has its first ingredient. The next section adds the second, a way to condition a policy on an instruction and turn those aligned features into motor commands, and gives it a name: RT-1.

11.2 Language-conditioned imitation: BC-Z, RT-1

Section 11.1 left you with an image encoder that already understands language and no way to make it do anything. This section closes that gap. The mechanism is the one you met in Chapter 6, behavior cloning, supervised imitation of a human demonstrator, with a single addition: the policy is told, in words, which task it is imitating. That addition sounds trivial and is not. It is the difference between a policy that learns one skill per network and a policy that learns hundreds in one network and can be asked for any of them at run time.

Recall the shape of plain behavior cloning. You collect demonstrations, each a sequence of observation-action pairs (o_t, a_t), and you fit a network $\pi_\theta(o_t) \to a_t$ by minimizing the difference between its output and the human's action. Nothing in that setup tells the policy what it is doing. If half your demonstrations put a can in a drawer and the other half wipe a table, the network sees contradictory targets for similar-looking images and averages them into mush. The standard fix before 2022 was one policy per task: separate networks, separate training runs, separate everything. That does not scale to a robot you would actually want in a kitchen.

Conditioning: add the task to the input

The fix is to make the task part of the observation. Instead of $\pi_\theta(o_t)$, train $\pi_\theta(o_t, c)$, where c is a task descriptor, an embedding of the instruction "put the can in the drawer." Now the two demonstration groups no longer contradict each other; they differ in c, and the network learns to route on it. Feed c for wiping and you get wiping behavior; feed c for the drawer and you get drawer behavior, from one set of weights. The contradictory targets become a feature: the network is learning a *conditional* map, and the condition disambiguates.

Where does c come from? This is exactly where §11.1 pays off. If you embed the instruction with a pretrained language model, two things follow. First, you get c for free, no per-task ID to hand-assign, just run the sentence through the encoder. Second, and this is the point of the whole chapter, the embedding for "put the can in the drawer" sits near the embedding for "place the soda in the cabinet" because the language model already knows those sentences mean nearly the same thing. A policy conditioned on that space inherits the similarity. Train it on the first instruction and it has a running start on the second, even though no demonstration ever used those words. Language conditioning turns the instruction from an arbitrary label into a coordinate in a space where nearby points ask for

nearby behaviors. That is the generalization plain imitation could not reach, and it is the chapter's third learning objective made concrete.

BC-Z: the conditioning idea, proved

BC-Z (Jang et al., 2022) is the cleanest early demonstration that the idea works. The team collected demonstrations for around 100 manipulation tasks on a single real robot, then asked the question that matters: can the policy do a task it never saw demonstrated, purely because you described it? To make that question answerable they conditioned the policy two ways. One conditioning signal was a language embedding of the task name; the other was an embedding of a human performing the task on video. Both are just vectors c fed alongside the image; the network does not care whether the vector came from words or from watching a person.

Two design choices in BC-Z are worth naming because the field kept them. First, the conditioning vector is injected through FiLM, feature-wise linear modulation, rather than concatenated onto the input pixels. FiLM lets c scale and shift the convolutional feature maps of the vision network, so the task descriptor reaches deep into perception rather than sitting off to the side. Concretely, for a feature map h, FiLM computes $\gamma(c) \odot h + \beta(c)$, where γ and β are small networks that turn the task embedding into per-channel gains and offsets. The instruction is thus allowed to change *what the vision stack looks for*, which is what you want: "find the can" and "find the sponge" should light up different features.

Second, BC-Z leaned on scale and continuous data collection to fight the compounding-error problem from §6.3. A cloned policy drifts into states its demonstrations never covered and has no idea what to do there. BC-Z's answer was a shared-autonomy loop: humans supervised the policy as it ran, took over when it faltered, and those interventions became fresh demonstrations exactly in the states the policy got wrong. The dataset grows where the policy is weakest. Hold this thought; it is the same instinct, more data, aimed well, that RT-1 turns into an industrial process.

The headline result was modest in absolute terms and large in what it implied: BC-Z completed a meaningful fraction of tasks it had never been trained on, driven only by the language or video description. A single network, told what to do in words, generalizing to new words. The recipe from §11.1 had cashed its first check.

RT-1: the same idea at fleet scale

RT-1 (Brohan et al., 2022, arXiv:2212.06817) is BC-Z's idea with the constants turned up and the architecture modernized from a convolutional policy to a transformer. The conditioning is the same in spirit, the instruction is embedded (with a Universal Sentence Encoder) and used to modulate perception through FiLM, but almost every other quantity is an order of magnitude larger, and the output representation changes in a way that the chapter's rest depends on.

Walk the forward pass. RT-1 takes a short history of camera images, six of them, not just the current frame, so the policy can see motion and infer things a single frame hides, whether the gripper is already closing, whether an object is sliding. Each image goes through a FiLM-conditioned EfficientNet, which turns the picture into a grid of feature vectors already shaped by the instruction. That grid is large, so a module called TokenLearner squeezes it down to a handful of the most informative tokens per frame, a learned attention-based compression that keeps inference cheap enough to run on a real robot. The resulting tokens, across all six frames, feed a decoder-only transformer, which attends over the sequence and emits the action.

The action itself is where RT-1 makes its most-copied move. Rather than regress continuous joint or gripper values, RT-1 *discretizes* each action dimension into 256 bins and predicts a token per dimension, arm translation, arm rotation, gripper, plus a discrete mode switch between moving the arm, moving the base, and terminating. Control becomes next-token prediction, the same objective a language model trains on, which is why a transformer is a natural fit. Why bins instead of real numbers gets a section of its own: this is the "small idea with large consequences" that §11.3 unpacks, and the chapter's hands-on exercise asks you to build the tokenizer yourself. For now, take it as given that the action is a short string of tokens the transformer produces one at a time.

The data is the part that does not fit in a diagram. RT-1 was trained on roughly 130,000 demonstrations spanning more than 700 distinct language instructions, collected by a fleet of 13 Everyday Robots over about 17 months in real office-kitchen environments. That last clause is the expensive one. Every one of those episodes is a human teleoperating a physical robot in a real space, not a simulation run overnight on a cluster. §11.5 is about what that

investment bought and at what point the curve of return justified it; the short version is that RT-1's generalization and robustness scaled with the breadth of that dataset in a way smaller efforts never showed, and §11.4 is about which of RT-1's gains came from the transformer and which came from the data, because the two are easy to confuse and the answer is not the flattering one.

What conditioning does and does not buy

Be precise about the win, because it is narrower than the demos suggest. A language-conditioned policy generalizes well to *new phrasings and recombinations of familiar skills*, "pick up the apple" when it has seen "pick up the orange" and "pick up the apple" separately in other contexts. The aligned language space (§11.1) is what makes that transfer cheap. What it does not do is invent motor skills it never practiced. Ask RT-1 to do something mechanically unlike anything in its 700 instructions, fold a shirt, when it has only ever done pick-and-place and drawer manipulation, and no amount of clever phrasing conjures the behavior. The language covers the *what*; the demonstrations still have to cover the *how*. Conditioning widens the front door of instruction; it does not fill the room behind it.

There is also a subtler cost. Because the instruction modulates perception, a bad or out-of-distribution instruction quietly corrupts what the policy sees, not just what it decides. Feed RT-1 a sentence unlike its training instructions and you do not get a clean "I don't know," you get confidently wrong perception feeding confidently wrong action. This is the flip side of FiLM reaching deep into the vision stack, and it is one reason later models (Chapter 12) move toward backbones pretrained on far broader language, so that "out of distribution" is a much larger set to fall out of.

Both BC-Z and RT-1 share one architectural commitment we have been deferring: they turn actions into tokens and predict them discretely. That choice is not obvious, actions are continuous, and binning them throws information away on purpose, so the next section makes the case for why it is worth it and shows what breaks if you get the discretization wrong.

11.3 Action tokenization: a small idea with large consequences

Section 11.2 slipped one decision past you on purpose. RT-1 does not output a robot action; it outputs a short string of tokens, one per action dimension, each chosen from 256 discrete bins. A transformer that was built to predict the next word now predicts the next slice of a gripper command, and the training objective is the same cross-entropy loss a language model uses. The move looks like an implementation detail. It is closer to a fork in the road, and most of the models in Part 4 are standing on one side of it.

Here is the tension. A robot action is continuous. The commanded velocity of a joint is a real number, and so is the change in gripper aperture, and so is every rotation you might send to an arm. Continuous quantities want continuous methods: regress the number, minimize squared error, done. Chapter 3 spent a section on exactly that loss. So why would anyone take a smooth, real-valued target and deliberately chop it into 256 buckets, throwing away all the precision that lives between bin edges?

The case for turning actions into tokens

The answer starts with what a transformer is good at, and it is not regression. A decoder-only transformer trained with cross-entropy is a machine for modeling distributions over discrete symbols. Feed it text and it learns $p(\text{next word} \mid \text{context})$. If you want to reuse that machine for control, and reusing it is the entire premise of the VLA program, the cheapest path is to make actions look like words. Discretize each action dimension into bins, treat each bin as a symbol, and control becomes next-token prediction over an action vocabulary. Nothing about the architecture has to change. The optimizer, the attention stack, the sampling code, even the pretrained weights carry over untouched. That reuse is the whole point, and it is why RT-2 (arXiv:2307.15818) could take a vision-language model trained on web text and images and teach it to drive a robot without redesigning the network: the actions were folded into the same token stream as everything else.

There is a second reason, and it matters more than the engineering convenience. Regression to a single number commits the policy to one answer. Averaged over a dataset where the same observation sometimes precedes a left turn and sometimes a right, least-squares regression splits the difference and drives straight into the obstacle between them. This is the multimodality problem from §10.4, and discretization sidesteps it for free. A distribution

over 256 bins can put mass on the left-turn bin *and* the right-turn bin and leave the middle empty. Sample from it and you get one or the other, never the disastrous average. Cross-entropy over bins is a full distribution, not a point estimate, so the policy can represent "either of these, but not the thing between them," which is precisely the shape real manipulation data has.

How RT-1 actually does it

RT-1's scheme is the one everyone learned first, so it is worth stating in mechanical detail. Take one action dimension, say the commanded x-translation of the end effector. It lives in some known range, clip it to $[-1, 1]$ after normalization. Slice that range into 256 equal-width bins. Any real value now falls into exactly one bin, and you replace the number with that bin's integer index, a symbol between 0 and 255. Do this independently for each of the action dimensions RT-1 controls, arm translation and rotation and gripper and the discrete mode switch, and one continuous action vector becomes a fixed-length sequence of integers. The transformer predicts them one at a time, left to right, exactly as it would predict a word.

Recovering the action at run time is the reverse, and it is where the cost hides. The model hands you a bin index; you map it back to a real number, usually the center of the bin. That center is off from the true value by up to half a bin width, always. With 256 bins over a unit range, half a bin is about $1/512 \approx 0.002$ in normalized units. For coarse pick-and-place at a few hertz, that quantization error disappears under the noise of the robot itself and nobody notices. Uniform binning is simple, it round-trips predictably, and for RT-1's task distribution it was good enough. The hands-on exercise at the end of this chapter has you build exactly this tokenizer and measure its reconstruction error on a real teleop dataset, so the number stops being abstract.

A worked micro-example makes the round trip concrete. Suppose the normalized x-command is 0.37. Map $[-1, 1]$ onto bin indices 0 through 255: the index is $\lfloor (0.37 + 1)/2 \times 256 \rfloor = \lfloor 175.4 \rfloor = 175$. The transformer emits the token 175. To decode, take the bin center, $(175 + 0.5)/256 \times 2 - 1 \approx 0.371$. You asked for 0.37 and got back 0.371. The gap is the quantization error, and it is bounded by half a bin no matter what value you started with.

Where uniform bins fall apart

Now change the task. Instead of moving a can across a table at 3 Hz, you are folding a cloth or threading a connector at 50 Hz, and the robot has many joints moving fast and together. Two things break at once.

The first is resolution. High-frequency control means each consecutive action is only slightly different from the last, tiny increments arriving fifty times a second. Slice the range into 256 uniform bins and those tiny increments all fall into the same bin or two adjacent ones. The tokenizer can no longer tell consecutive actions apart; it has quantized away the very signal the dexterous task depends on. You could add more bins, but the vocabulary grows and the model has to learn a much larger classification, and you are fighting the tool instead of using it.

The second problem is length, and it compounds the first. A dexterous episode at 50 Hz has thousands of timesteps, each with many action dimensions. Tokenize per dimension per timestep and the sequence the transformer has to predict becomes enormous. Attention cost grows with sequence length, training slows, and the autoregressive decode at run time, one token at a time, blows the latency budget a real robot lives under. The FAST paper (Pertsch et al., 2025, arXiv:2501.09747) reports that naïve per-dimension, per-timestep binning does not merely degrade on high-frequency dexterous data; on some tasks it fails completely, producing policies that never learn the skill at all. The small idea that carried RT-1 does not survive contact with harder robots.

Compression instead of binning: FAST

The fix in FAST is to stop treating each number in isolation and compress the action sequence first, the way an audio or image codec compresses a signal before storing it. A chunk of consecutive actions is highly redundant, since smooth motion means neighboring values are correlated, and redundancy is exactly what compression eats. FAST runs the action chunk through a discrete cosine transform, the same DCT that sits at the heart of JPEG, which re-expresses the motion as a handful of frequency coefficients. Most of those coefficients are near zero for smooth trajectories, so you keep the few that carry the energy, quantize those, and turn *them* into tokens. Slow, smooth motion collapses to a short token string; only genuinely fast or complex motion needs many tokens.

The payoff is that the token count now tracks the information content of the action, not the raw sample rate. High-frequency data that drowned uniform binning becomes tractable, sequences get shorter, and training speeds up. The FAST authors ship a ready-made tokenizer, FAST+, fit on roughly a million real robot trajectories, meant to be dropped in as a black box across different robots and control rates. Paired with the $\pi 0$ model you will meet in Chapter 13, FAST let an autoregressive VLA scale to thousands of hours of data and match the performance of the diffusion-based action heads from Chapter 10, while cutting training time substantially. That comparison is the one to hold onto: tokenized autoregressive control and continuous diffusion heads are the two live options, and §10.5 and Chapter 13 are where the trade-off gets settled.

Why this small choice ripples so far

Step back and the stakes come into focus. The tokenizer is a contract between the continuous world the robot lives in and the discrete world the transformer thinks in, and everything downstream inherits its terms. Pick uniform bins and you get RT-1's simplicity and RT-2's free reuse of a web-pretrained VLM, at the price of a resolution ceiling that caps how dexterous the policy can get. Pick a compression-based scheme like FAST and you buy back the dexterity, at the cost of a decode step that is no longer a trivial table lookup. Neither is wrong; they answer to different robots. What you cannot do is ignore the choice, because a policy can only ever be as precise as the tokens it is allowed to emit. Get the tokenizer wrong and the fanciest backbone in the world predicts the right bin for an action the bin cannot express.

That is the sense in which a discretization scheme, three lines of code in its simplest form, sets a ceiling on an entire model. The next section pulls the lens back from this one design choice to the model that made it famous, and asks which of RT-1's headline results came from the transformer and the tokens, and which came from the 130,000 demonstrations feeding them.

11.4 What RT-1 changed and what it did not

A paper this influential collects a lot of credit, and not all of it is deserved by the same part of the paper. RT-1 (arXiv:2212.06817) is treated as the moment robot learning turned into a scaling problem, and that reputation is mostly earned. But "RT-1 works" is a claim about several things at once: a transformer backbone, a tokenized action space, a language-conditioned recipe, and 130,000 real demonstrations. Pull those apart and you find that some were durable contributions the whole field kept, one was a placeholder that got replaced within a year, and one was less a technical idea than a demonstration that a certain kind of money buys a certain kind of result. This section does the pulling apart, because if you cannot say which lever moved the needle, you will copy the wrong one into your own robot.

What actually stuck

Start with the piece that outlived everything else: the framing. Before RT-1, the ambient assumption in manipulation was one policy per task, or at best one policy per narrow task family, each trained on its own carefully curated data. RT-1 made a different bet stick, which is that a single network absorbing hundreds of tasks from one big heterogeneous dataset beats a portfolio of specialists, and that its performance keeps climbing as you add data and task variety instead of plateauing. That claim had been made before in spirit. RT-1 backed it with numbers on real hardware across 700-plus instructions, and once the numbers existed the argument was over. Every model in Part 4 inherits this framing. When Chapter 12 talks about OpenVLA or Octo "generalizing," it is standing on the ground RT-1 cleared.

It helps to attach numbers to the claim. RT-1 reported near-ceiling success on the tasks it was trained on and, more tellingly, held up well when the world shifted underneath it: performance stayed high with unfamiliar distractor objects cluttering the scene and with backgrounds and lighting the policy had not been trained in, and it completed a solid majority of instructions it had never seen phrased that way. Those robustness axes, not the raw seen-task score, are what convinced people. A specialist can memorize its way to a high number on tasks it trained on; staying upright when you add a distractor or swap the kitchen is the thing a portfolio of specialists never gave you for free, and it is what breadth of data bought.

The second durable contribution is more concrete: control as next-token prediction over a discretized action space, trained with the same cross-entropy loss a language model uses. Section 11.3 already argued why this matters and where it breaks, so the short version here is that RT-1 proved the trick works at scale on a physical arm. That proof is what let RT-2 (arXiv:2307.15818) fold robot actions into the token stream of a web-pretrained vision-language model a year later without redesigning anything. The tokenization scheme itself got improved on, but the idea that a robot policy could be a plain sequence model with an action vocabulary was RT-1's to establish, and it held.

Third, and easy to overlook because it sounds like plumbing: RT-1 showed you could run a transformer policy on a real robot at a usable rate. A decoder-only transformer over six frames of history is not free, and a robot that thinks at 3 Hz is a robot that knocks the cup over while it deliberates. The TokenLearner compression from §11.2, squeezing each frame's feature grid down to a handful of tokens, was the unglamorous engineering that kept inference inside the control loop. People remember the scaling story and forget that none of it would have mattered if the thing ran too slowly to close a gripper on a moving target. Real-time feasibility on real hardware was part of what RT-1 demonstrated, and it is why the paper reads as an engineering result and not just a benchmark.

What got replaced almost immediately

Now the parts that did not last. The most obvious one is the vision-language conditioning stack. RT-1 embedded instructions with a Universal Sentence Encoder and modulated a FiLM-conditioned EfficientNet, a perfectly reasonable 2022 design that looks quaint by 2023. The problem is the one §11.2 flagged: the language understanding is only as deep as the sentence encoder, which knows about English but nothing about the visual world. Ask RT-1 for something phrased far outside its 700 instructions and the conditioning quietly corrupts perception rather than admitting confusion.

RT-2's whole thesis is a direct response to this. Instead of a small sentence encoder bolted to a vision network trained only on robot data, use a vision-language model that has already digested internet-scale image-text pairs, and let robot control ride on top of that far broader understanding. The gain is not subtle. RT-2 can act on concepts it never saw in any demonstration, "move the banana to the sum of two plus one," because the backbone already knows what a banana is and what addition is from the web, not from teleop. RT-1's conditioning was the right idea executed with the tools available; within a year those tools were the bottleneck, and the field routed around them. If you are building today, do not copy RT-1's perception stack. Copy its framing and use a modern VLM, which is exactly the move Chapter 12 walks through.

The tokenizer is the second casualty, though a gentler one. RT-1's uniform 256-bin, per-dimension scheme was fine for its task distribution and, as §11.3 showed, falls apart on high-frequency dexterous control. It did not get thrown out so much as outgrown; compression-based tokenizers like FAST and continuous diffusion heads (Chapter 10) now cover the regimes uniform binning cannot reach. Call it a contribution that aged into a special case.

The uncomfortable part: how much was the data?

Here is the question nobody wants on the first slide. RT-1's headline generalization and robustness scaled with the 130,000 demonstrations behind it. So how much of "RT-1 works" is the architecture, and how much is just 17 months of a 13-robot fleet teleoperating in real kitchens? The honest answer is that the paper cannot fully separate them, and neither can you, because there is no ablation where someone trained RT-1's exact architecture on a tenth of the data and a competing architecture on all of it under matched conditions. What the ablations do show is suggestive: RT-1's performance degrades gracefully as you cut data diversity and task count, and it degrades hard when you cut them a lot, which is the signature of a model whose competence is downstream of its dataset breadth more than its parameter count.

The strongest evidence that data was doing much of the heavy lifting arrived after RT-1, in the Open X-Embodiment project (arXiv:2310.08864). That effort pooled data from many labs and robots into one corpus and retrained RT-1-style and RT-2-style models on it, the "RT-X" models, and found that a policy trained on the pooled multi-robot data outperformed the same architecture trained on any single robot's data, sometimes by large margins, including positive transfer across different robot bodies. Same architectures, more and more varied data, better results. That is close to a controlled statement that the data axis was underexploited, and it reframes RT-1 in hindsight: not

primarily a clever network, but the first serious demonstration that robot policies obey a data-scaling story at all, plus the industrial pipeline to feed it.

None of this diminishes RT-1. It relocates the achievement. The transformer and the tokens were competent, unsurprising choices; the surprising, expensive, hard-to-replicate part was proving that a real-robot dataset large and diverse enough to make scaling visible could be built and would pay off. That is a contribution most academic labs cannot copy, which is itself an uncomfortable finding, and it is one reason the community pivoted so fast toward shared datasets like Open X-Embodiment: if data is the lever, then pooling data is how everyone without a 13-robot fleet gets to pull it.

What RT-1 did not even attempt

Two limits are worth naming so you do not expect from RT-1 what it never promised. It does not reason. There is no chain of thought, no planning, no decomposition of "make me a sandwich" into steps; RT-1 maps a short instruction and a few frames straight to the next action, and long-horizon structure has to come from somewhere else, which in the original SayCan-style deployments meant a separate high-level planner picking the skills RT-1 then executed. We return to this split, high-level reasoning over low-level control, in Chapter 14 on dual-system architectures.

And it does not invent motor skills. Section 11.2 made this point about conditioning and it applies to the whole model: RT-1 generalizes across phrasings and recombinations of skills it practiced, not to mechanically novel behavior. The language covers the what; the 130,000 demonstrations still had to cover the how. A VLA is a mirror of its demonstration distribution with a language-shaped index on top, and RT-1 was the first model big enough for that description to sound like a strength instead of an excuse.

With RT-1 sorted into what lasted and what did not, one thread is still dangling, the one this whole chapter keeps deferring: the data. We have asserted that scale is where RT-1's competence came from without saying when scale actually starts to pay, and that is the question §11.5 finally takes head-on.

11.5 The data side: when does scale start to pay off

Every chapter so far has leaned on the word "scale" as if it were free. It is not. In language modeling scale mostly means compute, because the text is already sitting on the internet waiting to be crawled. In robotics scale means a person moving a robot arm through a task, one demonstration at a time, and that difference reshapes the entire question. So the honest version of "does scale pay off" is narrower and more useful: given that each demonstration costs real human minutes on real hardware, at what point does collecting more of them start buying you generalization you could not get another way, and at what point are you just paying to memorize?

The scaling law you know does not transfer cleanly

The language-model scaling laws are clean because they hold across many orders of magnitude: loss falls as a smooth power of parameters, data, and compute, and you can read off how to trade one against another. Robot learning has nothing that clean, and the reason is not that roboticists are sloppier. It is that the axis that matters is not the raw count of demonstrations but their diversity, and diversity is much harder to put on the x-axis of a plot.

Consider what "one more demonstration" means. Collect a thousand more pick-and-place demos of the same block on the same table under the same lighting, and your model gets very good at that block on that table and learns almost nothing transferable. Collect a thousand demos spread across new objects, new surfaces, new phrasings of the instruction, and you buy robustness. Same count, wildly different payoff. RT-1's own ablations (arXiv:2212.06817) make the point without any fancy analysis: when the authors cut the number of distinct tasks in the training set while holding the demo count roughly fixed, performance fell off harder than when they cut demos while keeping task variety. The model was living off breadth, not volume. That is the first thing to internalize before you buy a fleet of robots: counting demonstrations is measuring the wrong thing.

The floor, the knee, and the plateau

It helps to picture three regimes, even though the boundaries are fuzzy and dataset-dependent.

Below a certain diversity floor, you are in memorization territory. The model reproduces what it saw and falls over the moment the world shifts, exactly the compounding-error failure from §6.3, because it never saw enough variation to learn what to ignore. Adding more of the same data here does close to nothing for generalization. A specialist policy trained on a narrow task will often beat a "scaled" model in this regime, which is why so many working robots in factories are still narrow specialists and not foundation models. When someone tells you scale did not help them, the usual diagnosis is that they never left the floor.

Then there is a knee, the interesting part, where enough variety accumulates that the model starts inferring the structure behind the tasks instead of the tasks themselves. This is where RT-1's headline results live: the robustness to unseen distractors, backgrounds, and instruction phrasings that §11.4 credited to breadth. You cannot predict exactly where the knee sits for a new problem, and that uncertainty is genuinely annoying, but you can recognize it after the fact because generalization to held-out conditions stops being flat and starts climbing with each increment of diversity.

Past the knee, returns compress again into a slow plateau. More data still helps, though each doubling buys less, and you are increasingly paying for the long tail of rare situations. Nobody has cleanly mapped where robot policies enter this plateau, partly because almost no group has enough data to get there on a single embodiment. RT-1's 130,000 demonstrations, gathered by thirteen robots over roughly seventeen months, are enough to show the knee and not much more. That number is worth sitting with. Seventeen months of continuous fleet operation produced a dataset that a language model would consume before breakfast.

How do you tell which regime you are actually in, before spending another quarter on data collection? The practical test is cheap: hold out conditions, not just trajectories. Split your evaluation so that some objects, some backgrounds, and some instruction phrasings appear only at test time and never in training. Then plot success on that held-out split as you grow the dataset in increments. If the held-out curve is flat while the seen-condition curve is high, you are on the floor and collecting more of the same will not save you; you need variety or a specialist. If the held-out curve has started to climb with each increment, you have found the knee and further diverse data is worth the money. This is the single most useful measurement in the whole enterprise, and it is astonishing how often teams skip it and report only in-distribution numbers, which tell you nothing about whether scale is paying.

Why the demonstrations are so expensive

The economics are the whole story, so it is worth being concrete about them. A single teleoperated demonstration requires a human operator, a working robot, a reset of the scene between attempts, and time. Call it a minute or two of skilled human labor per usable episode once you count failures, resets, and the ones you throw away. There is no equivalent of scraping. You cannot download a manipulation trajectory the way you download a webpage, because the trajectory only exists if someone generated it on hardware that matches, or at least resembles, the robot you intend to deploy.

This is the constraint that bends everything else in the field. It is why simulation is attractive despite the sim-to-real gap (Chapter 7). It is why people chase learning from human video, where the data is abundant but the embodiment does not match, a thread we pick up in Chapter 18. And it is the direct cause of the two dominant strategies for getting scale without personally collecting millions of demos: pool everyone's data, or borrow scale from a different modality entirely.

Strategy one: pool the data (Open X-Embodiment)

If your lab cannot afford to reach the knee alone, the obvious move is to combine your data with everyone else's. That is precisely what Open X-Embodiment (arXiv:2310.08864) did. The project pooled robot data contributed by many institutions into a single corpus spanning more than twenty distinct robot embodiments and on the order of a million trajectories, then retrained RT-1-style and RT-2-style architectures on the mixture, producing the "RT-X" models.

The result matters more than the corpus size. A policy trained on the pooled multi-robot data generally outperformed the same architecture trained on any single contributor's data alone, and the authors observed positive transfer across different robot bodies, meaning data collected on one arm improved performance on a mechanically different one. Read that against the diversity argument and it lands hard: pooling does not just add demo count, it adds exactly

the kind of variety that pushes a model past the floor and toward the knee, and it does so cheaply for any individual participant because everyone contributes a slice and receives the whole. Open X-Embodiment is the field's answer to the expense problem, and it works because diversity, not volume, was the binding constraint all along.

Strategy two: borrow the scale from the web

The second escape hatch is the one that defines everything in Chapter 12, so this section only plants the flag. If robot demonstrations are scarce and web image-text pairs are abundant, start from a model that already digested the abundant data and let the scarce data do only the last mile. That is the entire logic of building a VLA on top of a pretrained vision-language model. The web pretraining supplies the visual and linguistic breadth that would otherwise demand an impossible number of teleop episodes; the robot data only has to teach the mapping from that grounded understanding to motor commands.

OpenVLA (arXiv:2406.09246) is the concrete case. It is a 7-billion-parameter model fine-tuned for control on roughly 970,000 robot episodes drawn from the Open X-Embodiment collection, and it reaches strong generalization on a demonstration budget that would be hopeless if the model were starting from random weights. The robot dataset is large by robotics standards and still tiny next to the text and image corpora behind the backbone. That asymmetry is the point. You are not scaling the robot data to language-model sizes, which is infeasible; you are scaling a different, cheaper axis and spending your precious demonstrations only where nothing else will do.

The two strategies are not rivals; the strongest recent models use both at once. OpenVLA borrows web scale through its backbone and pooled-robot scale through its OX-E training mixture, and the mixture itself is weighted rather than uniform, oversampling the more varied datasets and downweighting the repetitive ones. That weighting is the diversity argument showing up as an engineering knob: given a fixed training budget, you spend gradient steps where the variety is, not where the raw episode count happens to pile up. When you fine-tune one of these models for your own robot in Chapter 16, you are effectively adding one more small, targeted slice to a mixture that already carries most of the breadth you need.

So when does scale start to pay?

Put the pieces together and the answer stops being mystical. Scale starts to pay when the diversity of your data crosses the knee for the distribution you actually care about, and not one demonstration sooner. Below that, you are buying memorization and a specialist would serve you better and cheaper. The knee arrives faster along the diversity axis than the raw-count axis, so a smaller, more varied dataset frequently beats a larger, more repetitive one. And you can move the knee dramatically closer by not starting from scratch: pooling with other robots (Open X-Embodiment) or standing on web pretraining (the OpenVLA move) both reduce how much of your own expensive data you need to spend before generalization switches on.

There is a deflating corollary that the RT-1 story from §11.4 already hinted at. For most people, the winning strategy is not to collect more data. It is to reuse data that already exists, from shared corpora and from pretrained backbones, and to spend a modest, targeted demonstration budget on the specific gap between what those give you and what your robot needs. Chapter 16 turns that into an actual fine-tuning recipe. The scaling question, framed this way, is less "how many demonstrations should I gather" and more "how little of my own data can I get away with collecting, given what I can borrow." That reframing is the practical payoff of this whole chapter, and it sets up the summary in §11.6, where we consolidate the CLIP-to-RT-1 arc into the handful of ideas worth carrying into the rest of Part 4.

11.6 Summary

Chapter 11 assembled the recipe that the rest of Part 4 keeps reusing at larger scale. A vision-language-action model is a policy that reads a camera image and a sentence and emits an action, and the reason any of them work is that the expensive half of the problem, understanding what the camera sees and what the sentence asks for, was paid for somewhere other than a robot lab. CLIP paid for it with web images. The robot's own scarce demonstrations then only have to teach the last mile, from language-aware perception to motor commands. Everything named in this chapter, and most of what gets named in Chapters 12 through 14, is a variation on where the inherited understanding

comes from and how the action step is bolted onto it. The chapter's job was to make you fluent in the four moving parts of that recipe before the model names start piling up.

The ideas worth carrying forward

Web-scale contrastive pretraining gave robotics a perception stack that already speaks language, and that inheritance is the whole game. §11.1 built CLIP (Radford et al. 2021) from its one matrix: encode a batch of images and captions, make the matched pairs on the diagonal score high and the mismatches score low, and you get an image encoder whose features are laid out in the same geometry as word meanings. The robotics payoff has almost nothing to do with zero-shot classification, the task CLIP was pitched for. It is that a policy starting from a CLIP-style front-end already knows mugs and cups and "the thing you drink from" point in similar directions, knowledge bought with 400 million web pairs rather than teleop hours, of which no robot dataset will ever have the equivalent. Robot data does not exist on the web at that scale, so borrowing the perception is not a shortcut; it is the only route.

Conditioning a cloned policy on an instruction turns one network into hundreds of skills, and the language space is what makes new phrasings cheap. §11.2 closed the gap between an aligned encoder and a policy that does something. Plain behavior cloning from Chapter 6 averages contradictory demos into mush; feed the same network a task descriptor c and the contradiction becomes a routing signal instead. BC-Z (Jang et al. 2022) proved the idea by completing tasks it had never seen demonstrated, driven only by a language or video description, and it introduced two mechanisms the field kept: FiLM modulation, which lets the instruction reach deep into perception and change what the vision stack looks for, and a shared-autonomy data loop that grows the dataset exactly where the policy is weakest. RT-1 (arXiv:2212.06817) ran the same recipe at fleet scale, a decoder-only transformer over six frames of history, 700-plus instructions, 130,000 real demonstrations. What conditioning buys is transfer across phrasings and recombinations of practiced skills; what it does not buy is a motor skill the robot never performed, and keeping those two straight is half of reading a VLA paper honestly.

Turning continuous actions into discrete tokens is the choice that lets a transformer drive a robot at all, and it sets a ceiling nobody can ignore. §11.3 pulled apart the move RT-1 slipped past you: discretize each action dimension into 256 bins and predict a token per dimension, so control becomes next-token prediction with the same cross-entropy loss a language model trains on. Two things follow. The architecture reuses cleanly, which is why RT-2 could fold actions into a web-pretrained VLM's token stream without redesigning anything, and a distribution over bins sidesteps the multimodality trap from §10.4, since it can put mass on the left-turn bin and the right-turn bin and leave the disastrous average between them empty. The catch is resolution. Uniform bins are fine for coarse pick-and-place at a few hertz and fall apart on 50 Hz dexterous control, where consecutive actions collapse into the same bin and the token sequence grows past what attention can chew. Compression-based tokenizers like FAST (Pertsch et al. 2025, arXiv:2501.09747) run the action chunk through a DCT and tokenize the frequency coefficients, so token count tracks information content rather than raw sample rate. The tokenizer is a contract between the continuous robot and the discrete transformer, and a policy can only ever be as precise as the tokens it is allowed to emit.

RT-1's real contribution was the framing and the proof that a robot dataset big enough to make scaling visible could be built. §11.4 sorted the credit. What stuck: the bet that one network absorbing hundreds of tasks beats a portfolio of specialists, control as next-token prediction on real hardware, and the unglamorous fact that a transformer policy can run inside a control loop at all, thanks to TokenLearner squeezing each frame to a few tokens. What got replaced within a year: the Universal Sentence Encoder conditioning stack, which RT-2 swapped for a full internet-pretrained VLM, and the uniform tokenizer, which FAST and diffusion heads outgrew. The uncomfortable part is that RT-1's generalization scaled with its 130,000 demonstrations more than with its architecture, and the Open X-Embodiment results (arXiv:2310.08864) that came after, where pooled multi-robot data beat any single robot's data on the same architecture, come close to proving the data axis was the underexploited one all along.

What you should be able to do now

Five things, roughly in the order Part 4 will lean on them.

You should be able to *explain why a robot policy inherits a vision-language backbone instead of learning perception from scratch*, and put a number on the asymmetry: hundreds of millions of web image-text pairs on one side, a five- or-six-figure demonstration count on the other. That gap is the reason the whole VLA program exists, and §11.1 and

§11.5 built it so that when a later model credits its generalization to its backbone, you read the claim as arithmetic rather than branding.

You should be able to *describe language conditioning and state precisely what it does and does not generalize over.* Given a policy conditioned on an instruction embedding, you can say that nearby sentences ask for nearby behaviors, that this transfers across phrasings and skill recombinations, and that it invents no new motor skill the demonstrations never covered. The language covers the what; the demos still have to cover the how.

You should be able to *walk the action tokenization round trip and say where it breaks.* Normalize an action dimension, slice it into bins, emit the bin index, decode to the bin center, and you have a quantization error bounded by half a bin. You should also be able to explain why that scheme survives coarse manipulation and fails on high-frequency dexterous control, and why a compression-based tokenizer moves the ceiling. The hands-on exercise in §11.x has you build the tokenizer and measure the error, so the ceiling stops being a slogan.

You should be able to *separate a model's architectural credit from its data credit.* RT-1 is the worked case: name which results came from the transformer and the tokens, which came from 130,000 teleop episodes, and why the two are hard to disentangle without a matched ablation nobody ran. This habit is the antidote to crediting scale to cleverness, and you will use it on every model in Chapter 12.

You should be able to *tell which data regime you are in before spending another quarter collecting.* Hold out conditions, not just trajectories, plot held-out success as the dataset grows, and read the curve: flat means you are on the memorization floor and a specialist would serve you better, climbing means you have found the knee where diversity starts paying. §11.5 argued this is the single most useful measurement in the enterprise, and that most teams skip it and report in-distribution numbers that say nothing.

Where the chapter has set up the rest of the book

Chapter 11 is the on-ramp to Part 4, so almost everything here hands forward. The largest handoff is to Chapter 12, which takes RT-1's framing and swaps its dated perception stack for full vision-language backbones: RT-2 acting on web concepts it never saw demonstrated, OpenVLA as the open 7B version of the move, Octo with a diffusion head instead of tokens. The head-versus-data reading from §11.4 is what keeps that chapter honest, because it is easy to credit a model's reach to its scale and forget the action head quietly capping its control frequency the whole time.

The tokenization thread from §11.3 lands twice more. It reaches Chapter 13, where $\pi 0$ pairs FAST with flow-matching heads and the discrete-versus-continuous question from §10.5 finally gets settled on real data, and it reaches Chapter 15, where the datasets that feed all of this get inspected in detail. The data argument of §11.5, that diversity rather than raw count is the binding constraint and that pooling or web pretraining moves the knee closer, is the direct setup for Open X-Embodiment in Chapter 12 and for the fine-tuning economics of Chapter 16, where "how little of my own data can I collect" turns into an actual recipe.

One limit named in §11.4 points further out. RT-1 does not reason: it maps a short instruction and a few frames to the next action, with no decomposition of a long task into steps. That gap is the entire premise of Chapter 14 on dual-system architectures, where a slow planner sits on top of a fast policy, and the SayCan-style split RT-1 relied on gets built properly.

What the chapter has not covered

Two omissions, so they do not read as gaps later. The chapter treated the vision-language backbone as a black box you inherit, and said little about how those VLMs are built, trained, or scaled. That is deliberate. Chapter 12 is where the backbone stops being a given and its size, pretraining, and co-training with robot data become the whole argument. Here it was enough to know the backbone exists and carries the perception.

The chapter also stayed almost entirely on discrete, tokenized action, mentioning the continuous alternatives from Chapter 10 only to contrast against them. That was to keep one idea clean: the tokenizer as a contract, and the resolution ceiling it imposes. The rematch between tokenized autoregressive control and continuous diffusion or flow-matching heads is fought in Chapter 13, on the hardware and data where the difference actually shows up.

Chapter 11's contribution to the book's overall argument is to name the recipe that turns web-pretrained understanding into a robot policy, and to be honest about which ingredient is doing the work. The findings to carry forward are that language-aligned perception is inherited rather than learned because robot data cannot reach web scale; that conditioning widens the door of instruction without filling the room of motor skill behind it; that action tokenization is a contract that caps a policy's precision no matter how good the backbone; and that scale pays only once data diversity crosses the knee, which pooling and web pretraining bring closer. With the recipe named and its levers sorted, Chapter 12 can start turning the scale up and arguing about what breaks and what emerges when it does.

§11.x closes the chapter with a hands-on exercise, building RT-1's uniform action tokenizer and measuring its reconstruction error on a real teleop dataset, followed by the full reading list for the chapter.

11.x Hands-on exercise + chapter references

Section 11.3 made a claim you took mostly on trust: chopping a continuous action into 256 bins throws away precision, but the amount it throws away is small enough that a language model's cross-entropy loss can drive a real arm. The way to stop trusting and start knowing is to build the tokenizer, run a dataset through it, and read the reconstruction error off the numbers you produced. That is the exercise the TOC names for this chapter, and it is worth doing even though the tokenizer is fifty lines of NumPy, because the fifty lines force you to confront every decision RT-1 made quietly: how many bins, uniform or not, per-dimension range or global, and what error you are willing to eat at the bin edges. The supporting drills push on the edges §11.3 and §11.5 argued about: where the resolution floor actually bites, and whether more bins is the free lunch it looks like.

Budget about two hours. None of it is GPU time; this chapter's exercise is almost entirely about the data structure between a float and a token, which is where most of the practical pain in early VLAs actually lived.

The data you will tokenize

If you have a teleop dataset of your own, use it. Otherwise the generator below stands in for one: a batch of action chunks, each chunk a short trajectory of 7-DoF end-effector deltas (three for translation, three for rotation, one for the gripper), with the kind of range mismatch across dimensions that real teleop always has. Translation deltas are small and centered near zero; rotations swing wider; the gripper is nearly binary. That mismatch is the whole reason per-dimension binning matters, and a synthetic set that ignored it would hide the lesson.

```
import numpy as np
rng = np.random.default_rng(0)

def fake_teleop(n=4000, horizon=8):
    # 7 dims: dx, dy, dz (small), droll, dpitch, dyaw (wider), gripper (~binary)
    trans = rng.normal(0, 0.02, size=(n, horizon, 3))
    rot   = rng.normal(0, 0.15, size=(n, horizon, 3))
    grip  = rng.choice([-1.0, 1.0], size=(n, horizon, 1))
    grip += rng.normal(0, 0.05, size=grip.shape)   # a little slop
    return np.concatenate([trans, rot, grip], axis=-1)  # (n, horizon, 7)

actions = fake_teleop()
```

Everything that follows runs against this array or your own dataset of the same shape.

Exercise 11.x.1 — Build the tokenizer and measure its round-trip error

This is the headline drill. Write a tokenizer that discretizes each action dimension into B uniform bins, encodes a chunk to integer token IDs, decodes those IDs back to floats, and reports how much the round trip cost. RT-1 used $B = 256$ per dimension; start there.

The one decision that matters more than any other is where the bin edges go. Fit them per dimension, from the data, using a robust range rather than the raw min and max. A single outlier teleop frame will blow your min-max range

wide, making every bin coarse and wasting most of the vocabulary on values that never occur. Clip to a percentile range instead, say the 1st and 99th, and let the tail values saturate at the end bins. That single choice is the difference between a tokenizer that works and one that quietly ruins your gripper precision.

```
class UniformActionTokenizer:
    def __init__(self, data, n_bins=256, lo_pct=1, hi_pct=99):
        flat = data.reshape(-1, data.shape[-1])          # (N, D)
        self.lo = np.percentile(flat, lo_pct, axis=0)    # per-dim edges
        self.hi = np.percentile(flat, hi_pct, axis=0)
        self.n_bins = n_bins

    def encode(self, x):                                 # floats -> token IDs
        z = (x - self.lo) / (self.hi - self.lo)          # to [0, 1]
        z = np.clip(z, 0, 1 - 1e-9)
        return (z * self.n_bins).astype(np.int64)        # (..., D)

    def decode(self, ids):                               # token IDs -> floats
        z = (ids + 0.5) / self.n_bins                    # bin center
        return z * (self.hi - self.lo) + self.lo

tok = UniformActionTokenizer(actions, n_bins=256)
ids  = tok.encode(actions)
recon = tok.decode(ids)
```

Now the measurement. Reconstruction error is not one number; report it per dimension, because the whole point of §11.3 is that a fixed bin count means different physical precision on a 2-centimeter translation range and a 0.6-radian rotation range. Compute per-dimension mean absolute error, and convert it to physical units so it means something: millimeters for translation, milliradians for rotation.

```
mae = np.abs(recon - actions).reshape(-1, 7).mean(axis=0)
print("per-dim MAE:", mae)
print("translation MAE (mm):", mae[:3] * 1000)
print("rotation MAE (mrad):", mae[3:6] * 1000)
```

Read what comes out. Each dimension's MAE should sit near a quarter of its bin width, which is the expected error of rounding a uniform variable to the nearest of B centers. The number you care about is whether that error clears your robot's tolerance. A pick-and-place task forgives a millimeter of slop at the gripper; threading a connector does not. Write one sentence stating, in physical units, the finest task your 256-bin tokenizer could support. That sentence is the resolution floor of §11.3, and you now have it as a measured quantity rather than an assertion.

Exercise 11.x.2 — Sweep the bin count and watch precision buy itself

The obvious fix for "not precise enough" is more bins, and the obvious question is why RT-1 stopped at 256. Turn the knob and find out. Re-tokenize the same dataset at $B \in \{8, 16, 32, 64, 128, 256, 512, 1024\}$ and plot per-dimension MAE against B on a log-log axis.

The curve is a clean power law: error falls roughly as $1/B$, because halving the bin width halves the rounding error. So far more bins look free. They are not, and the cost lives in a place this exercise cannot show you directly but you should name anyway. Each bin is a distinct token the policy must learn to predict, and the training signal for a bin is the number of demonstrations that land in it. Push B to 1024 and most bins are visited by a handful of chunks or none, so the cross-entropy classifier of §11.3 has almost nothing to learn from per bin. The reconstruction error keeps dropping while the *predictability* of the token quietly collapses. RT-1's 256 is a compromise between a floor that is fine enough for tabletop manipulation and a vocabulary small enough that every bin gets seen often. Mark on your plot the point where the round-trip error first clears your task tolerance; that, not the leftmost achievable error, is the bin count you would actually ship.

Exercise 11.x.3 — Break it with a distribution shift

§11.5 argued that a model living below the diversity floor is memorizing, and a tokenizer fit to a narrow slice of behavior has the same failure mode in miniature. Make it visible. Fit the tokenizer on one regime of the data, then run a different regime through it.

Split the synthetic set (or your teleop set) so that the tokenizer sees only small, slow motions during fitting: keep chunks whose translation magnitude sits below the median. Fit `lo` and `hi` on that half. Now encode and decode the *fast* half, the chunks you held out, and recompute per-dimension MAE.

```
speed = np.linalg.norm(actions[..., :3], axis=-1).mean(axis=1)   # per-chunk
slow, fast = actions[speed < np.median(speed)], actions[speed >= np.median(speed)]

tok_slow = UniformActionTokenizer(slow, n_bins=256)
mae_in   = np.abs(tok_slow.decode(tok_slow.encode(slow)) - slow).mean()
mae_out  = np.abs(tok_slow.decode(tok_slow.encode(fast)) - fast).mean()
print("in-distribution MAE:", mae_in, " shifted MAE:", mae_out)
```

The fast chunks saturate at the end bins the slow-fitted range never covered, so their large motions all decode to the same clipped value and the error jumps. This is the tokenizer's version of the compounding-error story from §6.3: fit your discretization to a narrow behavior distribution and it will silently clip anything outside it, which is exactly what happens to a VLA asked for a motion its demonstrations never contained. The lesson RT-1's authors internalized, and the reason §11.5 kept insisting diversity beats volume, is that the range you fit is a promise about the behaviors you expect, and the robot keeps that promise whether or not you meant it to.

Exercise 11.x.4 — Compare against a smarter tokenizer, on paper

You have now felt the two weaknesses of uniform per-dimension binning: it spends bins evenly even where the data is not, and its physical precision is hostage to the widest-ranging dimension in the chunk. Read the FAST paper (Pertsch et al., 2025, arXiv:2501.09747) with those two weaknesses in front of you. FAST replaces uniform binning with a discrete cosine transform of the action chunk followed by byte-pair encoding of the coefficients, which concentrates tokens on the parts of the signal that actually carry information and lets a chunk of many timesteps collapse into far fewer tokens.

No code here; the drill is analytical. In two or three sentences, explain which of your measured failures FAST fixes and which it does not. It attacks the wasted vocabulary of Exercise 11.x.2, since the DCT-plus-BPE stage puts token capacity where the motion has structure instead of spreading it uniformly. It does nothing about the promise-of-range problem from Exercise 11.x.3, because any learned tokenizer still fits its scheme to the behaviors it saw. Getting that distinction right is the payoff of the whole exercise: you can now read a new action-tokenization paper and predict, before the experiments, which of your own measured pains it will and will not relieve. That skill is exactly what Chapter 12 asks of you when the model names start arriving three per section.

Chapter 11 reading list

The works below are cited in §11.1–§11.6, grouped by role. Full bibliographic entries for everything cited in the book live in Appendix E.2; this is the chapter-local subset.

Multimodal pretraining, the inherited half of the recipe

- Radford, A., et al. (2021). "Learning Transferable Visual Models From Natural Language Supervision" (CLIP). *ICML 2021.* The aligned image-text space §11.1 builds the whole VLA recipe on top of.
- Jia, C., et al. (2021). "Scaling Up Visual and Vision-Language Representation Learning With Noisy Text Supervision" (ALIGN). *ICML 2021.* §11.1's evidence that the CLIP result was about scale and noisy web data, not one lab's trick.
- Driess, D., et al. (2023). "PaLM-E: An Embodied Multimodal Language Model." arXiv:2303.03378. §11.1's bridge from a frozen aligned encoder to a policy that reasons in the same space, the idea Chapter 12 scales.

Language-conditioned imitation

- Jang, E., et al. (2022). "BC-Z: Zero-Shot Task Generalization with Robotic Imitation Learning." *CoRL 2021 / PMLR 164*. §11.2's first demonstration that a language-conditioned imitation policy generalizes to unseen task phrasings.
- Brohan, A., et al. (2022). "RT-1: Robotics Transformer for Real-World Control at Scale." arXiv:2212.06817. The center of gravity of this chapter: the tokenized action space Exercise 11.x.1 rebuilds, the Everyday Robots dataset of §11.5, and the scaling framing of §11.4.

Action tokenization and its successors

- Brohan, A., et al. (2023). "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control." arXiv:2307.15818. §11.3's proof that RT-1's action tokens fold cleanly into a web-pretrained VLM's token stream; Chapter 12 in full.
- Pertsch, K., et al. (2025). "FAST: Efficient Action Tokenization for Vision-Language-Action Models." arXiv:2501.09747. The DCT-plus-BPE tokenizer Exercise 11.x.4 reads against your own measurements; the answer to uniform binning's wasted vocabulary.

The data that made scale a real question

- Padalkar, A., et al. (2023). "Open X-Embodiment: Robotic Learning Datasets and RT-X Models." arXiv:2310.08864. §11.5's evidence on where diversity starts to pay off across many robots; the dataset Chapter 12 treats in detail.
- Kim, M. J., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246. §11.5's open checkpoint standing on the RT-1 recipe; the model Chapter 12 opens up component by component.

Chapter summary

Chapter 11 assembled the recipe every later chapter reuses, and this closing exercise made its most quietly consequential ingredient concrete. You can now trace the path from CLIP's aligned image-text space to a policy that follows a typed instruction, and say why the alignment had to be paid for on web data rather than robot data: the demonstrations are too scarce and too expensive to teach a machine what a mug is, so they are spent on the last mile from aligned perception to motor command instead. You can build the action tokenizer at the heart of RT-1, discretizing each dimension into uniform bins, and you have measured its round-trip error in physical units rather than accepting §11.3's claim on faith, which means you can state the finest task a 256-bin scheme supports and why more bins stop helping before the reconstruction error says they should. You can explain why language conditioning unlocked generalization that plain behavior cloning could not, and you can read the diversity-versus- volume story of §11.5 as a statement about which axis of a dataset actually buys robustness. Carry the last skill into Part 4: given a new tokenizer, a new backbone, or a new action head, you can now predict which of the pains you measured here it relieves before you read anyone's experiments. Chapter 12 takes the RT-1 recipe and asks what happens when the frozen half stops being a vision encoder and becomes a full web-pretrained vision-language model, starting with RT-2.

Chapter 12 — Scaling up: RT-2, OpenVLA, and Octo

12.1 RT-2: a VLM that also outputs actions

Chapter 11 ended on an uncomfortable finding: RT-1's competence came mostly from 130,000 real demonstrations, not from anything clever in the network, and a 13-robot fleet teleoperating in kitchens for 17 months is not a lever most labs can pull. RT-2 (arXiv:2307.15818) responds to that finding in the only way that scales. If robot data is scarce and web data is not, stop trying to learn perception and reasoning from teleop, and inherit them from a model that already read the internet. The idea is one sentence long. Take a vision-language model that can already answer questions about images, and teach it to emit robot actions as if they were just more words. That's it. The rest of this section is why that sentence works, what it buys you that RT-1 could never reach, and what it costs at 3 in the morning when the robot has to decide where to move in the next 300 milliseconds.

This is the first section of a chapter about scale, so keep the shape of the argument in view. RT-2 is the proof that a web-pretrained backbone transfers to control. OpenVLA (§12.2) makes that proof open and reproducible, and Octo (§12.3) shows a different action head bolted to the same idea. All three sit on the dataset that §12.4 is about. RT-2 is where the recipe first cooks.

The one trick: actions are strings

Recall from §11.3 how RT-1 handled actions. A 7-dimensional command, three numbers for end-effector translation, three for rotation, one for the gripper, gets each dimension quantized into 256 bins, and the policy predicts the bin with a cross-entropy loss. RT-2 keeps the discretization but changes what the bins are. Instead of a fresh action head with its own 256-way output, it writes each action as a short string of integers and feeds that string through the VLM's existing text vocabulary.

Concretely, an action becomes something like `1 128 91 241 5 101 127`: the first integer is the discretized gripper command, the next six are the position and rotation bins, all in the range 0 to 255. The model already has tokens for those numbers, or is made to, by overwriting the 256 least-frequently-used entries in its tokenizer with dedicated action-bin tokens (this is the PaLI-X variant; the PaLM-E variant reuses number tokens the model already owns). The upshot is that "predict the next action" and "predict the next word" are now the same operation running through the same softmax. No separate regression head, no auxiliary loss, no architectural surgery on the backbone. A robot episode is just a training example where the target output happens to be seven numbers instead of a sentence.

Two backbones carried the paper. One is PaLI-X, a vision-language model in the tens of billions of parameters trained on web image-text and visual question answering. The other is PaLM-E (arXiv:2303.03378), the embodied language model from §11.1 that already interleaves images and text in one token stream. Both were built to caption pictures and answer questions about them. Neither had ever touched a robot. RT-2's contribution is showing that the same weights, lightly retrained, can drive one.

Co-fine-tuning, or how not to lobotomize your VLM

Here's the failure mode that the naive version of this idea walks straight into. Take your beautiful web-pretrained VLM, fine-tune it only on robot trajectories, and within an epoch or two it forgets almost everything it knew. The demonstrations contain no photos of the Eiffel Tower, no trivia about mammals, no arithmetic, so gradient descent happily overwrites all of that with kitchen-table object-picking. You'd end up with an expensive RT-1 that has thrown away the exact web knowledge you paid for. Catastrophic forgetting is the standard name for this, and it's the whole reason a smaller model trained from scratch on robot data was the sensible 2022 choice.

RT-2's fix is co-fine-tuning: during robot training, keep feeding the original web data too, mixing visual-question-answering batches in alongside the robot-trajectory batches. The model never stops practicing "what is in this image" while it learns "what action comes next," so the web competence stays resident. The paper is direct that this matters, reporting that co-fine-tuning beats fine-tuning on robot data alone by a wide margin on the generalization tests, and that the ratio drifts toward more robot data over the course of training. The web knowledge isn't a nice-to-have you preserve out of thrift. It's the ingredient that produces everything interesting in the next subsection, and forgetting it collapses RT-2 back into a worse RT-1.

At deployment there's one more wrinkle worth naming, because it trips people up. A VLM asked to produce free-form text could, in principle, emit "I think the robot should" instead of `1 128 91`. RT-2 constrains decoding so that when the model is acting, only valid action tokens are allowed out. The sampling is restricted to the action vocabulary and the format is fixed at seven numbers, which turns the general-purpose language head into a well-behaved policy without changing any weights.

What web knowledge buys: the emergent skills

Now the payoff, and it's the part that made RT-2 famous. Because the backbone still knows what it learned from the web, RT-2 can act on concepts that appear in zero robot demonstrations. The paper groups these into a few kinds of generalization that RT-1 simply could not do.

Symbol and number understanding: told to "move the banana to the sum of two plus one," RT-2 moves it to the object labeled 3, chaining arithmetic it learned from text with a grounding of digits it learned from images, none of it taught by teleop. Novel object categories: asked to "pick up the extinct animal" from a lineup that includes a toy dinosaur alongside ordinary objects, it picks the dinosaur, because "dinosaur" and "extinct" are linked in its web knowledge and the demonstrations never had to enumerate that. It can pick out a specific person's face, identify a flag, or move an object toward a logo, all riding on associations that live in the VLM and were never in the action data.

The mechanism is the one §11.1 set up. The demonstrations teach the mapping from aligned, language-aware features to motor commands, a small thing, and the enormous prior on what the world contains and how words refer to it comes free from the backbone. RT-1's generalization stopped at recombinations of skills and phrasings it had practiced; RT-2's extends to semantic categories it has never physically manipulated. That is a different kind of generalization, and it's the first time a robot policy exhibited it.

RT-2 also inherits a faint version of chain-of-thought. Prompted to reason in text before acting, some variants can produce an intermediate plan, "to pick up the object that could be used as a hammer, I should grab the rock," and then emit the action, using the language model's reasoning to bridge an abstract instruction to a concrete grasp. It's brittle and slow, and it foreshadows the explicit high-level reasoning that Chapter 14's dual-system architectures make a first-class citizen rather than a prompting trick. Don't oversell it here; RT-2's reasoning is a hint of what's coming, not a solved capability.

What it costs

The price of putting your policy inside a tens-of-billions-parameter VLM is that your policy now runs like a tens-of-billions-parameter VLM. RT-2's largest model is 55B parameters, far too big to run on the robot, so the paper serves it from a cloud TPU and queries it over the network, landing control frequencies in the low single digits to about 10 Hz depending on model size. For the slow tabletop pick-and-place tasks in the paper that's tolerable. For anything needing fast, reactive, contact-rich control it is not, and this latency wall is a running theme through the rest of Part

4: §13's flow-matching heads and §14's dual-system split are both, in large part, answers to "the big VLM is too slow to close a loop at."

The second cost is openness, or the lack of it. PaLI-X and PaLM-E are proprietary, the training data mix is Google's, and nobody outside could download the checkpoint and poke at it. For a field trying to build on a result, an unreproducible result is only half a gift. That gap is precisely the hole OpenVLA (arXiv:2406.09246) was built to fill, which is why §12.2 comes next.

And a quieter limitation, easy to miss under the flashy demos: RT-2's new skills are almost all about *what* to manipulate, not *how*. The web taught it that a dinosaur is extinct and that 2 plus 1 is 3; it did not teach it a new grasp, a new force profile, or a motion it never saw demonstrated. The emergent generalization lives in the semantic, perceptual half of the problem. The motor half is still bounded by the demonstration distribution, same as RT-1, a point the Open X-Embodiment work (arXiv:2310.08864) leans on when it argues that pooling *action* data across robots is the way to widen the motor half too.

Why this section anchors the chapter

RT-2 settled an argument. Before it, "use a big pretrained model as the policy backbone" was a plausible bet; after it, the bet had numbers on real hardware and a set of skills no from-scratch policy had shown. Every model in this chapter and the next assumes that settled result and asks a follow-up question. Can we make it open? Can we make the action head continuous instead of discrete? Can we make it fast enough to deploy? RT-2 doesn't answer any of those well, but it's the reason they became the right questions to ask.

The most reproducible of those follow-ups is the first one, and it's where we go next: OpenVLA takes RT-2's recipe, swaps the closed backbone for open weights, and publishes everything, giving us a checkpoint we can actually take apart component by component.

12.2 OpenVLA: an open-source 7B-parameter VLA

RT-2 settled the argument and then walked off with the evidence. The checkpoint sat on Google's TPUs, the backbone was proprietary, and the training mix was internal. A lab that wanted to build on the result could read the paper and admire the demos, but it could not download the weights, inspect what the fine-tuning had touched, or run its own ablation. OpenVLA (arXiv:2406.09246) exists to close that gap. Kim and colleagues took RT-2's core recipe, rebuilt it on open components, trained it on the public Open X-Embodiment corpus, and released everything: weights, PyTorch training code, and fine-tuning notebooks. The headline number is that this open 7B model beats the closed 55B RT-2-X by 16.5 percentage points of absolute success across 29 evaluation tasks, with seven times fewer parameters. That is the kind of result that reorganizes a field's default starting point, and for most of 2024 and 2025 OpenVLA was the checkpoint people reached for first.

This section does three things. It opens up the architecture so you can name which pieces are frozen, which get fine-tuned, and which are new, because the chapter's learning objective is precisely that you can inspect this checkpoint and account for every component. It explains why a plain discrete action head beat a fancier baseline. And it walks through the fine-tuning and quantization findings, since those are the parts you will actually touch when Chapter 16 has you adapt a VLA to your own robot.

What the checkpoint is made of

OpenVLA is not built on a bespoke network. It sits on Prismatic-7B, an off-the-shelf open vision-language model, and the whole design philosophy is to change as little as possible. Prismatic has the three-part shape that every VLM in §11.1 shared: a visual encoder that turns an image into a set of patch embeddings, a small projector that maps those embeddings into the language model's input space, and a large language backbone that does the actual sequence modeling. The backbone is Llama 2 7B. The projector is a two-layer MLP. The visual encoder is the interesting choice.

Rather than a single vision transformer, Prismatic fuses two pretrained encoders whose features get concatenated: DINOv2, which is strong on spatial and geometric structure, and SigLIP, which is strong on semantic, language-

aligned content. Together they run about 600M parameters. The bet is that a policy needs both halves. It has to know *what* the object is, which is SigLIP's job, and *where* it sits in the frame, which is where DINOv2 earns its place. The paper's ablations credit this fused encoder as one of the reasons OpenVLA generalizes better than a single-encoder baseline would.

The input is deliberately spare. One third-person camera image at 224×224, plus the language instruction as text. The authors tried 384×384 and found no improvement for the extra compute, which is a small but telling result: higher resolution helps generic VLM benchmarks but did not, at least here, help control. No wrist camera, no proprioceptive state vector, no history of past frames. A single picture and a sentence go in; seven numbers come out.

Those seven numbers are the added component. Following RT-2, OpenVLA discretizes each dimension of the 7-DoF end-effector action into 256 bins, with the bin edges set to span the 1st-to-99th percentile of that dimension in the training data rather than its full range, which keeps a few outlier actions from stretching the bins and wasting resolution on motions the robot almost never makes. The clever, cheap trick is how those bins enter the vocabulary. Llama's tokenizer has thousands of tokens; OpenVLA overwrites the 256 least-frequently-used ones (the last 256 entries) and reassigns them as action tokens. Nothing about the architecture changes. The output softmax is Llama's original language head, now occasionally emitting token IDs that happen to mean "translate +3cm in x" instead of a rare subword. Predicting an action is still just predicting the next token, trained with the same cross-entropy loss the language model was born with.

So the inventory reads: Llama 2 7B backbone (pretrained, then fine-tuned), fused DINOv2+SigLIP encoder (pretrained, then fine-tuned), MLP projector (fine-tuned), and 256 repurposed vocabulary slots (the only genuinely new parameters, and there are almost none of them). Pretraining on the 970k robot episodes fine-tunes all of it end to end.

The vision-encoder finding worth remembering

There is one result in the pretraining setup that contradicts standard VLM practice, and it is the kind of detail that separates people who read the paper from people who skimmed it. When you train a vision-language model, the received wisdom is to *freeze* the visual encoder. A frozen encoder keeps the robust, internet-pretrained features intact, and unfreezing it usually costs you generalization. Prismatic itself was trained that way.

OpenVLA does the opposite, and the authors are blunt that fine-tuning the vision encoder during robot training was crucial for good performance. Their reading is that internet-pretrained features capture plenty of high-level semantics but not enough of the fine spatial detail a manipulation policy needs, the precise geometry of a gripper approaching a drawer handle, so the encoder has to adapt to the control task rather than stay locked to its captioning-era weights. If you were going to guess wrong about this checkpoint, this is where you would do it. Note it now, because it comes back when we discuss which layers to unfreeze during your own fine-tuning.

Why a discrete head, and how it stacks up

By 2024 the field already had Diffusion Policy and ACT (Chapter 10) showing that continuous action heads produce smoother, more multimodal trajectories than a discrete classifier can. OpenVLA looked at that and chose the discrete head anyway. The reason is architectural honesty about what the paper was testing. Bolting a diffusion head onto a Llama backbone means the action no longer flows through the language model's own output layer, which breaks the "actions are just tokens" simplicity that lets you reuse the entire VLM training stack unchanged. OpenVLA's contribution is the open recipe and the fine-tuning story, not a new action representation, so it kept the representation dead simple and spent its novelty budget elsewhere. Octo (§12.3) makes the other choice, and comparing the two is the cleanest way to see what a diffusion head buys and what it costs, which is why the chapter puts them back to back.

The simple head was not a handicap. Across the 29-task generalist evaluation spanning multiple embodiments, OpenVLA outscored RT-2-X by 16.5 points absolute, and in the multi-task-from-scratch fine-tuning comparisons it beat Diffusion Policy by 20.4 points. The lesson is not that discrete beats continuous in general; it doesn't, and Chapter 13 is largely about the cases where it loses badly. The lesson is that a 7B web-pretrained backbone trained on nearly a million diverse episodes has enough raw competence that the crudeness of its action encoding stops being the bottleneck, at least for the slow tabletop manipulation these benchmarks measure.

Fine-tuning: what to unfreeze, and what it costs

This is the part of OpenVLA that mattered most for adoption, and it is the second half of §12.1's complaint answered. RT-2 never showed you how to efficiently specialize the model to a new robot; OpenVLA ran the experiment. The authors compared several strategies for adapting the pretrained checkpoint to a new task, and the results form a small decision table you can more or less memorize.

Fine-tuning only the last layer plus the token embeddings: cheap, and bad, landing around 30% success. Freezing the vision encoder and tuning everything else: also weak, near 47%, which is the mirror image of the pretraining finding above; if adapting the visual features matters during pretraining, it matters during fine-tuning too. "Sandwich" fine-tuning, which unfreezes the vision encoder, the token embeddings, and the last layer while leaving the middle of the LLM alone: much better at roughly 62%, and lighter on memory because it never touches the full backbone. LoRA (arXiv:2106.09685), the low-rank adaptation method that inserts small trainable matrices into the frozen backbone: best overall at about 68%, matching sandwich on quality while using far less compute.

LoRA is the recommendation you walk away with. It fine-tunes OpenVLA on a single A100 in a fraction of the time full fine-tuning demands, roughly an 8x compute reduction, and it does so without a quality penalty. That is what makes the "fine-tune a foundation VLA on your own 50 demonstrations" workflow of the chapter exercise realistic on hardware a lab actually owns, instead of a hyperscaler's cluster.

Running it: quantization and the honest latency number

The last open contribution is inference cost. Loaded in bfloat16, OpenVLA needs about 15GB of GPU memory and runs at roughly 6Hz on a single RTX 4090, a consumer card. The paper then shows you can quantize the model to 4-bit precision for serving with no measurable drop in success rate, which pushes it onto smaller GPUs still. Six hertz is worth sitting with. It is fast enough for the deliberate pick-and-place tasks in the benchmarks, and it is nowhere near fast enough for reactive, contact-rich control that needs to close a loop at 50Hz or more. This is the same latency wall RT-2 hit, just at a friendlier price point, and it is exactly the pressure that §13's flow-matching heads and §14's dual-system split are built to relieve. An open 7B model you can run on a gaming GPU is a real gift; it is not a real-time controller.

Where this leaves us

OpenVLA converted RT-2 from a demonstration into infrastructure. Anyone could now download a state-of-the-art generalist policy, see exactly which components were pretrained and which were adapted, fine-tune it on consumer hardware with LoRA, and serve it quantized. The dataset that made this possible, Open X-Embodiment (arXiv:2310.08864), is important enough that §12.4 is devoted to it. But OpenVLA also drew a clean boundary around its own choices: a discrete token head, a single image, no history, six hertz. Each of those was a decision, not a law, and the next model in the chapter reopens the first one. Octo keeps the open, reproducible spirit and asks what changes when the action head is a small diffusion model instead of a token classifier.

12.3 Octo: a generalist policy with a diffusion head

OpenVLA ended on a decision it was honest about calling a decision: a discrete token head, chosen for recipe simplicity, not because discrete beat continuous. Octo (arXiv:2405.12213) is the section where someone takes the other fork. Released within weeks of OpenVLA by a Berkeley-led team, it keeps the open, download-everything spirit and trains on the same Open X-Embodiment pool, but it makes two choices OpenVLA declined. The action head is a small diffusion model rather than a classifier over bins. And the whole architecture is built to be reconfigured after pretraining, so you can bolt on a wrist camera or swap to a robot with a different action space without touching the pretrained weights. If OpenVLA is a big pretrained brain with a fixed set of ports, Octo is a smaller brain designed so you can rewire the ports.

Reading the two side by side is the point. They agree on the dataset and the spirit and disagree on almost everything about the head and the backbone, which makes the comparison the cleanest way in the book to see what a diffusion action head costs and what modularity buys.

Smaller than you expect

The first surprise is the parameter count. OpenVLA is 7 billion parameters riding on a Llama backbone. Octo ships in two sizes, Octo-Small at 27 million parameters and Octo-Base at 93 million, and neither of those has a comma in the wrong place. It is roughly two orders of magnitude smaller. Octo does not inherit a language model that read the internet; it is a transformer trained from scratch on robot data, with language handled by a frozen off-the-shelf text encoder that turns the instruction into a few embeddings. There is no PaLM, no Llama, no web-scale semantic prior baked into the weights.

That is a real limitation and worth stating up front, because it sets expectations for everything that follows. Octo will not pick "the extinct animal" out of a lineup the way RT-2 could, since it never learned what "extinct" means. What it offers instead is a policy that trains fast, runs fast, and fine-tunes on a single GPU in a couple of hours, and an architecture whose flexibility is the actual research contribution. The team's bet was that for a generalist manipulation policy, breadth of robot data and a well-designed interface matter more than a giant frozen language prior. Whether that bet holds depends entirely on the task, and §12.5 comes back to it when we ask what "generalist" is really buying.

The block-wise transformer that makes rewiring possible

Octo's central idea is architectural, so it helps to picture the token stream. Everything the model consumes becomes a token: the language instruction (or a goal image) becomes a small block of task tokens, each observation timestep becomes a block of observation tokens (image patches plus any proprioceptive state), and the model inserts its own learned readout tokens at each timestep, which function a little like the [CLS] token in a vision transformer. Attention runs block-wise. Observation tokens can attend to task tokens and to earlier observations; readout tokens attend to everything before them but nothing attends back to the readouts. The action head hangs off the readout tokens.

Why arrange it this way? Because a block that nothing attends back to can be added or removed without disturbing the rest of the network. This is the mechanism behind the whole "reconfigure after pretraining" claim from the paper's abstract. Suppose you pretrained on third-person camera images and now your robot has a wrist camera. You add a new observation-token block for the wrist stream, initialize its lightweight encoder, and fine-tune, and the pretrained attention over the existing blocks keeps working because the new tokens slot in without rewiring what was already there. Same story for a new proprioceptive input, or a new action space: you swap the small head, keep the transformer trunk. Contrast this with OpenVLA, where the input was fixed at one image and a sentence and changing it meant surgery. Octo pushed the flexibility into the tokenization scheme itself, and the block-wise attention (§8.1) is what lets that flexibility exist without retraining from scratch.

One consequence is that Octo natively supports a short observation history. Because each timestep is its own block, feeding the last two frames instead of one costs you another block and nothing else. OpenVLA's single-frame input was a deliberate simplification; Octo's design makes history nearly free, which matters for tasks where velocity or a partially occluded object needs a couple of frames to disambiguate.

The diffusion head, concretely

Now the part that gives the section its title. Instead of predicting an action token, Octo's readout tokens feed a diffusion head that generates a continuous action chunk, a short sequence of future actions, by iterative denoising. If §10.2 is fresh, this is Diffusion Policy's action head grafted onto a generalist transformer trunk. If it isn't, here is the shape of it. The head starts from a vector of pure Gaussian noise the same size as the action chunk. A small denoising network, conditioned on the readout token's embedding, is called repeatedly, and each call nudges the noisy vector a step closer to a clean action; after a fixed number of steps you read off the result and send it to the robot. Training is the reverse: add noise to a demonstrated action at a random level, ask the network to predict the noise, minimize the error. No bins, no discretization, no percentile edges to tune.

What does this buy over OpenVLA's 256 bins? Two things the discrete head structurally cannot do well. First, resolution: a diffusion head outputs a real-valued vector, so it never rounds a delicate motion to the nearest of 256 levels, and the smoothness that §10.4 flagged as diffusion's advantage carries over. Second, multimodality. When a demonstration set contains two valid ways to do a task, say, reaching around the left or the right side of an object, a classifier trained with cross-entropy tends to smear probability across both and can emit an averaged, invalid action

straight down the middle. A diffusion head samples from the distribution instead of averaging it, so it can commit to one mode. This is exactly the behavior Chapter 10 built up, now doing real work inside a foundation-scale generalist policy rather than a single-task demo.

The cost shows up at inference, and Octo is honest about it. A discrete head produces its action in one forward pass. A diffusion head runs its denoising loop every single control step, so the same action costs several network calls instead of one. Octo keeps this cheap in absolute terms because the trunk is only 93M parameters and the denoising network is tiny, so it still runs comfortably faster than a 7B model, but the per-step multiplier is real and it is the same pressure that §13 addresses head-on when flow matching cuts the number of denoising steps toward one. Diffusion buys expressiveness and charges you iterations.

What the pretraining actually used

Octo was pretrained on 800,000 robot trajectories drawn from Open X-Embodiment (arXiv:2310.08864), which the team describes as the largest robot manipulation pretraining mix assembled at the time. That corpus spans many robot embodiments, camera setups, and action spaces, and Octo's flexible tokenizer is partly a response to that heterogeneity: when your training data comes from dozens of labs with incompatible sensor rigs, a model that can absorb variable inputs is not a luxury, it is a precondition for using the data at all. The data-curation story here is a preview of §12.4 and §15.2, both of which take Open X-Embodiment apart in detail, so I will not reproduce the weighting and normalization mechanics now beyond noting that they exist and that they mattered.

The headline evaluation claim is that Octo serves as an effective initialization for downstream fine-tuning across nine robot setups from four institutions, reaching new observation and action spaces with a couple of hours of training on one GPU. That framing is deliberate and it is different from OpenVLA's. OpenVLA led with zero-shot generalist success rates against RT-2-X; Octo leads with "how good a starting point is this for your robot." The two papers are optimizing for adjacent but distinct notions of what a released checkpoint is for, and it is worth noticing which one matches your situation before you pick a base model in Chapter 16.

Reading Octo against OpenVLA

Put the two next to each other and the trade space of mid-2024 open VLAs comes into focus. OpenVLA: 7B parameters, a web-pretrained Llama backbone, a discrete token head, one fixed image input, strong zero-shot semantic generalization, six hertz on a 4090. Octo: 93M parameters, a from-scratch transformer, a diffusion head, a reconfigurable set of inputs and outputs, weak semantic priors, fast and cheap to fine-tune. Neither dominates. If your task leans on recognizing objects and concepts the robot never manipulated, OpenVLA's backbone is doing something Octo cannot fake. If your task needs smooth, multimodal control and you expect to adapt the model to a nonstandard sensor rig on a modest budget, Octo's head and interface were built for exactly that.

The deeper lesson is that "foundation action model" was not yet one thing in 2024. It named a cluster of designs that disagreed about the two questions this chapter keeps circling: what should the backbone know before it sees a robot, and how should the action leave the network. RT-2 answered "know the web, leave as tokens." OpenVLA kept that and opened it. Octo answered "know robots broadly, leave as denoised continuous chunks, and let me rewire the rest." $\pi 0$ in Chapter 13 will offer a third answer that takes the diffusion-style continuous head seriously enough to rebuild it as flow matching, which is where the latency cost we just flagged finally gets paid down.

Octo also quietly makes the case that the dataset, not the architecture, is the load-bearing part. A 93M model with no language prior held its own as a generalist initialization because it was trained on 800k diverse trajectories, and that is a claim about data doing the heavy lifting. Both models in this chapter so far point at the same corpus, and it has now come up in every section, so the next one stops treating Open X-Embodiment as a footnote and looks directly at the dataset that made all of this possible.

12.4 Open X-Embodiment: the dataset that made all of this possible

Every section of this chapter so far has leaned on the same corpus and then deferred the details. RT-2 was trained on Google's own robot data and then re-created as RT-2-X on this larger pool. OpenVLA drew its 970,000 pretraining trajectories from it. Octo pulled 800,000. Three of the most-cited open VLAs of 2024 share a single ancestor, and it

is time to stop calling it a footnote. Open X-Embodiment (arXiv:2310.08864) is not a model. It is the pile of robot data that made the models in this chapter trainable at all, and the paper that introduced it is really two arguments stapled together: here is a dataset assembled from labs that had never pooled their data before, and here is evidence that training one policy across all of it beats training a separate policy per robot.

What "X-embodiment" is reacting against

The default in robot learning, right up until 2023, was one model per robot per task. A lab collects a few hundred demonstrations on its own arm, trains a policy, publishes, and the data never leaves the building. This is not laziness; it is a consequence of the MDP-to-robot translation problem from §5.5. A Franka Panda and a WidowX and a Google mobile manipulator do not share an action space, a camera placement, a control frequency, or a gripper. Data from one looks like noise to a policy trained on another. So everyone trained in their own silo, and the field had thousands of small datasets that could not be added together.

The X-embodiment bet, borrowed straight from the NLP and vision playbook that §11.1 traced through CLIP, is that heterogeneity is survivable if the pile gets big enough. Pretrain on many robots at once, let the model see a WidowX pushing a block and a Franka opening a drawer and a mobile manipulator picking fruit, and the shared structure of manipulation might transfer across the hardware differences. Nobody knew whether that would work. The whole point of the paper was to find out.

The corpus, by the numbers

The assembled dataset gathers 60 individual robot datasets contributed by 21 institutions, covering 22 distinct robot embodiments. Counting by demonstrated behavior, that comes to 527 skills spread across roughly 160,000 tasks. The trajectory count runs past a million. These are not synthetic rollouts or scripted motions; they are real teleoperated and demonstrated episodes, each one a sequence of observations paired with the actions a human or a controller actually commanded, which makes the entire corpus a behavior-cloning dataset in the sense of §6.2.

The distribution is lopsided in the way real collections always are. A handful of large datasets, chiefly Google's RT-1 data and the Bridge dataset from Berkeley, dominate the trajectory count, while many contributed datasets are small. The Franka arm shows up in more individual datasets than any other robot because it is the common research platform, so "number of datasets per embodiment" and "number of trajectories per embodiment" tell different stories: lots of labs own a Franka, but the biggest single piles came from the mobile manipulators at Google. That imbalance matters for what the RT-X experiments could and could not show, which is where the section turns next.

Assembling this took work that never shows up in a benchmark table. Sixty datasets came in sixty formats. The collaboration converted everything to a shared representation, RLDS on top of TensorFlow Datasets, so that a single data loader could stream a WidowX episode and a Franka episode without special-casing each one. Action spaces were harmonized to a common end-effector convention where possible. None of this is glamorous, and all of it was the precondition for any of the models in this chapter to exist. When §15.2 takes the dataset apart in detail, the plumbing gets its own treatment; for now the thing to hold onto is that "just use Open X-Embodiment" hides months of format-wrangling by dozens of people.

The RT-X result, and why it splits in two

The dataset alone proves nothing. To argue that pooling helps, the collaboration trained two models on the corpus and checked whether cross-embodiment data made them better than the same architecture trained on single-robot data. RT-1-X is the RT-1 architecture from §11.2, a FiLM-conditioned EfficientNet feeding a transformer that emits discretized end-effector actions. RT-2-X is the RT-2 recipe from §12.1, a 55-billion-parameter vision-language model fine-tuned to output those same action tokens. Same two families you already know, now trained on everyone's data at once.

The result does not come out as a single clean win. It splits, and the split is the most instructive part.

On small-scale domains, robots whose own datasets were modest, RT-1-X outperformed the original per-robot method on four of the five datasets tested, with a large average improvement. This is co-training paying off exactly where §11.5 predicted it would. A robot with only a few hundred demonstrations of its own borrows structure

from the million trajectories collected on other hardware, and its policy gets better than anything its own data could have produced in isolation. Scarce data benefits most from the pool.

On large-scale domains, the story inverts. When a robot already had a big dataset, like the Bridge and RT-1 piles, the small RT-1-X model trained on the full mixture did not beat the specialist trained on that robot alone. It had spent too much of its limited capacity absorbing other embodiments to master its own. Only the much larger RT-2-X recovered the win, and it did more than recover it: the 55B model showed skills that neither its own robot data nor the web-pretrained backbone alone had exhibited, capabilities that emerged from the combination. Capacity was the deciding variable. A small model asked to be a generalist pays for breadth by giving up depth; a large enough model does not have to choose.

That two-part finding is the empirical spine under the whole "scale" story of Part 4. Pooling helps unconditionally when data is scarce. When data is abundant, pooling only helps if the model is big enough to hold it all, and the payoff at that size includes emergent behavior. §12.5 takes the word "emergent" and asks what it actually means here, because it is doing a lot of load-bearing work and it is easy to oversell.

Why this dataset, specifically, unlocked the chapter

Return to the three models. OpenVLA and Octo are both, in a real sense, downstream of the RT-X result. Once the collaboration had shown that a single policy trained across 22 embodiments could match or beat specialists, the obvious next move was to make that recipe open and reproducible, and that is precisely what OpenVLA and Octo did within a year. Neither team had to assemble a corpus. Neither had to prove that cross-embodiment pretraining was worth attempting. The dataset was sitting in a public repository in a uniform format, and the argument for using it had already been made. That is what infrastructure does: it moves a question from "is this even possible" to "here is my variation on the known-good recipe."

Consider the counterfactual. Suppose Open X-Embodiment did not exist and you wanted to train OpenVLA. You would first need to contact twenty-one labs, negotiate data sharing, and reconcile sixty file formats before writing a single line of model code, and most groups would give up before the model. The dataset collapses that barrier to a download. This is exactly the role ImageNet played for vision and Common Crawl played for language, and the parallel is not decoration; it is the reason a subfield that had been a thousand silos in 2022 looked like a shared enterprise by 2024.

There is a limit worth stating plainly, because the next chapters live inside it. Open X-Embodiment is overwhelmingly tabletop manipulation. Arms, grippers, objects on surfaces. It is thin on legged locomotion, thin on dexterous multi-finger hands, thin on the whole-body control that Chapter 14's humanoids need. A policy pretrained on this corpus inherits its blind spots, and part of why the dual-system and humanoid models later in Part 4 had to gather their own data is that the tabletop pile does not cover their embodiment. The dataset that unlocked this chapter does not automatically unlock the next one.

What to take from this

Open X-Embodiment is the answer to a question the earlier sections kept raising without resolving: where does the data come from, and does more of it actually help. The answer is that 60 datasets from 21 institutions, harmonized into one format and one loader, gave the field its first shared pretraining corpus for manipulation, and the RT-X experiments on top of it showed that cross-embodiment training helps scarce-data robots outright and helps abundant-data robots only at sufficient scale, where it also buys emergent capability. Every open VLA in this chapter is standing on that result.

The next section presses on the word that keeps sneaking into these summaries. RT-2-X showed "emergent" skills; foundation-model papers reach for the term constantly. Before we build on it any further, §12.5 asks what emergence really denotes in this setting, and what it does not.

12.5 What "emergent" really means in this context

Twice already this chapter has leaned on the word and promised to come back to it. §12.1 said RT-2 shows "emergent skills"; §12.4 said the 55B RT-2-X exhibited capabilities that "emerged from the combination" of web pretraining and pooled robot data. Both times the word was doing real work, and both times it was quietly borrowing credibility from a much larger and much messier debate in the language-model literature. This section pays that debt. What does a foundation-model paper actually claim when it calls a skill emergent, which part of the claim holds up on a robot, and where does the word paper over something we should look at more skeptically?

Start with the honest version, because there is one. Then take apart the oversold version, because there is one of those too.

The word came from language models, and it came with baggage

In the LLM literature "emergent" has a specific technical meaning, and it is worth stating precisely before we import it into robotics. Wei et al. (2022) defined an emergent ability as one that is absent in smaller models and present in larger ones, where the capability appears sharply rather than improving smoothly as you scale parameters or training compute. Their canonical plots show a task sitting at chance accuracy across three orders of magnitude of model size, then jumping to well above chance once the model crosses some threshold. Multi-digit arithmetic, certain reading-comprehension tasks, and a scattering of others behaved this way. The claim that grabbed everyone was the discontinuity: you could not have predicted the big model's competence by extrapolating the small models' failure.

That claim did not go unchallenged. Schaeffer, Miranda, and Koyejo (2023) argued that a good fraction of these "emergent" jumps are an artifact of how the ability gets measured, not a property of the model. Their argument is worth understanding because it transfers directly to robots. Suppose you score a task with exact-match accuracy: the model gets credit only when every token of a five-token answer is right. A model that is steadily, smoothly getting better at each token will still score zero on exact-match until all five tokens cross the line at once, at which point the score leaps. The underlying capability grew continuously; the metric hid that growth and then revealed it all at once. Swap in a metric that gives partial credit, per-token likelihood say, and the sharp jump flattens into a smooth curve. Their paper found this pattern held for most of the tasks people had been calling emergent. The abilities were real. The sharpness, the sense of something appearing from nothing, was often manufactured by the scoring rule.

So the LLM community ended up with two claims wearing one word. The strong claim: new capabilities switch on discontinuously at scale, unpredictably. The weak claim: large models can do things small models can't, and the improvement is often smooth once you measure it sensibly. The weak claim is nearly unarguable. The strong one is contested and, for many tasks, probably wrong. Keep both in mind, because RT-2 and RT-2-X make claims that land on different sides of this line.

What RT-2 actually showed, stated carefully

Go back to the RT-2 skills from §12.1. The model moves a banana to the object labeled "3" when told to put it on the sum of two plus one. It picks the toy dinosaur when asked for the extinct animal. It nudges a can toward a company logo it was never trained to recognize as a target. None of these behaviors appear in the robot demonstrations. All of them work because the vision-language backbone learned the relevant concepts from web text and images before it ever saw a trajectory.

Is that emergence? Under the weak reading, plainly yes: the small from-scratch policy of RT-1 cannot do any of it, and the large web-pretrained policy can. Capability that scales with the backbone, delivered. Under the strong reading, be careful. The dinosaur skill did not switch on at some magic parameter count during robot fine-tuning. It was sitting in PaLI-X the whole time, learned during web pretraining, and RT-2's contribution was routing it through to a motor command. The "emergence," such as it is, happened somewhere in a 55-billion-parameter model's exposure to internet-scale text and images, and it happened for reasons that have nothing to do with robots. What robot fine-tuning added was a thin bridge from that pre-existing knowledge to an action. This matters because it tells you

where to look when the skill is missing: if RT-2 can't identify some object, the fix is almost never more robot data, it's a backbone that knows the object.

There is a cleaner way to say this. RT-2's impressive generalization is inherited, not emergent in the robot-training sense. The web model knows things; the demonstrations teach it to point its knowledge at a gripper. Calling the result emergent is fair as marketing and slightly misleading as mechanism, and §12.1 flagged exactly this when it noted that RT-2's new skills are almost all about *what* to manipulate, never *how*. The motor half of the problem showed no comparable jump. No new grasp emerged. No force profile the demonstrations lacked appeared out of scale. Emergence, in RT-2, is a property of the perception-and-language stack, and it stops at the boundary where physical skill begins.

The RT-X capacity split is the more interesting case

The RT-2-X result from §12.4 is where the word earns closer attention, because it looks more like the strong claim. Recall the split: on robots with large existing datasets, the small RT-1-X trained on the full cross-embodiment mixture failed to beat the specialist, while the 55B RT-2-X not only recovered the win but displayed skills that neither its own robot data nor its web backbone had shown alone. That last clause is the interesting one. The paper is careful with it, and so should we be.

What "emerged" here was combinatorial. RT-2-X could position objects using spatial relations drawn from one robot's data applied to objects it had only manipulated in another robot's data. The capability was not present in any single contributing dataset; it required the model to be large enough to hold all of them at once and compose across them. That is closer to a genuine scale effect than the dinosaur trick, because it depended on model capacity during the pooled training, not on knowledge pre-baked into the backbone. A small model asked to absorb 22 embodiments spent its capacity just memorizing each one and had nothing left for cross-embodiment composition. The big model had headroom, and the composition showed up.

Now apply the Schaeffer skepticism. Was the appearance of this skill actually discontinuous, or does it only look sharp because success on a manipulation task is scored as a binary did-it-or-didn't-it? A robot either completes the pick-and-place or it fails; there is no partial credit for getting 80% of the way to the drawer. Robot evaluation is exact-match by nature, which is precisely the metric family that manufactures apparent sharpness. We genuinely do not know whether RT-2-X's combinatorial skill switched on abruptly or grew smoothly under a friendlier metric, because nobody ran the fine-grained per-subtask measurement that would settle it. The honest position: the capability is real and capacity-dependent, and the *sharpness* of its arrival is unestablished and quite possibly a scoring artifact, exactly as the LLM debate would predict. Chapter 15 comes back to why success-rate metrics distort our picture of what models can do, and this is the first place it bites.

Why the distinction is worth your time

This is not pedantry about a buzzword. How you read "emergent" changes what you do next.

If you believe the strong version, that capabilities appear unpredictably at scale, the rational move is to keep scaling and wait for surprises. If you believe the calibrated version, that large models inherit semantic competence from pretraining and gain combinatorial competence from capacity, then you know two concrete levers. To widen *what* a policy understands, improve or enlarge the vision-language backbone; the robot data barely touches that axis. To widen *how well* it composes across situations, you need both a big enough model and a pretraining pool broad enough to compose from. Neither lever is magic, and neither will conjure a motor skill that no demonstration ever contained. That last point is the one most worth carrying forward: no amount of web knowledge or cross-embodiment pooling has yet produced a genuinely new low-level skill, a grasp or a contact strategy absent from all training data. The generalization is semantic and combinatorial, not sensorimotor.

There is a rhetorical hazard here that the field has not fully escaped. "Emergent" is a word that discourages inquiry; it suggests the capability arrived on its own and need not be explained. Every time a VLA paper reaches for it, the useful question is the deflationary one: which part of the training pipeline actually produced this behavior, and could a smaller, cheaper, more legible model have produced it too? Often the answer is that a specific, nameable ingredient did the work, and the mystery dissolves on inspection.

What to take from this

The word "emergent" in a VLA paper usually points at one of two real things and one thing to be suspicious of. The first real thing is inherited semantic generalization: RT-2 acting on concepts like "extinct" or "the sum of two plus one" because its web-pretrained backbone already carried them, which is capability that scales with the backbone and never touches the motor system. The second is combinatorial capability from capacity: RT-2-X composing across embodiments in ways no single dataset supported, which is a genuine scale effect that a small model could not reach. The thing to be suspicious of is sharpness, the claim that these skills switch on discontinuously, because robot success rates are exact-match metrics and, as Schaeffer et al. showed for language models, that kind of metric routinely manufactures the appearance of a jump where the underlying curve is smooth. Read "emergent" as "capacity-gated and pretraining-inherited," not as "arrived by magic," and the rest of Part 4 stays honest.

§12.6 pulls the whole chapter together: RT-2, OpenVLA, and Octo as three readings of one recipe, all standing on Open X-Embodiment, all sharing the strengths and the ceilings this scaling story implies.

12.6 Summary

Chapter 11 named the VLA recipe and then left it sitting at the scale of one Google fleet. Chapter 12 turned the scale up and watched what happened. The through-line is a single idea seen from four angles: take a model that already understands images and language because it read the internet, teach it to emit actions, and train the result on more robot data than any one lab owns. RT-2 proved the idea works. OpenVLA made it something you can download and inspect. Octo took the other fork on the action head and showed what that costs. Open X-Embodiment is the pile of data all three stand on, and the RT-X experiments run on that pile are where "more data, pooled across robots, beats a specialist" stopped being a hope and became a measured result. The last section did the unglamorous work of pinning down what "emergent" is allowed to mean once the demos are honest about it. The chapter's job was to give you three worked models and one dataset in enough detail that the model names in Chapters 13 and 14 read as variations rather than a fresh alphabet.

The ideas worth carrying forward

A web-pretrained VLM can be taught to output actions as tokens, and that one move inherits perception and reasoning the robot could never learn from teleop. §12.1 built RT-2 (arXiv:2307.15818) from its central trick: put action tokens in the same output vocabulary as words, co-train on web vision-language data and robot trajectories at once, and the model learns to answer "where do I move next" with the same machinery it uses to answer "what is in this image." The payoff is skills RT-1 could not touch. RT-2 moves a banana to the object labeled with the sum of two plus one, picks the extinct animal, aims for a logo it never saw as a target in any demonstration. The cost is the one that recurs all chapter: a 55-billion-parameter model does not run inside a 10 Hz control loop for free, and RT-2 leaned on cloud inference and a small action space to stay real-time. The recipe transfers. The bill for the backbone comes due at inference.

Reproducing RT-2 on open components did not just democratize the result, it beat the closed model at a seventh of the size. §12.2 opened up OpenVLA (arXiv:2406.09246): a 7B model built from a Llama-2 language backbone and fused DINOv2-plus-SigLIP vision features, trained on 970,000 Open X-Embodiment trajectories, with everything released. The headline is that it clears the closed 55B RT-2-X by 16.5 points of absolute success across 29 tasks. The number that reorganized 2024 was not the win, though, it was the parameter ratio behind it, because it said the earlier scale was buying reproducibility and inspectability more than raw capability. OpenVLA also chose a plain 256-bin discrete head on purpose and said so, and it showed that LoRA fine-tuning plus 4-bit quantization put adaptation within reach of a single consumer GPU. That last fact is the one Chapter 16 cashes in.

The action head is a real fork in the road, and Octo took the branch OpenVLA declined. §12.3 read Octo (arXiv:2405.12213) against OpenVLA precisely because they agree on the corpus and the open spirit and disagree on almost everything downstream of the backbone. Octo puts a small diffusion model on the output instead of a bin classifier, which buys the smoothness and multimodality Chapter 10 argued for, and it builds the architecture to be rewired after pretraining, so a new wrist camera or a different action space attaches without disturbing the pretrained weights. It is also much smaller than people expect, tens of millions of parameters, not billions. Put the two side by side and

you get the cleanest read in the book on what a diffusion head costs and what modularity earns, which is why the section refused to pick a winner: they answer different questions.

None of it trains without Open X-Embodiment, and the dataset carries an argument, not just data. §12.4 stopped treating the corpus as a footnote. Open X-Embodiment (arXiv:2310.08864) pooled robot data from labs that had never combined it, reconciling arms that share no action space, no camera placement, no control rate, no gripper. The RT-X experiments on top of it are the payoff: one policy trained across the whole mixture beats the per-robot specialist on robots with lots of data, at least once the model is large enough to hold every embodiment at once. That caveat is the bridge to §12.5, because the small RT-1-X could not reproduce the win and the 55B RT-2-X could, which is a fact about capacity, not about the dataset.

"Emergent" in a VLA paper means one of two real things and one thing to distrust. §12.5 separated them. The first real thing is inherited semantic generalization: RT-2 acting on "extinct" because its backbone already carried the concept from web pretraining, capability that scales with the backbone and never touches the motor system. The second is combinatorial capability from capacity: RT-2-X composing spatial relations across embodiments in ways no single dataset supported, a scale effect a small model cannot reach. The thing to distrust is sharpness, the claim that these skills switch on discontinuously, because robot success is scored exact-match and Schaeffer et al. (2023) showed that metric family manufactures apparent jumps out of smooth underlying curves. Read "emergent" as capacity-gated and pretraining-inherited, never as arrived by magic, and no amount of pooling has yet produced a genuinely new low-level skill.

What you should be able to do now

Five things, in roughly the order the rest of the book leans on them.

You should be able to *explain the RT-2 move in one breath and price it*: a web-pretrained VLM co-trained to emit action tokens inherits perception and language for free from the internet, and pays for that inheritance in inference latency and a cramped action space at deployment. When a later model claims web-scale reasoning, you now ask the follow-up question, which is what it costs to run the thing at control rate.

You should be able to *open up an open VLA checkpoint and account for every component.* Given OpenVLA, you can name the frozen language backbone, the fused vision encoders, the discrete action head that was a deliberate simplicity choice rather than a performance one, and the LoRA-plus-quantization path that makes fine-tuning affordable. This is the exact skill §12.2 was built around, and Chapter 16 assumes you have it.

You should be able to *state what a diffusion action head buys and what it costs, using Octo versus OpenVLA as the matched pair.* Smoothness and multimodality and post-hoc modularity on one side; a more delicate training recipe and heavier per-step inference on the other. You can say why neither is strictly better and which robot properties push you toward one.

You should be able to *explain why pooling robot data beats a specialist, and name the condition.* The RT-X result is not "more data always wins," it is "cross-embodiment pooling wins once the model is large enough to absorb the whole mixture without spending all its capacity memorizing each robot." You can reproduce the RT-1-X-versus-RT-2-X contrast as the evidence.

You should be able to *deflate the word "emergent" on sight.* Faced with a paper that calls a skill emergent, you can sort it into inherited-semantic, capacity-combinatorial, or metric-artifact, and you know that the deflationary question, which specific ingredient produced this behavior, almost always has an answer. §12.5 built this habit so the model zoo of Chapters 13 and 14 does not read as a parade of miracles.

Where the chapter has set up the rest of the book

Chapter 12 is the hinge of Part 4, so most of it hands forward. The largest handoff is to Chapter 13, where the action-head fork from §12.3 gets settled on real hardware. Octo showed diffusion is viable; $\pi 0$ pairs flow-matching heads with FAST tokenization and argues the continuous head is the right default for smooth, high-frequency control, which is the discrete-versus-continuous question from §10.5 finally decided where it matters. OpenVLA's discrete head is the baseline that argument is measured against, so keep it in view.

The RT-2 latency problem from §12.1 is the direct setup for Chapter 14. A big backbone that reasons well but cannot close a fast control loop is exactly the tension dual-system architectures resolve, with a slow VLM planner riding on top of a fast sensorimotor policy. Helix and GR00T N1 are that idea built properly, and the reason they exist is the bill RT-2 could not pay.

The OpenVLA fine-tuning economics from §12.2, LoRA and 4-bit quantization on a single GPU, are the concrete on-ramp to Chapter 16, where "pick a base model and adapt it to your robot without a fleet" turns into a recipe card. And the Open X-Embodiment treatment in §12.4 sets up Chapter 15, which inspects the datasets and benchmarks in detail and asks what has and has not succeeded the pooled-teleop model since 2024.

One thread runs the other way, back into the classical material. Every model in this chapter is behavior cloning at heart, mapping observations to actions from demonstrations, with all the compounding-error exposure Chapter 6 warned about. Scale and web pretraining widened what these policies understand; they did not repeal the imitation-learning failure modes underneath. Chapter 17 on safety and deployment is where that unpaid debt comes back.

What the chapter has not covered

Two omissions worth naming so they do not read as gaps. The chapter treated the action head as a binary fork, discrete tokens versus a diffusion model, and said almost nothing about flow matching as the third option. That was deliberate. Flow matching is the whole argument of Chapter 13, and folding it in here would have muddied the clean OpenVLA-versus-Octo comparison the section was built to deliver. It was enough to establish that the head is a real choice with real trade-offs.

The chapter also stayed almost entirely inside a single forward pass, one observation in, one action or short chunk out. It did not touch task decomposition, long-horizon planning, or any split between a deliberative layer and a reactive one. RT-2's inability to break a long task into steps was noted and then parked, because building the slow-planner-over-fast-policy structure properly is the entire premise of Chapter 14. Here the models were single systems, and keeping them single kept the scaling story legible.

Chapter 12's contribution to the book's argument is to show what happens when the VLA recipe meets scale honestly measured: web-pretrained backbones transfer to control and carry semantic generalization the demos never supplied; open reproductions can beat closed giants at a fraction of the size, which relocates the value of scale from capability to inspectability; the action head is a fork with no free branch; pooled cross-embodiment data beats specialists once the model has capacity to hold it; and "emergent" is a capacity-gated, pretraining-inherited effect, not a miracle, with its apparent sharpness likely an artifact of exact-match scoring. With three models and one dataset understood at this depth, Chapter 13 can zoom into a single lineage, π0 and its flow-matching successors, and ask what changes when the action head goes fully continuous.

§12.x closes the chapter with a hands-on exercise, loading the released OpenVLA checkpoint and running inference on a held-out LIBERO task to see the discrete head and its resolution ceiling in action, followed by the full reading list for the chapter.

12.x Hands-on exercise + chapter references

Chapter 11 gave you a fifty-line tokenizer and no GPU bill. This chapter's exercise is the opposite, and that is the point. Every claim in §12.2 about OpenVLA being the checkpoint people reach for first rests on one property you have not yet tested for yourself: that a 7B model pretrained on somebody else's 970,000 trajectories can be bent to a task it never saw, using a few dozen of your own demonstrations, in an afternoon rather than a month. The exercise the TOC names for Chapter 12 is exactly that test. Fine-tune OpenVLA on a single new task with about fifty demonstrations, then put it next to three baselines that had to learn the task from those same fifty demonstrations with no web pretraining behind them. The gap between the fine-tuned VLA and the baselines is the whole argument of Part 4, rendered as a success-rate number you produced.

Budget half a day, and know where the time goes. Almost none of it is your thinking; most of it is the fine-tune converging and the evaluation rollouts running one slow episode at a time. Set the job going, read the π0 abstract for Chapter 13 while you wait, and come back to numbers.

What you need before you start

You already did the hard setup in Chapter 2: the OpenVLA weights are on disk, LIBERO runs, and the inference loop from §2.3 turns an image and an instruction into a 7-DoF action. This exercise reuses all of it. Pick one LIBERO task suite and hold out a single task as your target, say "put the black bowl in the drawer." Collect or subsample roughly fifty demonstration trajectories for that one task. LIBERO ships demonstrations, so you can pull fifty from the suite rather than teleoperating them yourself, which keeps the exercise about the learning question and not about your joystick skills.

One honest caveat up front. Full fine-tuning of a 7B model wants more memory than a single consumer card has, which is why §12.2 spent a paragraph on LoRA and 4-bit quantization. Use them here. A LoRA adapter over a 4-bit base fits on one 24 GB GPU and is the configuration the OpenVLA authors themselves report for adaptation, so you are not cutting a corner; you are doing what the paper did.

Exercise 12.x.1 — Fine-tune OpenVLA on the held-out task

This is the headline drill. Load the pretrained checkpoint, attach a LoRA adapter, and train on your fifty demonstrations until the action-token accuracy on a small validation split stops climbing. The recipe is deliberately close to the one in the OpenVLA repository, because reproducing a known-good recipe is the right way to learn what its knobs do before you start turning them on your own robot in Chapter 16.

```
from transformers import AutoModelForVision2Seq, AutoProcessor
from peft import LoraConfig, get_peft_model
import torch

proc  = AutoProcessor.from_pretrained("openvla/openvla-7b", trust_remote_code=True)
model = AutoModelForVision2Seq.from_pretrained(
    "openvla/openvla-7b", torch_dtype=torch.bfloat16,
    load_in_4bit=True, trust_remote_code=True)

lora = LoraConfig(r=32, lora_alpha=16, target_modules="all-linear")
model = get_peft_model(model, lora)          # ~110M trainable of ~7.5B total
model.print_trainable_parameters()
```

The single number to watch during training is the fraction of trainable parameters that line prints: on the order of one to two percent of the model. Everything else is frozen, which is the mechanism behind the whole result. You are not teaching OpenVLA what a bowl is or how a drawer opens; the web pretraining and the Open X-Embodiment pretraining already paid for that. You are teaching it to map perception it already understands onto the specific action distribution of this one task. Watch the validation token accuracy climb past roughly ninety percent within a few hundred steps and stall. When it stalls, stop; more steps past that point start memorizing your fifty trajectories rather than generalizing across them, and the rollout success rate will tell on you if you overtrain.

Record two things: wall-clock time to convergence, and peak GPU memory. You will want both when Chapter 16 asks you to estimate what adapting a VLA to a real robot actually costs.

Exercise 12.x.2 — Build the three baselines

The fine-tuned VLA means nothing without something to beat. Stand up three comparisons, each of which strips away one thing the VLA had.

The first is zero-shot OpenVLA: the same checkpoint, no fine-tuning, evaluated directly on your held-out task. This isolates what the fifty demonstrations bought, separate from what pretraining bought. Expect it to do something non-trivial and still fail the task most of the time, because the target task's exact action distribution was not in its pretraining mix in the form your LIBERO setup presents it.

The second is behavior cloning from scratch: a small ResNet-plus-MLP policy, maybe fifteen million parameters, trained only on your fifty demonstrations with no pretraining behind it. This is §6.2's plain BC, and it is the honest

control for the question "did we need a foundation model at all, or would any network trained on fifty demos do?" Keep the architecture unremarkable; the point is the absence of transferred knowledge, not a clever design.

The third is Diffusion Policy, the §10.2 action head trained from scratch on the same fifty demonstrations. This one matters because it is the strongest non-foundation baseline and because it shares an action-head philosophy with Octo (§12.3). If Diffusion Policy from scratch matches your fine-tuned OpenVLA on this narrow task, that is a real and interesting finding about how much a single task actually needs pretraining, and you should report it rather than bury it.

```
# evaluation harness shared by all four policies
def rollout(policy, env, task, n_episodes=50, max_steps=400):
    wins = 0
    for _ in range(n_episodes):
        obs, done, t = env.reset(task=task), False, 0
        while not done and t < max_steps:
            action = policy.act(obs["image"], task.language)
            obs, _, done, info = env.step(action)
            t += 1
        wins += int(info.get("success", False))
    return wins / n_episodes
```

Run the same harness on all four policies so the success rates are comparable. Fifty episodes per policy is the floor; real-robot evaluation variance (§15.4, when you get there) is large enough that fewer than that tells you almost nothing.

Exercise 12.x.3 — Read the numbers honestly

Now put the four success rates in a table and read them against what §12.2 and §12.5 led you to expect. The pattern you are most likely to see, and the one the OpenVLA paper reports across its own tasks, is fine-tuned OpenVLA on top, the two from-scratch baselines well below it, and zero-shot OpenVLA somewhere in between or at the bottom depending on how far your task sits from its pretraining distribution.

The interesting cases are the ones that break that pattern. If Diffusion Policy from scratch beats your fine-tuned VLA, ask whether your task is narrow and repetitive enough that pretrained breadth is dead weight; a foundation model earns its keep on variety, and a single fixed task is the case where it has the least to offer. If zero-shot OpenVLA beats your fine-tune, you almost certainly overtrained the adapter, so revisit where you stopped in Exercise 12.x.1. Write one paragraph explaining your table. The paragraph is the deliverable, not the numbers; anyone can produce a table, and the skill Chapter 12 is trying to build is the reading of it.

Exercise 12.x.4 — Predict a result before you read it

The last learning objective for this chapter is the one that pays off for the rest of the book: read a paper from this era and predict its main result before the experiments section. Practice it now, on a paper you have not studied. Octo (arXiv:2405.12213) is a good choice, because §12.3 told you its design without walking you through its numbers.

Cover the results. From the architecture alone, write down three predictions. Will Octo, with its diffusion head, produce smoother trajectories than token-based OpenVLA on a task that rewards smoothness? Its modular design lets you attach a new camera without retraining the trunk, so does the paper claim cheap adaptation to a new observation setup, and roughly how cheap? Octo is smaller than OpenVLA by an order of magnitude, so where do you expect it to lose, and by how much? Commit the guesses to paper, then uncover the results and score yourself. Getting a prediction wrong is more useful here than getting it right, because a wrong prediction tells you exactly which part of your mental model of these architectures is miscalibrated. That calibration is what carries you into Chapter 13, where the model names arrive faster than any exercise can keep up with, and reading them by their design rather than their leaderboard row becomes the only way to keep your footing.

Chapter 12 reading list

The works below are cited across §12.1–§12.6, grouped by the role they play. Full bibliographic entries for everything cited in the book live in Appendix E.2; this is the chapter-local subset.

The recipe scaled up

- Brohan, A., et al. (2023). "RT-2: Vision-Language-Action Models Transfer Web Knowledge to Robotic Control." arXiv:2307.15818. §12.1's proof that a web-pretrained VLM transfers to control when actions ride in its token stream; the closed-weights result the rest of the chapter reproduces in the open.
- Kim, M. J., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246. The center of the chapter and of this exercise: the open 7B checkpoint of §12.2, the LoRA and 4-bit adaptation recipe you ran, and the discrete-token head you compared against a diffusion baseline.
- Octo Model Team, Ghosh, D., Walke, H., et al. (2024). "Octo: An Open-Source Generalist Robot Policy." arXiv:2405.12213. §12.3's diffusion-head, modular alternative to OpenVLA, and the paper Exercise 12.x.4 asks you to predict before reading.

The data underneath all of it

- Open X-Embodiment Collaboration, Padalkar, A., et al. (2023). "Open X-Embodiment: Robotic Learning Datasets and RT-X Models." arXiv:2310.08864. §12.4's shared corpus: the 970,000 and 800,000 trajectories that OpenVLA and Octo pretrained on, and the RT-X evidence that one policy across many robots beats a policy per robot.

The tools and the caution

- Hu, E., et al. (2021). "LoRA: Low-Rank Adaptation of Large Language Models." arXiv:2106.09685. The low-rank adapter that made Exercise 12.x.1 fit on one GPU; the reason fine-tuning a 7B VLA is an afternoon and not a cluster job.
- Wei, J., et al. (2022). "Emergent Abilities of Large Language Models." *Transactions on Machine Learning Research.* §12.5's source for the honest version of the emergence claim.
- Schaeffer, R., Miranda, B., Koyejo, S. (2023). "Are Emergent Abilities of Large Language Models a Mirage?" *NeurIPS 2023.* §12.5's counterweight: the argument that emergence is often an artifact of a discontinuous metric, the skepticism you should carry into every "emergent" claim in Part 4.

Chapter summary

Chapter 12 took the RT-1 recipe of Chapter 11 and turned the scale up, and this exercise made the payoff of doing so a number on your own screen. You can now explain how RT-2 reuses an off-the-shelf vision-language model as a policy backbone, folding actions into the same token stream the model already uses for words, and why that inheritance is the answer to the data scarcity that limited RT-1. You can open the OpenVLA checkpoint and name every part of it as frozen, fine-tuned, or added, which is exactly what you did when you attached a LoRA adapter and watched two percent of the parameters carry a new task. You can describe how Octo's diffusion action head differs from a token classifier and what its modularity buys, and you tested that contrast directly by putting a from-scratch Diffusion Policy in your baseline table. Above all you can read a robot-learning paper from this era and predict its headline result from its design before the experiments section confirms or embarrasses your guess, a habit that turns the flood of model names in the coming chapters from noise into signal. Chapter 13 spends that habit immediately, on $\pi 0$, where the discrete action tokens you just fine-tuned give way to a flow-matching head built for the smooth, dexterous control that binning could never quite reach.

Chapter 13 — Smooth control: π0 and flow-matching action heads

13.1 The trouble with discrete action tokens

Section 11.3 sold you on action tokens and then, in its last two paragraphs, started taking the sale back. Discretizing an action into bins lets a transformer treat control as next-token prediction, reuses a web-pretrained backbone for free, and dodges the averaging catastrophe that sinks naive regression. All true. All the reason RT-1, RT-2, and OpenVLA are built the way they are. This chapter is about the tasks where that same choice stops paying and starts costing, and about π0 (arXiv:2410.24164), the model that answered the cost by throwing the token head out and regressing continuous actions with a flow-matching objective instead.

Keep the shape of the chapter in mind while we work through the problem. This section is the diagnosis: what specifically breaks when you tokenize actions for a fast, dexterous, multi-stage task. Section 13.2 is π0's architecture, the VLM-plus-flow-expert design that replaces the token head. Section 13.3 explains flow matching itself as a training objective, and §13.4 shows what the continuous head buys on real hardware. So before any of that pays off, we need to be precise about the ceiling we are trying to break, because "discrete is worse" is a slogan, not an argument, and the field spent real effort finding out where the slogan is actually true.

Three costs, and only one of them is obvious

The resolution ceiling from §11.3 is the cost everyone names first, so start there and then keep going, because it is the least interesting of the three.

Uniform binning splits each action dimension into 256 buckets and reports the bin center. The reconstruction error is bounded by half a bin, roughly 0.002 in normalized units, and for a can slid across a table at 3 Hz that error vanishes under the robot's own mechanical slop. Now thread a connector or seat a peg. The motion that distinguishes success from a jam is smaller than a bin, so the tokenizer rounds it away before the policy ever sees it. You can add bins, but the vocabulary grows and the classification the model has to learn gets harder, and you have started fighting your own representation. That much §11.3 already established.

The second cost is the one that quietly decides whether training even converges: sequence length. A transformer that predicts actions token by token has to emit one token per action dimension per timestep. A seven-DOF arm at 3 Hz over a two-second reach is manageable. A two-armed system at 50 Hz over a thirty-second folding task is thousands of timesteps times a dozen-odd dimensions, and now the autoregressive decode is a sequence in the tens of thousands of tokens. Attention cost grows with that length, training slows to a crawl, and the FAST paper (arXiv:2501.09747) reports the sharper failure: on some high-frequency dexterous datasets, per-dimension per-timestep binning does not just degrade, it fails to learn the task at all. The gradient signal drowns in the length of the sequence it has to model. FAST's fix, compressing the action chunk with a DCT before tokenizing, buys back a lot of this, and it is a real fix, not a patch. But notice what it concedes. The moment you need a codec in front of your tokenizer to make the token count track the information in the motion rather than the raw sample rate, the "actions are just words" story has already sprung a leak. You are no longer reusing the language recipe unchanged; you are engineering around it.

The third cost is latency, and it is structural in a way the first two are not. Autoregressive decoding is inherently

sequential: token $k+1$ cannot be sampled until token k is out, because the model conditions on what it already emitted. A full action for a high-DOF robot might be dozens of tokens, and each one is a forward pass through the decoder. Put that inside a 7B-parameter VLM, as OpenVLA (arXiv:2406.09246) does, and every action costs you dozens of serial passes through a large network. RT-2 (arXiv:2307.15818) hit the same wall from the other side, serving a 55B model from a cloud TPU and landing control rates in the low single digits to about 10 Hz. For slow tabletop pick-and-place that is survivable. For contact-rich control where the arm has to react to a slipping grasp inside 20 milliseconds, a policy that thinks one token at a time is not in the running.

Why the costs cluster on exactly the tasks you care about

Here is the pattern worth internalizing, because it explains why the tokenization debate got loud right as the field moved toward capable robots rather than staying an academic footnote. The three costs above are cheap on easy tasks and expensive on hard ones, and they get expensive together.

Slow, coarse, single-stage manipulation, the RT-1 task distribution, is forgiving on all three axes at once: low control rate means short sequences, coarse motion means bin error is invisible, and a few hertz means latency has slack. Dexterous, high-frequency, multi-stage manipulation is unforgiving on all three at once: the fine motion needs resolution the bins do not have, the high rate and long horizon blow up the sequence length, and the tight reaction window leaves no room for serial decoding. There is no regime where tokenization is fine on two axes and only pinched on the third. The costs are correlated because they all scale with the same thing, the demand for fast precise motion over a long horizon, which is precisely the demand a useful home or factory robot places on its policy.

A concrete example fixes this. Take shirt-folding, one of the tasks $\pi 0$ was built to demonstrate. It runs at 50 Hz on a two-arm setup, it lasts on the order of a minute, and the difference between a folded shirt and a wadded one lives in millimeter-scale adjustments of grip and tension. Every one of the three costs bites: the fold precision is below bin resolution, the minute-long 50 Hz two-arm trajectory is an enormous token sequence, and the cloth's dynamics demand reactions faster than a dozen serial decoder passes allow. Try to fold a shirt with a per-dimension uniform tokenizer and you do not get a slightly worse fold; you get a policy that never learns to fold. That is the empirical wall FAST documents, and it is the wall $\pi 0$ was designed to go around rather than climb.

The averaging trap you cannot simply regress your way out of

At this point the obvious rebuttal is: fine, drop the tokens and regress the continuous action directly, the way §11.3's opening tension suggested. Squared error on a real-valued target, no bins, no vocabulary, no resolution ceiling. Why not?

Because naive regression walks straight into the multimodality problem from §10.4, and this is the reason discretization existed in the first place. When the same observation precedes a left turn in some demonstrations and a right turn in others, mean-squared-error regression minimizes its loss by predicting the average, which is straight ahead, into the obstacle both demonstrators avoided. Discretization sidestepped this for free: a distribution over 256 bins can put mass on the left bin and the right bin and leave the middle empty, and sampling gives you one real choice instead of a lethal compromise. Any continuous head that wants to replace the token head has to keep this property. It has to represent a truly multimodal distribution over actions, not collapse to a conditional mean.

That single requirement is why $\pi 0$ does not use plain regression and instead reaches for flow matching. The action head has to be an expressive generative model over continuous actions, one that can carve out "either of these motions, not the thing between them" while still emitting smooth real-valued commands at full resolution. Diffusion policies and the ACT-style approaches from Chapter 10 are one family of answers to exactly this requirement; flow matching, which §13.3 develops, is the one $\pi 0$ adopts, chosen partly because it generates an action in a handful of integration steps rather than the many denoising steps a diffusion sampler needs, which claws back the latency the token head was bleeding.

What this section sets up

So the case against discrete action tokens is not that tokens are bad. RT-1 and OpenVLA are proof they work, and FAST is proof that clever tokenization stretches them further than you would guess. The case is narrower and

sharper: on fast, dexterous, long-horizon tasks the resolution ceiling, the sequence-length blowup, and the serial-decode latency all bite at once, and no amount of tokenizer engineering makes the third one go away, because autoregression is serial by construction. If you want a foundation model that folds laundry and buses tables at 50 Hz, you need an action head that emits a full continuous action chunk in one shot, represents multimodal action distributions honestly, and does it fast enough to close a real control loop.

π0 is one answer to that specification, and it is the answer this chapter takes apart. The next section lays out its architecture end to end: a pretrained vision-language backbone that carries the semantic knowledge from Part 4, feeding through cross-attention into a separate flow-matching "action expert" that does the continuous generation. Watch, as we go, for how cleanly the design splits the two jobs the token head was trying to do at once, understanding the scene and producing the motion, because that split is the whole idea.

13.2 π0's architecture, end to end

Section 13.1 ended with a specification, not a model. A continuous action head has to emit a full action chunk in one shot, represent multimodal action distributions without collapsing to the mean, and run fast enough to close a 50 Hz loop. π0 (arXiv:2410.24164) is Physical Intelligence's answer, and the cleanest way to understand it is to watch how it splits one job the token head was doing badly into two jobs handled by two parts of the network that were built for different things.

The two parts are a pretrained vision-language backbone and a smaller action expert bolted onto it. The backbone reads pixels and language and produces a representation of what is going on in the scene; the action expert reads that representation plus the robot's proprioceptive state and produces the motion. Total parameter count is about 3.3B, which is less than half of OpenVLA's 7B, and yet it drives a two-arm robot at frequencies OpenVLA cannot touch. The parameter count is not the point. The division of labor is.

The backbone: PaliGemma, doing exactly what it already knew how to do

π0 does not train a vision-language model from scratch. It starts from PaliGemma (arXiv:2407.07726), a 3B open VLM that pairs a SigLIP image encoder with a Gemma language model, already pretrained on the kind of image-caption and visual-question-answering data that teaches a network to connect the word "mug" to the round ceramic thing in the corner of a frame. This is the payoff of everything in Part 4. The semantic grounding that RT-2 got by co-finetuning on web data, π0 inherits for free by initializing from a model that already has it.

Feed the backbone three things at each timestep: the camera images (π0 uses multiple views, wrist and overhead), the language instruction as text tokens, and nothing else that it has to reason about semantically. It runs its transformer stack over that multimodal input the same way PaliGemma does when answering a question about a photograph. The output is a sequence of hidden states that encode, roughly, "there is a shirt here, the instruction says fold it, the left gripper is near the collar." No action has been produced yet. The backbone's only job is to understand the situation, and because it was pretrained to understand situations described in images and words, that job costs almost nothing to teach.

Here is the design decision that matters, and it is easy to skate past. The backbone is not asked to output actions at all. RT-2 and OpenVLA overloaded the VLM's own token vocabulary with action bins, which forced the language model to spend its output distribution on motor commands and dragged every action through the full weight of the language decoder, one token at a time. π0 refuses that. The VLM stays a VLM. Motion generation is handed off to a separate set of weights.

The action expert: a second transformer sharing the same attention

The action expert is a smaller transformer, roughly 300M parameters, and the way it connects to the backbone is the part worth being precise about, because "cross-attention into a separate head" is the loose description and it is not quite what π0 does.

π0 uses what the paper calls a mixture of experts arrangement, though it is not the sparse routing you might associate with that phrase. There is one shared sequence of tokens and one shared attention operation running over it, but the

tokens come in two flavors and each flavor is processed by its own set of weights. Image and language tokens go through the PaliGemma weights. Robot-state and action tokens go through the action-expert weights. Both sets of tokens sit in the same attention matrix, so an action token can attend to an image token and vice versa, but the feed-forward and projection parameters that transform each are separate. Think of it as two transformers interleaved into one attention computation rather than two transformers connected by a bridge. The action expert sees everything the backbone saw, through attention, without inheriting the backbone's mass on its own forward pass.

Why bother with the separation instead of one homogeneous stack? Two reasons, and they are the same two reasons the split exists at all. First, the action expert can be small and therefore fast, since it does not need billions of parameters to turn an already-understood scene into a trajectory; the understanding is upstream. Second, the two halves want different numerics. The backbone operates on discrete language and image tokens; the action expert operates on continuous, noisy action vectors under a flow-matching objective, which we get to in §13.3. Forcing both through identical weights would compromise both. Keeping them separate lets each be what it is.

What actually flows through at inference

Walk one control step end to end, because the shape of the data is where the design becomes concrete.

The robot presents its observation: two or three camera images, the current joint positions and gripper state, and the standing language instruction. Images and text enter the backbone and get encoded. The proprioceptive state gets projected into a token and handed to the action expert. Now $\pi 0$ wants to produce not a single action but an action chunk, a block of the next H actions, where H is around 50. At 50 Hz that is roughly one second of motion committed in a single generation. Chunking is not incidental; it is how $\pi 0$ keeps the control loop fast and the motion temporally coherent, and it is the same trick ACT and Diffusion Policy used in Chapter 10 for the same reasons.

The action expert generates that chunk by flow matching. Starting from a chunk of pure Gaussian noise the same shape as the action block, it runs a small number of integration steps, ten or so, each step a forward pass through the action expert conditioned on the backbone's scene representation via the shared attention. Each step nudges the noisy chunk toward a clean trajectory. After the last step you have H real-valued action vectors, at full continuous resolution, no bins anywhere. The robot executes some prefix of the chunk, the observation updates, and the whole thing runs again. Ten small forward passes through a 300M expert is a very different latency budget than dozens of serial token decodes through a 7B decoder, which is precisely how $\pi 0$ buys back the speed §13.1 said autoregression was bleeding.

Notice what did not happen. The 3.3B backbone did not run ten times. It ran once, produced the scene representation, and that representation conditioned all ten cheap expert steps. The expensive part happens once per observation; the iterative part is cheap. That asymmetry is the whole efficiency argument, and it falls straight out of putting the semantics in one place and the motion sampling in another.

The three-part spec, checked off

Return to §13.1's specification and hold the architecture against it.

One shot, full chunk. The action expert emits all H actions of the chunk together, not one dimension at a time, so there is no per-dimension per-timestep sequence to blow up the way FAST (arXiv:2501.09747) documented for tokenizers. Multimodality preserved. Flow matching, developed next section, is a genuine generative model over the action distribution; it can place probability on a left trajectory and a right trajectory and sample one, rather than averaging them into the obstacle, which was the failure that killed naive regression. Fast enough. The heavy backbone runs once per observation and the light expert iterates a handful of times, landing control rates around 50 Hz on real two-arm hardware. All three met, and met by structural choices rather than by tuning.

Where the knowledge lives, and why that matters for Part 5

One consequence of this architecture is worth flagging now because it shapes everything in the fine-tuning chapters. $\pi 0$ keeps almost all of its transferable, task-general knowledge in the PaliGemma backbone and almost all of its embodiment-specific, motor knowledge in the action expert. That is not a tidy accident; it is close to a design goal. When you later fine-tune $\pi 0$ for a new robot, the interesting question becomes how much of the backbone you can

freeze, since the semantics of "fold the shirt" do not change when you swap arms, and how much of the action expert you have to retrain, since the motor commands very much do. We return to exactly this split in Chapter 16, and π0's clean separation is part of why it fine-tunes as gracefully as it does.

There is a data story stacked underneath all of this that the architecture alone does not tell. π0 was trained on roughly 10,000 hours of robot trajectories spanning several embodiments, a corpus Physical Intelligence assembled well beyond what any single academic dataset offers, and the flow-matching head is only as good as the demonstrations it regressed onto. Architecture makes the capability possible; the data makes it real. We take the dataset side apart properly in Chapter 15.

The one piece deliberately left as a black box here is the flow-matching objective itself: how you train the action expert so that ten integration steps turn noise into a clean, multimodal action chunk, and why that beats the many denoising steps a diffusion sampler would need for the same job. That objective is the engine inside the expert, and it is the whole of the next section.

13.3 Flow matching as a control objective

Section 13.2 handed the action expert a black box and promised to open it here. The box is the training objective: how do you teach a 300M transformer so that ten cheap integration steps turn a chunk of Gaussian noise into a clean, multimodal block of robot actions? Section 10.3 already introduced flow matching as one of the action-generation families, so the mechanics will look familiar. What this section adds is the control-specific reason π0 (arXiv:2410.24164) reaches for it over the diffusion alternative from Chapter 10, and the precise form the loss takes when the thing you are generating is an action chunk conditioned on a scene.

Start from the problem flow matching solves, stated without the machinery. You have a pile of expert action chunks from demonstrations. You want a sampler that, given a scene, produces action chunks distributed the way the demonstrations were: mostly near the good motions, spread across the different good motions when there is more than one, and never parked on the average of two that a mean-squared regressor would settle for. Flow matching builds that sampler by learning to move probability mass from an easy distribution you can sample directly, a standard Gaussian, onto the hard distribution you only have samples of.

A velocity field, not a denoiser

The mental picture is a flow of particles. At time $\tau = 0$ every particle is a fresh draw of noise. At time $\tau = 1$ every particle has landed on a real action chunk. In between, each particle follows a smooth path, and the whole population morphs from the Gaussian blob into the shape of the demonstration data. If you knew the velocity of a particle at every position and every time, generating a sample would be trivial: draw noise, then integrate that velocity forward from $\tau = 0$ to $\tau = 1$. The learned object in flow matching is exactly that velocity field, a function $v_\theta(a_\tau, \tau, \text{scene})$ that reads the current noisy chunk, the current time, and the backbone's scene representation, and returns which way to push.

This is the point of contact with §13.2. The action expert *is* v_θ. Every one of the ten integration steps at inference is one evaluation of the velocity field, one forward pass through the expert, conditioned through the shared attention on the scene the backbone already encoded. So training the expert means learning a good velocity field, and the elegance of flow matching is that you can learn it with an ordinary regression loss, no simulation of the flow during training, no adversarial game, no variational bound.

The loss is a regression, and that should worry you until it doesn't

Here is the move that makes flow matching cheap to train, and it is worth slowing down for because it looks like it should reintroduce the averaging trap that killed naive regression in §13.1.

Pick a straight path between noise and data. For a clean action chunk a_1 drawn from the demonstrations and a noise chunk a_0 drawn from the Gaussian, define the interpolated point at time τ as

$$a_\tau = (1 - \tau)\, a_0 + \tau\, a_1.$$

Differentiate with respect to τ and the velocity along this particular path is just $a_1 - a_0$, a constant. That is the target. The training step is: sample a demonstration chunk, sample a noise chunk, sample a random time $\tau \in [0, 1]$, form a_τ, and regress the expert's output onto the constant velocity $a_1 - a_0$:

$$\mathcal{L}(\theta) = \mathbb{E}_{a_1, a_0, \tau}\Big[\, \big\| v_\theta(a_\tau, \tau, \text{scene}) - (a_1 - a_0) \big\|^2 \,\Big].$$

A squared-error loss. The same loss that averages left and right turns into a crash. Why does it not collapse here?

Because the target is conditioned on both the noise draw and the time, and that conditioning is what breaks the tie. When the scene admits two good action chunks, a left fold and a right fold, the demonstrations contain both. During training, a given noisy point a_τ gets pulled toward the left chunk in one sampled pair and toward the right chunk in another. The expert cannot satisfy both with a single vector, so at that shared point it learns the *average* velocity, which does point roughly straight ahead. That sounds like the trap. It isn't, and the reason is the whole trick: the region of noise space that flows toward the left mode and the region that flows toward the right mode are mostly *different regions*, separated by the noise draw a_0. A particle that starts in the left basin has an unambiguous target throughout almost all of its path and only sees the averaged, ambiguous velocity in the thin seam between basins. Integrate from a specific noise sample and you commit to one basin early and ride a clean path into one mode. Draw a different noise sample and you land in the other. The multimodality lives in the noise you draw at inference, not in a distribution the network has to emit in one shot, and that is how a plain regression loss ends up sampling honestly from a multimodal action distribution.

Why π0 uses this instead of diffusion

Diffusion policies (Chapter 10) attack the same multimodality requirement and get there. π0's designers chose flow matching over diffusion, and the reason is the one §13.1 kept circling: steps cost latency, and control has no latency to spare.

A diffusion sampler learns to reverse a noising process, and the standard training recipe produces a curved, stochastic reverse trajectory that wants many small denoising steps to follow accurately. Cut the step count too aggressively and sample quality falls off. Flow matching with the straight-line path above learns paths that are much closer to straight, because the target velocity is literally constant along each training path. Straighter paths integrate accurately with a coarse solver, so you can use a handful of Euler steps instead of dozens of denoising steps and still land on a clean chunk. This is the rectified-flow idea from Liu et al. (arXiv 2209.03003), and it is the property §13.2 was cashing in when it said the expert iterates "ten or so" times. Ten forward passes through a 300M expert is a latency budget a 50 Hz two-arm robot can afford; fifty passes through a diffusion head might not be.

One honest qualification. The straight-*path* construction does not by itself guarantee a straight *flow*, since the learned field averages over many crossing paths and the actual sampling trajectory can bend where basins meet. Rectified flow can be iterated to straighten the field further, and in practice π0 uses a small fixed step count that works well enough without that extra machinery. The engineering claim is not "provably one step," it is "few enough steps to close the loop," and that is what matters on hardware.

One training step, in code

The objective is small enough to write out. This is the core of the π0-style training step for the action expert, stripped to the shape of the computation, with the backbone's scene encoding passed in as a conditioning tensor.

```
def flow_matching_loss(expert, scene_repr, action_chunk):
    # action_chunk: (B, H, action_dim), a clean expert demonstration
    B = action_chunk.shape[0]

    a1 = action_chunk                       # target: clean chunk
    a0 = torch.randn_like(a1)               # source: Gaussian noise
    tau = torch.rand(B, 1, 1)               # random time in [0, 1]

    a_tau = (1 - tau) * a0 + tau * a1       # point on the straight path
```

```
    target_v = a1 - a0                      # constant velocity along it

    pred_v = expert(a_tau, tau, scene_repr) # the action expert IS v_theta
    return ((pred_v - target_v) ** 2).mean()
```

Notice what is absent. No forward simulation of the flow during training, no reverse process to unroll, no discretization of the action into bins. The whole objective is one interpolation and one squared error, and the network learns a field it never had to roll out to be scored on. That is why flow matching trains stably at the scale π0 needs.

Inference inverts the picture with an ODE solver, and the plainest one, forward Euler, is enough to see the shape:

```
def sample_chunk(expert, scene_repr, shape, n_steps=10):
    a = torch.randn(shape)                  # start from noise
    dt = 1.0 / n_steps
    for k in range(n_steps):
        tau = torch.full((shape[0], 1, 1), k * dt)
        a = a + expert(a, tau, scene_repr) * dt   # step along the field
    return a                                # a clean action chunk
```

Ten additions, each gated by one expert evaluation, and the scene representation is computed once outside the loop. Hold this next to OpenVLA emitting dozens of action tokens by serial decode through a 7B backbone and the latency asymmetry from §13.2 stops being an assertion and becomes arithmetic.

The toy version you can hold in your head

The chapter's hands-on exercise builds this on a 2-D toy dataset, and it is worth previewing because two dimensions is where the whole objective becomes visible. Take target points arranged in two clusters, a stand-in for two demonstrated action modes. Train a tiny v_θ with exactly the loss above, no scene conditioning at all. Then plot the velocity field over the plane. You will see arrows that sweep the central noise cloud outward and split it, funneling the left half of the noise toward the left cluster and the right half toward the right, with a visible seam down the middle where the field goes slack because that is the ambiguous region no single velocity can resolve. Drop a hundred noise points and integrate them and they fan into the two clusters in the right proportion. That picture is the entire section: a regression loss that produces a splitting flow, and multimodality that comes out of where each particle started rather than out of anything the network emits at once. Scale the toy up, condition the field on a PaliGemma scene encoding, make the points 50-step action chunks for a two-arm robot, and you have π0's action expert.

Flow matching, then, is not a detail of π0's plumbing; it is the reason the continuous head clears all three bars from §13.1 at once. It emits a full chunk from one noise draw, it keeps genuine multimodality by routing it through the noise, and it samples in few enough steps to run at control rate. What none of this has shown yet is what that combination actually buys on a real robot against the token-head baselines, which is where §13.4 takes the argument next.

13.4 What π0 can do that earlier VLAs cannot — and the π0.5 → π0.6 → π0.7 lineage

The last three sections built π0 (arXiv:2410.24164) from parts. Now the question is whether the parts add up to anything a robot operator would notice, because architecture that only wins on a latency spreadsheet is not worth a chapter. The honest answer is that π0 clears a specific bar earlier VLAs kept tripping over, and the clearest demonstration is a task no RT-2 or OpenVLA demo ever tried to show: folding laundry.

The laundry test

Pull a shirt out of a dryer and it comes out as a wad. There is no canonical shape, no fixed set of grasp points, no way to write down "the state" the way you can for a block on a table. Two operators folding the same shirt will not produce the same trajectory, and the same operator will not repeat their own trajectory twice. The task runs

for a minute or more of continuous manipulation, it demands both arms working in concert, and it fails softly in a hundred small ways rather than crisply in one.

Every property that makes laundry hard is a property the discrete token head from §13.1 handled badly. The wad has no clean discretization, so binning the action space throws away the fine adjustments that separate a caught sleeve from a dropped one. The multimodality is real and unavoidable, so a head that averages competing folds settles on a motion that folds nothing. The horizon is long, so compounding error from autoregressive decoding has a minute to accumulate. π0 folds the shirt. Not perfectly, not every time, but it folds a shirt it has never seen, from a fresh wad, at a success rate that made the original demo notable rather than staged. That is the headline result, and it is worth being precise about why the three components from this chapter are each load-bearing for it.

The continuous action chunk from §13.2 gives the fine motor resolution: no bins between the gripper and the fabric. Flow matching from §13.3 keeps the competing folds as separate modes instead of blending them, so the robot commits to one plausible fold rather than hallucinating the mean of several. And chunking a full second of motion per generation keeps the two arms coordinated over stretches where a per-step token decoder would drift. Take away any one of the three and the shirt stays a wad.

It helps to put a concrete failure next to the success. Point OpenVLA (§12.2) at the same wad and the trouble is not that it produces nothing; it produces a smeared reach. Its discretized head, asked to average the several reasonable ways to grab a crumpled sleeve, emits a command that grabs none of them, and its per-token decode is slow enough that by the time the chunk lands the fabric has already shifted under the first gripper. The failure is quiet and it is the same failure whether the shirt is on the left or the right of the table, because the cause is structural rather than a tuning miss. π0 does not have a better shirt heuristic. It has a head that never had to average in the first place.

Where the gap shows up in numbers

The laundry demo is vivid but anecdotal, so it helps to name the axis where π0 separates from the token-head generation more measurably. On the dexterous, multi-stage tasks the π0 paper reports (folding, bussing a table, assembling a box), the continuous head holds a usable success rate over horizons where discrete-token baselines degrade toward zero. The mechanism is the one §13.1 predicted rather than a surprise: token heads pay a per-decode latency and a per-step error tax that both scale with how long and how finely the task has to be controlled, and laundry maxes out both.

There is a cost worth stating plainly, because π0 is not free. Flow matching needs a lot of clean demonstrations, and π0's roughly 10,000 hours of cross-embodiment teleoperation is far past what a university lab collects in a year. The architecture makes dexterity possible; the data bill is what makes it real, and for most readers that bill is the actual barrier. We take the data side apart in Chapter 15, and the fine-tuning shortcuts that get you a useful π0 without 10,000 fresh hours are the whole of Chapter 16.

π0.5: the same recipe, aimed at houses it has never entered

The first π0 could fold a shirt in a lab that resembled its training kitchens. It could not walk into a stranger's home and clean up, because the scenes, the objects, and the layouts were all off-distribution. π0.5 (arXiv:2504.16054) is Physical Intelligence's answer to that gap, and the interesting part is that it barely changed the action machinery. The flow-matching expert stays. What changed is the training mixture and the way the model reasons about a novel scene before it acts.

π0.5 co-trains on a broader stack than robot teleoperation alone: web data, verbal instructions, high-level subtask predictions, and multi-robot data all feed the same model, so the semantic backbone sees far more of the world than a robot's wrist camera ever will. The payoff the paper reports is generalization to genuinely new homes: a robot cleaning a kitchen and a bedroom it was never trained in, chaining the long sequence of subtasks that a real cleanup demands. Cleaning a bedroom is not one skill. It is picking clothes off a floor, putting them somewhere sensible, clearing a nightstand, and making a bed, and the order is not scripted for the robot in advance. π0.5 handles the sequencing by predicting the next high-level subtask in language and then acting on it with the same flow-matching head, so the model that decides "pick up the shirt now" and the model that produces the arm motion are the same weights reasoning at two levels.

The lesson stacks directly on §11.5's scaling argument. Dexterity came from the action head; open-world generalization came from feeding the backbone data that is not itself robot data. Those are two different scaling stories in one model, and π0.5 is where they combine. Notice what did not change: the action expert and its flow-matching objective are inherited from π0 essentially intact. If you were expecting the leap in generalization to demand a new action head, that expectation is exactly what π0.5 refutes, and the refutation is the point.

π0.6: learning from its own mistakes

Imitation has a ceiling that Chapter 6 named. A policy cloned from demonstrations is bounded by the demonstrations; when it drifts into a state the experts never visited, it has nothing to imitate and no signal telling it how to recover. For a shirt-folder that ceiling shows up as the failure that never quite gets fixed no matter how much more human data you pour in, because the humans rarely demonstrate the specific recovery the robot needs.

π*0.6, Physical Intelligence's late-2025 iteration, attacks the ceiling with a method the paper calls RECAP, "RL with Experience and Corrections via Advantage-conditioned Policies." The recipe starts from a generalist pretrained with offline RL, then improves it on data the robot generates during real deployments: autonomous rollouts, plus expert teleoperated interventions when a human steps in to correct a failure. This is the move Chapter 7 set up: use RL to push past the demonstrator, but do it on top of a policy that already works instead of from scratch, which is the only way RL is tractable on real hardware. The reported gains concentrate exactly where imitation stalls. On the hardest of the tasks the paper studies, folding laundry in real homes, assembling cardboard boxes, and pulling espresso shots on a professional machine, RECAP more than doubles throughput and roughly halves the failure rate against the imitation baseline. That is the difference between a robot that folds twenty shirts and quietly gives up on the twenty-first and one that recovers a fumbled grasp and keeps going. RL-from-experience does not replace the flow-matching head from §13.3; it fine-tunes it against advantage estimates the robot's own runs supply.

The recovery point deserves a concrete shape, because it is where the imitation ceiling bites hardest. Suppose the gripper catches a shirt by one sleeve and the fabric hangs wrong, a state no teleoperator bothered to demonstrate a fix for. An imitation-only policy in that state is guessing, since its whole competence came from states experts actually visited. A policy that has run the task thousands of times on hardware and been rewarded for finishing has, in effect, demonstrated recoveries to itself, and that self-generated data is exactly the part of the distribution human teleoperation is worst at covering. This is why RL-from-experience and imitation are complements rather than rivals: imitation gets the robot into the neighborhood of competence cheaply, and RL cleans up the long tail of failures that no reasonable amount of extra teleoperation would have reached. The cost, and there is always a cost, is that on-robot RL is slow and finicky in ways a simulator hides, which is a thread §13.5 picks up.

π0.7: steerability

By early 2026 the lineage had a generalist that could act in new homes and improve from its own runs. The remaining complaint from operators was control: a capable generalist that does roughly the right thing is hard to correct when you want the shirt folded a particular way, or the table cleared in a particular order. π0.7, subtitled "a Steerable Generalist Robotic Foundation Model with Emergent Capabilities" (Physical Intelligence, 2026), foregrounds exactly that. It keeps the generalist competence while making the model take direction, so a human instruction redirects behavior at a finer grain than "which task" and closer to "how, exactly." The training diet is the now-familiar mix stretched wider, robot demonstrations alongside autonomous data, multimodal web data, and egocentric human video, and the reported headline is out-of-the-box performance on new tasks without fine-tuning, which is the property a foundation model is supposed to have and earlier VLAs mostly did not.

The through-line

Read the four models as one argument and the shape is clear. π0 proved a flow-matching continuous head could do dexterous, long-horizon manipulation that token heads could not. π0.5 showed the bottleneck on generalization was backbone data, not the action head, and fixed it by co-training on the non-robot world. π*0.6 showed the bottleneck on reliability was imitation's ceiling, and fixed it with RL on the robot's own experience. π0.7 turned a capable generalist into a controllable one. None of these threw away the previous step; each kept the flow-matching engine from §13.3 and changed what was stacked around it. That is a healthier pattern than the field's usual habit

of replacing the whole model every six months, and it is a useful thing for a student to notice early: the durable contribution here is the action head, and almost everything after π0 is a data-and-training story wrapped around it.

Which raises the obvious next question. If the flow-matching head is so durable, what does it still not do, and where are the open problems that the π0 lineage has papered over rather than solved? That is §13.5.

13.5 Open questions in continuous-action foundation models

§13.4 sold you on the flow-matching head. A student reading only that section could walk away thinking the continuous-action problem is solved and the rest is data plumbing. It is not solved, and the gaps are not minor engineering to-dos that a bigger cluster will close. Four of them are worth naming precisely, because each one is a place where the field currently has a workaround rather than an answer, and knowing which is which will save you from believing a demo video.

The reward loop nobody wants to run

§13.4 ended on a promissory note: π0.6 fixes imitation's ceiling with reinforcement learning on the robot's own experience, and on-robot RL is "slow and finicky in ways a simulator hides." That parenthetical hides the single largest unsolved problem in the whole lineage.

Here is the shape of it. Imitation learning is cheap to run once you have the data, because you collect demonstrations offline and then train on a GPU with no robot in the loop. RL is the opposite. To improve from experience the robot has to actually attempt the task, thousands of times, on real hardware that wears out, needs resetting between trials, and occasionally throws a shirt on the floor that a human has to pick up before the next attempt can start. A simulated agent learning to fold laundry can run a million episodes overnight; a real bimanual rig folding real shirts runs maybe a few hundred a day if someone babysits it, and the reward signal for "did it fold well" is not something you can read off a sensor. Somebody, or some learned model, has to score each attempt.

So the open question is not "does RL help" (§13.4 says it does) but "how do you make the loop cheap enough to be worth running." Three partial answers exist and none is satisfying. You can build a simulator good enough that policies trained in it transfer to hardware, which works for rigid objects and falls apart on cloth, fluids, and contact-rich assembly where the physics is exactly the part sim gets wrong. You can learn a reward model from human preference so the robot scores itself, which then imports every failure mode of reward hacking. Or you can accept the slow real-world loop and pour engineering at reset automation and throughput, which is what the well-funded labs quietly do and what a university group cannot afford. The honest state of things is that RL-from-experience works when you have a fleet and a budget, and remains out of reach for everyone else.

Where do the next 10,000 hours come from

π0 needed roughly 10,000 hours of cross-embodiment teleoperation. That number is the barrier §13.4 flagged, and it points at a question flow matching does not answer on its own: teleoperation does not scale the way web text scaled for language models. Every hour of robot demonstration costs an operator an hour, a working robot, and a physical setup. There is no equivalent of scraping the internet, because the internet does not contain many recordings of a specific gripper's joint velocities.

Two escape routes are under active work and both belong in the "open" column. The first is learning action-relevant structure from human video, of which the internet has an ocean. V-JEPA 2-AC (arXiv:2506.09985) is the sharpest recent evidence that this can work: Meta's team pretrained a world model on unlabeled video, then post-trained it into a zero-shot pick-and-place policy using under 62 hours of robot interaction data, and it reached roughly 80% on a held-out task where Octo (§12.1) managed about 15%. That is a real result, and it says the video prior carries a lot of the load that teleoperation used to carry alone. It is also pick-and-place, not laundry, so nobody should extrapolate it to dexterous long-horizon control yet.

The second route is distilling human first-person video into robot-usable demonstrations, which is part of how the recent large open models get their hours. LingBot-VLA 2.0 (Ant Group, July 2026) reports training on 60,000 hours built from 50,000 hours of robot teleoperation plus 10,000 hours of distilled first-person human video, and running

one policy across twenty robot morphologies. Whether the human-video fraction actually substitutes for teleoperation, or just pads the total, is the open question, and the papers do not yet isolate it cleanly. The field is betting heavily that video is the way out of the data trap. The bet is not settled.

Nobody agrees on how to measure this

Suppose you build a new continuous-action head and want to claim it beats $\pi 0$. On what? For image classification you report ImageNet accuracy and the field trusts the number. Robotics has no such number, and the substitutes are worse than they look.

The simulation benchmarks a student will meet first (LIBERO, CALVIN, SimplerEnv) run in a simulator, which means they measure performance on exactly the rigid-body, clean-physics tasks that were never the hard part. A policy can top LIBERO and still fumble a real shirt, because the benchmark never tests the cloth deformation, the sensor noise, or the compounding contact dynamics that decide real success. Reporting a sim score for a dexterous policy is a bit like grading a swimmer on a dry-land treadmill: the numbers are precise and mostly beside the point. Real-robot evaluation avoids that trap and creates a new one, because "we folded 8 of 10 shirts in our lab" is not reproducible by anyone who lacks your robot, your shirts, and your kitchen, and small differences in setup swing the number more than the algorithm does.

RoboArena is the most interesting attempt to fix this, borrowing the head-to-head, human-judged comparison format that ranked chatbots and applying it to policies evaluated across many labs, so no single group's shirts define the score. It is early, and it does not solve the deeper problem that dexterous manipulation success is partly a matter of taste, since "folded well" has no crisp definition. Until the field agrees on evaluation, published comparisons between continuous-action models should be read as suggestive, not decisive, and a student reviewing this literature should always ask what "success rate" was measured against before believing the ranking.

Steerable, but not yet controllable, and not yet safe

Flow matching earns its place by keeping genuine multimodality alive: when a shirt affords two reasonable folds, the head represents both instead of averaging them into a fold that grabs nothing (§13.3). That is the right behavior right up until you want a specific fold, at which point the model's willingness to pick either mode becomes the operator's problem. $\pi 0.7$ pushed on steerability (§13.4) and made instructions bite at a finer grain, yet the underlying tension has no clean resolution. A policy that commits hard to your instruction loses the flexible generalization that made it useful; a policy that stays flexible is hard to pin down when you need one exact outcome. The lineage manages the trade-off empirically and cannot yet dial it on purpose.

Sitting under steerability is a harder issue the chapter has mostly deferred: these models offer no guarantees. A flow-matching head is a learned velocity field integrated by an ODE solver, and nothing in that machinery certifies that the resulting motion is safe, stays inside a workspace, or refuses an instruction it should refuse. Worse, the same continuous output that makes the head expressive also gives an attacker a smooth surface to push on. Recent work on adversarial patches for VLA policies shows that a printed pattern in the scene can hijack the action output, and the continuous-action models are not exempt. We take safety and adversarial robustness apart in Chapter 17; for now the point is that "it folds the shirt" and "it is safe to deploy near a person" are different claims, and the second is much further from settled than the demos suggest.

Why does any of this work, and how far does one forward pass reach

The last open question is the one a theorist would have asked first: we do not really know why flow matching is the right objective for control, only that it is empirically better than the alternatives from §13.1. There is no scaling law for action the way there is for language, no clean curve telling you that ten times the data buys a predictable drop in error, so labs discover the returns to scale by spending the money and looking. Chapter 11 gave the scaling intuition; this chapter is where you see how coarse that intuition still is once the output is a robot trajectory instead of a token.

And there is a ceiling built into $\pi 0$'s shape that the next chapter is entirely about. $\pi 0$ reasons and acts in one forward pass through one model, which is what makes it fast, and which also means it cannot think for longer on a harder

problem the way a person pauses before a tricky fold. When the task needs deliberation on a slow clock and reaction on a fast one at the same time, a single head has to serve both, and something gives. Splitting the reasoner from the sensorimotor controller is the response the field converged on, and it is exactly the dual-system design Helix and GR00T N1 use.

Those are the four open problems worth carrying forward: a reinforcement-learning loop too expensive for most labs to run, a data supply that teleoperation cannot fill and video has not yet proven it can, an evaluation culture with no trusted number, and a controllability-and-safety gap that demos paper over. None of them is closed by the flow-matching head that made the rest of the chapter work, which is the useful thing to remember about foundation models for action: the architecture is very good and the surrounding problems are wide open, and confusing the first for the second is how you end up trusting a laundry video too much. §13.6 pulls the chapter's threads together.

13.6 Summary

Chapter 12 left the action head as a two-way fork, discrete bins on one branch and a diffusion model on the other, and then set flow matching aside on purpose so the OpenVLA-versus-Octo comparison stayed clean. Chapter 13 picked the piece back up and spent a whole chapter on the third branch. The through-line is one model taken apart from every side. $\pi 0$ (arXiv:2410.24164) keeps the web-pretrained backbone that Part 4 spent three chapters building, throws out the token head, and regresses continuous actions with a flow-matching objective instead. §13.1 was the diagnosis of why you would want to. §13.2 laid out the architecture that results. §13.3 opened up the flow-matching engine that makes the action expert work. §13.4 showed what the finished model buys on real hardware and traced the $\pi 0.5 \rightarrow \pi 0.6 \rightarrow \pi 0.7$ lineage that grew out of it. §13.5 named the four problems the head does not touch. The chapter's job was to make one continuous-action foundation model legible enough that Chapter 14's dual-system designs read as a response to a specific ceiling rather than a new direction out of nowhere.

The ideas worth carrying forward

Discrete action tokens fail on exactly the tasks a foundation model is supposed to be good at, and one of the three failures cannot be engineered away. §13.1 named the costs: a resolution ceiling from uniform binning, a sequence-length blowup when you tokenize a long high-frequency chunk, and serial-decode latency because autoregression emits one token at a time by construction. FAST (arXiv:2501.09747) compresses the first two and stretches the token head further than you would guess. It does nothing for the third. On a 50 Hz laundry-folding task all three bite at once, and the latency one stays bitten no matter how clever the tokenizer gets. That is the specification $\pi 0$ was built to meet: emit a full continuous action chunk in one shot, represent genuinely multimodal action distributions, and do it fast enough to close a real control loop.

$\pi 0$ works by splitting the two jobs the token head was doing at once. §13.2 walked the architecture. A PaliGemma backbone (arXiv:2407.07726) handles the scene and the language exactly as it already knew how to, and a separate flow-matching action expert, a second transformer sharing the same attention, does the continuous motion generation. The scene gets encoded once, outside the sampling loop; the expert then integrates a noise draw into a clean action chunk in around ten steps. That split is not cosmetic. It puts the transferable, task-general knowledge in the backbone and the embodiment-specific motor knowledge in the expert, which is the fact Chapter 16 leans on when it asks how much of each you can freeze while fine-tuning for a new robot.

Flow matching is a plain regression loss that somehow produces multimodal behavior, and the trick is where the multimodality lives. §13.3 was the part that should worry a careful reader until it doesn't. You train the expert to predict a velocity field, the loss is mean-squared error against a straight-line target between noise and data, and a regression loss is exactly the averaging trap §13.1 warned about. It escapes the trap because the multimodality is routed through the noise, not emitted by the network in one shot: which mode a sample lands in depends on where its noise particle started, so the field can split the noise cloud toward two demonstrated motions with a slack seam between them rather than averaging to the dead center. The 2-D toy in the hands-on exercise makes the splitting field visible. Ten integration steps against a diffusion sampler's many denoising steps is also where the latency $\pi 0$ clawed back becomes arithmetic rather than assertion.

The finished model does dexterous long-horizon manipulation token heads cannot, and its successors kept the head and changed everything around it. §13.4 ran the laundry test and then read four models as one argument. $\pi 0$

proved the flow-matching continuous head could fold laundry and bus tables where discrete heads stalled. π0.5 (arXiv:2504.16054) found the generalization bottleneck was backbone data rather than the action head, and co-trained on the non-robot world to enter houses it had never seen. π0.6 added reinforcement learning on the robot's own experience once imitation hit its reliability ceiling. π0.7 turned a capable generalist into a steerable one. Not one of them replaced the flow head; each stacked a different data-and-training story on top of it. For a student that pattern is worth noticing early, because the durable contribution is the architecture and almost everything after π0 is what you wrap around it.

The head that made the chapter work closes none of the field's hard problems. §13.5 was deliberately deflationary. The RL loop π0.6 uses is too expensive for most labs to run. The next 10,000 hours of training data have to come from somewhere teleoperation cannot fill and video has not yet proven it can, which is where V-JEPA 2-AC (arXiv:2506.09985) comes back into view in Chapter 18. There is no trusted evaluation number and no scaling law for action the way there is for language, so labs find the returns to scale by spending the money and looking. And "steerable" is not the same as controllable or safe. Confusing a very good architecture for solved surrounding problems is how you end up trusting a laundry video more than it has earned.

What you should be able to do now

Four things, in the order the rest of the book uses them.

You should be able to *say precisely when a discrete action head stops paying and why FAST only fixes two-thirds of it.* Given a task, you can predict whether the resolution ceiling, the sequence-length blowup, or the serial-decode latency is the binding constraint, and you know the latency one is structural to autoregression, so no tokenizer removes it. This is the diagnosis §13.1 built, and it is what lets you read a continuous-head paper as an answer to a stated problem instead of a fashion.

You should be able to *draw π0's architecture from memory and say what each half knows.* Backbone for scene and language, action expert for motion, one shared attention, scene encoded once outside the sampling loop. You can point at where the task-general knowledge sits versus the embodiment-specific motor knowledge, which is the exact partition Chapter 16 turns into a freeze-versus-retrain decision.

You should be able to *explain how a regression objective yields multimodal actions.* You can state the velocity-field loss, explain that it looks like the averaging trap and isn't, and locate the multimodality in the noise particle's starting point rather than in anything the network emits at once. You can also say why ten integration steps beats a diffusion sampler's many, and connect that directly to running at control rate.

You should be able to *read the π0 lineage as one argument and separate the durable part from the churn.* π0 fixed the action head; π0.5 fixed backbone data; π0.6 fixed reliability with RL; π0.7 fixed steerability. You can name which bottleneck each release attacked and see that the flow head survived all four, which is the habit §13.4 was building so the model zoo does not read as a parade of replacements.

Where the chapter has set up the rest of the book

Chapter 13 hands forward more than it closes. The largest handoff is to Chapter 14, and §13.5 wrote the setup explicitly: π0 reasons and acts in one forward pass through one model, which is what makes it fast and also means it cannot think longer on a harder problem the way a person pauses before a tricky fold. When a task needs deliberation on a slow clock and reaction on a fast one at the same time, a single head has to serve both and something gives. Splitting the reasoner from the sensorimotor controller is the field's answer, and it is exactly the dual-system design Helix and GR00T N1 use.

The freeze-versus-retrain partition from §13.2 is the on-ramp to Chapter 16. π0's clean separation of backbone knowledge from motor knowledge is part of why it fine-tunes gracefully, and "how much of each do I retrain for my robot" is the question the recipe card in Part 5 answers. The data story stacked under §13.2 and §13.5, roughly 10,000 hours of multi-embodiment trajectories that Physical Intelligence assembled beyond any academic dataset, is the setup for Chapter 15, which takes the dataset side apart and asks what has and has not succeeded the pooled-teleop corpus. And the evaluation complaint from §13.5, that there is no trusted number for this, is the problem Chapter 15 confronts head-on when it gets to real-robot evaluation and its variance.

One thread runs the same way it did in Chapter 12, back into the classical material. π0 is still behavior cloning at heart, a map from observations to actions regressed onto demonstrations, with the compounding-error exposure Chapter 6 warned about. The flow-matching head made the imitation smoother and multimodal; it did not repeal the imitation-learning failure modes underneath, which is why π0.6 needed RL to push past the reliability ceiling in the first place. Chapter 17 on safety is where that unpaid debt returns.

What the chapter has not covered

Two omissions worth naming. The chapter treated π0 as the representative continuous-action foundation model and said little about the diffusion-head alternative it competes with directly, Octo and the ACT-style policies from Chapter 10. That was a choice: §13.1 established that diffusion policies are one valid answer to the same specification, and π0 picked flow matching partly for its few-step sampling, but comparing the two families head-to-head on the same hardware is an evaluation question the book parks until Chapter 15. Here it was enough to develop one lineage in depth.

The chapter also stayed inside a single model doing one forward pass, the same frame Chapter 12 kept. It did not touch task decomposition, an explicit slow planner over a fast controller, or any structured split between deliberation and reaction. That absence is not an oversight; §13.5 named it as π0's built-in ceiling and handed it to Chapter 14, whose entire premise is the two-system structure π0 deliberately does without.

Chapter 13's contribution to the book's argument is to settle the action-head question left open in Chapter 12 by developing the continuous branch in full. A web-pretrained backbone can be paired with a flow-matching action expert that emits a whole action chunk in one shot, keeps genuine multimodality by routing it through the noise, and samples fast enough to run a real control loop, which is what token heads could not manage on fast dexterous tasks. The head proved durable enough that three successive releases changed the data and the training without touching it. And the problems that head does not solve, an unaffordable RL loop, a data supply teleoperation cannot fill, no trusted evaluation number, and a controllability-and-safety gap, are the ones the rest of the book keeps returning to. With one continuous-action model understood at this depth, Chapter 14 can ask what happens when you stop trying to do reasoning and control in the same forward pass.

§13.x closes the chapter with a hands-on exercise, training the 2-D flow-matching toy from §13.3 so the splitting velocity field becomes something you plot rather than something you take on faith, followed by the chapter's full reading list.

13.x Hands-on exercise + chapter references

Chapter 12's exercise cost you a GPU afternoon and a fine-tune. This one costs a couple of minutes on a laptop CPU, and it is arguably the more important of the two, because it makes the single claim the whole chapter rests on into something you can watch happen on a plot. §13.3 argued that a squared-error regression loss produces multimodal, splitting behavior, and it asked you to take the argument partly on faith. The faith ends here. You are going to build the two-cluster toy from §13.3, train a tiny velocity field on it with exactly the π0 training step, and then draw the field over the plane. If the argument in §13.3 is right, you will see the arrows fork: the left half of the noise cloud swept toward one cluster, the right half toward the other, and a slack seam down the middle where no single velocity can decide. If the argument is wrong, you will see the arrows collapse everything to the midpoint, which is the averaging failure §13.1 warned about, and you will know the section oversold its case. It does not, but you should not believe that until the plot shows it.

The TOC names this exercise for Chapter 13 in one line: implement a minimal π0-style training step on a 2-D toy dataset and visualize the learned flow field. Two dimensions is not a toy chosen for convenience. It is the smallest setting where the geometry of the flow is still visible to your eye, and every property that matters at the scale of a two-arm robot folding laundry is already present in it. Scale the point count up to an action chunk, condition the field on a PaliGemma scene encoding (arXiv:2407.07726) instead of nothing, and the code you write below is π0's action expert. That is not a metaphor. It is the same loss.

What you need before you start

Almost nothing. PyTorch, matplotlib, and a CPU. No robot, no simulator, no downloaded checkpoint, no scene conditioning of any kind. The point of stripping all of that away is to isolate the one mechanism §13.3 hinges on, so that when it works you know it was the flow-matching objective doing the work and not some prop the rest of the system was quietly holding up. If you did the fifty-line training loop in §3.3, this will feel like a variation on it, because it is one: sample data, compute a loss, step the optimizer, repeat.

Exercise 13.x.1 — Build the two-mode dataset

Make the data first, because the data is the point. You want a target distribution with two clearly separated modes, so that "did the model learn both or did it average them" is a question your eye can answer. Two Gaussian blobs, one on the left and one on the right, will do.

```
import torch

def sample_data(n):
    # two clusters: left mode at (-2, 0), right mode at (+2, 0)
    left  = torch.randn(n // 2, 2) * 0.3 + torch.tensor([-2.0, 0.0])
    right = torch.randn(n // 2, 2) * 0.3 + torch.tensor([ 2.0, 0.0])
    return torch.cat([left, right], dim=0)
```

Read the two clusters as two demonstrated action modes for the same scene, the left fold and the right fold from §13.3. A mean-squared regressor asked to map one input to this data has exactly one option, the midpoint at the origin, which is a point the data never visits. That is the failure you are about to route around. Plot a scatter of a few hundred samples before you go on, so the target is fixed in your head: two clouds near $x = \pm 2$, nothing at the center.

Exercise 13.x.2 — Write the π0 training step

Now the velocity field. Keep it small, a three-layer MLP is plenty, and give it the two things the field is a function of: the current point a_τ and the current time τ. No scene input, because there is no scene. This is the only place the toy simplifies π0's expert, and it simplifies it by deletion rather than by cheating.

```
import torch.nn as nn

class VelocityField(nn.Module):
    def __init__(self, hidden=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(3, hidden), nn.SiLU(),   # input: (a_x, a_y, tau)
            nn.Linear(hidden, hidden), nn.SiLU(),
            nn.Linear(hidden, 2),              # output: velocity (v_x, v_y)
        )

    def forward(self, a, tau):
        return self.net(torch.cat([a, tau], dim=-1))
```

The training step is the loss from §13.3, transcribed to two dimensions with no changes to its shape. Sample a batch of real points a_1, sample an equal batch of Gaussian noise a_0, pick a random time τ per example, form the point on the straight line between them, and regress the field onto the constant velocity $a_1 - a_0$.

```
model = VelocityField()
opt = torch.optim.Adam(model.parameters(), lr=1e-3)

for step in range(5000):
    a1  = sample_data(256)                    # clean targets
```

```
    a0  = torch.randn_like(a1)                # noise
    tau = torch.rand(a1.shape[0], 1)          # random time in [0, 1]

    a_tau    = (1 - tau) * a0 + tau * a1       # point on the straight path
    target_v = a1 - a0                         # constant velocity along it

    pred_v = model(a_tau, tau)
    loss   = ((pred_v - target_v) ** 2).mean()

    opt.zero_grad(); loss.backward(); opt.step()
    if step % 1000 == 0:
        print(step, loss.item())
```

Watch the printed loss. It will not go to zero, and that is the tell worth pausing on. A field that perfectly fit every (a_τ, τ) pair to its target would have to output two different velocities at the same point in the seam between basins, which is impossible for a function, so the loss floors out at the irreducible variance the seam contributes. A loss that does drop near zero means your two clusters overlap too much to be distinct modes, so push them farther apart and retrain. The residual loss is not a bug. It is the mathematical signature of the multimodality you asked for.

Exercise 13.x.3 — Draw the field and find the seam

This is the payoff. Evaluate the trained field on a grid over the plane at a fixed mid-flow time, say $\tau = 0.5$, and draw it as a quiver plot.

```
import matplotlib.pyplot as plt

xs = torch.linspace(-4, 4, 20)
ys = torch.linspace(-4, 4, 20)
gx, gy = torch.meshgrid(xs, ys, indexing="xy")
grid = torch.stack([gx.reshape(-1), gy.reshape(-1)], dim=-1)
tau  = torch.full((grid.shape[0], 1), 0.5)

with torch.no_grad():
    v = model(grid, tau)

plt.quiver(grid[:, 0], grid[:, 1], v[:, 0], v[:, 1])
plt.scatter([-2, 2], [0, 0], c="red", s=80, zorder=3)
plt.show()
```

Look at the arrows near the vertical line $x = 0$. To the left of it they lean left, toward the $(-2, 0)$ cluster; to the right they lean right, toward $(+2, 0)$. Along the line itself, right where a naive regressor would have planted every prediction, the arrows go short and undecided, some pointing nowhere in particular, because that is the ambiguous region no single velocity can resolve. The field learned to split the noise rather than average it, and it did so from a plain squared-error loss with no special multimodal machinery anywhere in the code. That splitting picture is §13.3 stated in ink.

Exercise 13.x.4 — Sample, and count the modes

A field is a promise; sampling is the promise kept. Draw a hundred noise points, integrate each one forward with the ten-step Euler solver from §13.3, and see where they land.

```
def sample(model, n=100, n_steps=10):
    a  = torch.randn(n, 2)
    dt = 1.0 / n_steps
    for k in range(n_steps):
        tau = torch.full((n, 1), k * dt)
```

```
        with torch.no_grad():
            a = a + model(a, tau) * dt
    return a

pts = sample(model)
plt.scatter(pts[:, 0], pts[:, 1])
plt.show()
```

Two things to check, and they are the two things that carry over to the real model. First, the samples should land in both clusters, in roughly the fifty-fifty proportion the data had, and almost none should sit at the origin. That is honest multimodal sampling from a regression loss, the claim §13.1 said was hard and §13.3 said flow matching delivers. Second, drop the step count from ten to two and resample. The clusters should still be recognizable, a little sloppier, which is the rectified-flow property (Liu et al., arXiv:2209.03003) that lets π0 run its expert in "ten or so" steps instead of a diffusion sampler's dozens. Now push it to one step and watch it fall apart into a smear near the center, which is the straight-path construction reminding you that few steps is not the same as one. That failure at one step is the honest boundary of the trick, and it is worth provoking on purpose so you know where it lives.

If you want the deciding-between-heads objective the TOC lists, run the token baseline against this: bin each axis into, say, 32 uniform cells, train the same MLP to classify which cell instead of regressing a velocity, and sample. On two clean clusters the token head will do fine, which is exactly the point §13.1 made. The token head fails on resolution and latency, not on this easy multimodality, so a 2-D toy is the wrong instrument to show its weakness. Seeing the token head succeed here is a useful reminder that the flow head's advantage is specific, earned on high-frequency dexterous chunks, and not a blanket superiority you can demonstrate on a scatter plot.

Chapter 13 reading list

Cited across §13.1–§13.6, grouped by the job each reference does. Full entries for everything in the book live in Appendix E.2; this is the chapter-local subset.

The model and its lineage

- Black, K., et al. (2024). "π0: A Vision-Language-Action Flow Model for General Robot Control." arXiv:2410.24164. The spine of the chapter. The architecture of §13.2, the flow-matching objective of §13.3, and the laundry-and-bussing results of §13.4 all come from here; the training step you wrote in this exercise is its loss at 2-D scale.
- Physical Intelligence (2025). "π0.5: A VLA with Open-World Generalization." arXiv:2504.16054. §13.4's evidence that the generalization bottleneck sat in the backbone data rather than the action head, and the co-training move that let the model walk into houses it had never seen.
- Physical Intelligence (2025). "π0.6: recovery and RL-from-experience." Technical report. §13.4's step past the imitation reliability ceiling, and §13.5's example of an RL loop most labs cannot afford to run.
- Physical Intelligence (2026). "π0.7: steerable generalist." Technical report. §13.4's turn from a capable generalist into a controllable one, and §13.5's caution that steerable is not the same as safe.

The generative machinery underneath

- Lipman, Y., et al. (2023). "Flow Matching for Generative Modeling." The objective §13.3 built on: learn a velocity field with a regression loss, no simulation of the flow during training.
- Liu, X., et al. (2022). "Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow." arXiv:2209.03003. The straight-path idea that makes few-step sampling accurate, and the reason Exercise 13.x.4 still works at two steps. It is what buys π0 its control-rate latency.

The token head it replaced

- Pertsch, K., et al. (2025). "FAST: Efficient Action Tokenization for Vision-Language-Action Models." arXiv:2501.09747. §13.1's evidence that clever tokenization compresses the resolution and sequence-length costs but leaves the serial-decode latency untouched, which is the gap π0 was built to close.

- Beyer, L., et al. (2024). "PaliGemma: A Versatile 3B VLM for Transfer." arXiv:2407.07726. §13.2's backbone: the web-pretrained vision-language model π0 keeps intact and pairs with the flow-matching expert.

The thread that reaches Chapter 18

- Assran, M., et al. (2025). "V-JEPA 2 and V-JEPA 2-AC." arXiv:2506.09985. §13.5's candidate answer to where the next 10,000 hours of training data come from when teleoperation runs out; developed in full in Chapter 18.

Chapter summary

Chapter 13 took the action-head fork that Chapter 12 left open and followed the continuous branch all the way down, using π0 as the one model taken apart from every side. You can now say precisely when a discrete action head stops paying its way: a resolution ceiling from uniform binning, a sequence-length blowup on long high-frequency chunks, and a serial-decode latency that is structural to autoregression, with FAST fixing the first two and nothing fixing the third short of leaving tokens behind. You can draw π0's architecture from memory, a web-pretrained backbone for scene and language beside a flow-matching action expert for motion, sharing one attention, with the scene encoded once outside the sampling loop, and you can point to which half holds the transferable knowledge and which holds the embodiment-specific motor skill. You can explain how a plain regression loss yields multimodal actions, and after this exercise you can do more than explain it, because you plotted the splitting field and counted the samples landing in both modes rather than at their average. And you can read the π0⟶π0.5⟶π0.6⟶π0.7 releases as one argument, separating the durable contribution, the flow head that survived all four, from the churn of data and training stacked around it. §14 picks up the ceiling §13.5 named: π0 reasons and acts in a single forward pass, so it cannot think longer on a harder problem, and the dual-system designs of Helix and GR00T N1 are the field's answer to exactly that limit.

Chapter 14 — Dual-system architectures: Helix and GR00T N1

14.1 Why a single forward pass is not always enough

Chapter 13 left us with a clever asymmetry. π0 (arXiv:2410.24164) runs its 3.3B vision-language backbone once per observation, then lets a small flow-matching expert iterate ten cheap times over that frozen scene representation to produce a one-second action chunk. The heavy thinking happens once; the sampling is light. That asymmetry is most of why π0 closes a 50 Hz loop where OpenVLA (arXiv:2406.09246), decoding tokens one at a time through a 7B decoder, cannot. It is a good trick. This chapter is about the tasks where it is not enough, and about the two networks that answer the shortfall by running not once per observation but on two separate clocks at the same time.

Start with what π0's asymmetry does and does not fix. It fixes the cost of generating an action: ten small forward passes beat dozens of serial token decodes. What it does not fix is the coupling between reasoning and reacting. When π0 commits a chunk, it commits roughly a second of motion conditioned on one snapshot of the world. For a tabletop arm sliding a mug, a second is fine; the world barely moves and nothing catastrophic happens if the plan is slightly stale by the end of the chunk. Put the same policy on a humanoid that has to stay upright while it reaches, and one second is an eternity. The robot's balance can go from fine to falling in eighty milliseconds, well inside a committed chunk, and no amount of clever chunk generation helps if the chunk was decided before the disturbance arrived.

Two jobs, two clocks

The deeper issue is that a general-purpose robot is really doing two different jobs, and they want to run at rates that differ by more than an order of magnitude.

The first job is figuring out what to do. Parse "clear the table and put the mugs in the sink," notice that the blue mug is a kind you have never seen, decide to grab it by the rim because it is full, sequence the whole thing so you do not knock over the glass on the way. This is semantic work, and it is exactly what a large vision-language model is good at after the web pretraining we spent Part 4 on. It is also slow, both because a 7B-plus model is expensive to run and because it does not need to be fast. The decision "pick up the blue mug next" stays valid for a second or two. Re-deriving it two hundred times a second would waste enormous compute to arrive at the same answer over and over.

The second job is running the body. Track the commanded trajectory, hold balance, feel the mug start to slip and tighten the grip before it drops, absorb the small shove when a coworker brushes past. This is sensorimotor work, and its defining property is that it has to be fast. A bipedal robot maintaining balance is an unstable system in the control-theory sense; leave it alone for a tenth of a second and it topples. Contact events are worse, because the useful reaction window when a grasp slips is on the order of tens of milliseconds. None of this work needs to understand the instruction. It needs fresh sensor readings and a tight loop.

Section 4.3 already showed you a version of this division, though we did not frame it this way then. Classical robots have always run a fast inner control loop, computed-torque or impedance control at hundreds of hertz, wrapped inside a slow outer planner that reasons about goals at a leisurely pace. The inner loop does not know what the task

is; it just tracks a reference and rejects disturbances. The dual-system VLAs in this chapter are, in a real sense, a rediscovery of that layered structure, with a learned VLM standing in for the planner and a learned sensorimotor policy standing in for the tracking controller. What changed is that both layers are now neural networks trained end to end on demonstrations rather than hand-derived from a dynamics model.

What a single forward pass forces you to give up

Fold both jobs into one network that produces an action from an observation, and you are forced to pick a single rate for the whole thing, because the two jobs share one forward pass. Every choice of that rate is bad in a different way.

Run the whole model fast enough for control, say two hundred hertz, and you cannot afford a 7B backbone in the loop; a model that big does not finish a forward pass in five milliseconds on hardware that fits on a robot. So you shrink the model, and now your reasoning is worse: the shrunken network handles familiar tabletop tasks but loses the open-world semantic grounding that made the big VLM worth using. Run the whole model slow enough for the big backbone, say five to ten hertz, and the control loop inherits that rate. Now the humanoid falls over, because balance cannot wait a hundred milliseconds between corrections. You can try to split the difference and land somewhere unhappy in the middle, too slow to balance and too dumb to reason, which is the worst of it.

$\pi 0$'s chunking is the most sophisticated single-system dodge, and it is worth being honest that it only bends this constraint rather than breaking it. By generating a chunk, $\pi 0$ lets one slow backbone pass cover a second of fast motion, so the effective control rate rises without re-running the backbone. But the chunk is open-loop within itself. The fast corrections inside that second are replayed from what the policy decided at the top of the chunk, not computed against sensor readings arriving mid-chunk. For quasi-static manipulation that gap is tolerable. For anything dynamic it is the whole problem.

The dual-system answer

The fix is to stop sharing the forward pass. Run two networks, at two rates, at the same time.

A slow system, call it System 2, is a full vision-language model. It looks at the cameras and the instruction a few times a second, does the semantic reasoning, and emits not an action but an intent: a compact latent vector that means something like "reach for the blue mug's rim." A fast system, call it System 1, is a small sensorimotor network running at a couple hundred hertz. It takes the latest latent from System 2 as a standing goal and closes the actual control loop against fresh proprioception and vision, turning "reach for the rim" into torques or joint targets, correcting for slip and shove on every cycle. The two share only the thin latent channel. System 2 sets direction; System 1 does the driving.

The naming is a deliberate nod to Kahneman's fast-and-slow split in human cognition, and the analogy is decent as long as you do not lean on it: a deliberate, effortful, general reasoner supervising a quick, automatic, specialized reactor. The engineering payoff is concrete. The expensive model runs at the low rate its job actually tolerates, so you get to keep it big and smart. The cheap model runs at the high rate control demands, so balance and contact stay stable. Neither compromises to accommodate the other, because they are no longer chained to one clock.

Figure's Helix (figure.ai/news/helix) is the cleanest deployed instance. Its System 2 is a 7B open-source VLM that runs on the order of 7 to 9 Hz; its System 1 is an 80M-parameter visuomotor transformer that runs at 200 Hz, driving a humanoid's full upper body including individual fingers. NVIDIA's GR00T N1 (arXiv:2503.14734) takes the same two-rate shape for humanoids, pairing a VLM reasoner with a diffusion-transformer action module, and we will pull both apart in the next two sections. The interesting design questions live in the seam between the systems: how wide the latent channel should be, how the fast system stays stable when the slow system's intent updates underneath it, and whether the two get trained together or separately.

Trace one task through both clocks to see why the split matters. Suppose a Figure humanoid on a logistics line has to pick a plastic tote off a moving conveyor and set it on a rack. System 2 looks at the scene a handful of times a second: it reads the instruction, locates the tote, notices it is sliding left on the belt, and settles on the intent "grasp the near handle, lift, pivot right." That intent holds steady for a second or more; the reasoning does not need refreshing while the hand is still closing. System 1, meanwhile, is doing something System 2 never sees. As the fingers wrap the

handle the tote's weight turns out higher than expected and the grip starts to slip, so the fast loop, sampling force and joint state at 200 Hz, tightens the fingers and adjusts the wrist inside maybe thirty milliseconds. A coworker bumps the robot's shoulder; the same loop rejects the push and holds the trajectory. At no point did System 2 re-plan for the slip or the bump, because it could not have reacted in time and did not need to. The two failures that would sink a single-system policy, a stale plan and a missed contact correction, land on different networks running at the rates that can actually handle them.

None of this comes free. You have doubled the number of networks, invented a communication protocol between them, and taken on the job of keeping a 200 Hz loop stable while its goal shifts every hundred milliseconds or so. A single-system model like OpenVLA has none of that machinery, and for a slow tabletop arm it does not need it. The dual-system design earns its complexity only when the task is fast, dynamic, or dexterous enough that one clock genuinely cannot serve both jobs, which is why §14.7 comes back to the decision of whether your robot needs this at all rather than assuming it does.

We now have the argument in hand: reasoning and reacting want different rates, one forward pass can only offer one rate, and splitting the model into a slow reasoner and a fast controller lets each run at the rate its job requires. The next section makes this concrete by walking through Helix end to end, from the 7B System 2 down to the 200 Hz System 1 that moves the fingers.

14.2 Helix: a high-level VLM and a low-level sensorimotor model

Section 14.1 gave you the argument for splitting a robot policy across two clocks. Helix (figure.ai/news/helix) is the version of that argument you can point at a running humanoid. Figure released it in February 2025 as, by their description, the first VLA to control a full humanoid upper body at high rate from a single set of weights: two wrists, the torso, the head, and all the fingers on both hands, roughly thirty-five degrees of freedom moving together under one policy. What makes it a good teaching example is not that it works but that the seam between its two systems is drawn so plainly. You can name what crosses that seam and what does not.

The two halves, sized

System 2 is a 7B open-source vision-language model, pretrained on internet-scale image and text data before Figure ever touched it. That pretraining is the whole point: it is what lets System 2 look at a mug it has never seen on this robot and still know it is a mug, graspable by the rim, part of the category "dishes" that the instruction "put the dishes away" refers to. System 2 takes in the robot's cameras and the language instruction, runs at roughly 7 to 9 Hz, and produces one thing: a continuous latent vector. Not an action, not a joint angle, not a word. A vector that encodes intent, and that System 1 has learned to read.

System 1 is the small one, an 80M-parameter transformer, and it runs at 200 Hz. Twenty-plus times faster than System 2, on a network almost a hundred times smaller. It takes the latest latent vector from System 2 as a standing goal, combines it with its own view of the cameras and the robot's proprioception (joint positions, velocities, wrist forces), and outputs the actual continuous actions that drive the motors. When System 2's latent says "reach for the near mug's rim," System 1 is the part that turns that into a stream of joint targets, cycle after cycle, adjusting for the mug's real position and the hand's real trajectory as they unfold.

The size gap is not an accident, it is the design. System 2 gets to be big because it runs slowly, and it can run slowly because semantic decisions age slowly; "pick up the mug next" is still true a quarter-second later. System 1 has to be small because it runs fast, and it has to run fast because balance and contact will not wait. Each network is sized to the clock its job runs on. That is the §14.1 principle turned into two concrete parameter counts.

What crosses the seam

The single most important design choice in Helix is what System 2 hands down to System 1, and the answer is: as little as possible, in the form of one latent vector.

Contrast this with the obvious alternative. You could have System 2 emit language, an actual string like "grasp the blue mug's handle and lift slowly," and have System 1 parse it. Figure did not do that, and the reason is instructive.

A discrete language token is a bottleneck; you cannot smoothly interpolate between "grasp" and "grasp a little more to the left," and you pay a tokenizer's latency to encode and decode text on every handoff. A continuous latent has neither problem. It can shift by a small amount to mean a slightly different intent, and System 1 can follow that shift continuously rather than waiting for the next discrete word. The channel between the systems is deliberately thin and deliberately continuous, which is what lets a 7 Hz reasoner steer a 200 Hz controller without either one stuttering.

This thin continuous channel is also what makes the two systems trainable as one thing. Because the latent is a differentiable vector rather than a sampled word, gradients from System 1's behavior-cloning loss can flow back through the latent into System 2 during training. Helix is trained end to end on human teleoperation data, roughly 500 hours of it, with an auto-labeling step that runs a VLM over the recordings after the fact to generate the natural-language instruction each clip should be paired with. No task-specific heads, no per-skill fine-tuning: one network learns to fold "put away the dishes" and "hand me the bottle" and a long tail of other instructions into the same latent vocabulary that System 1 knows how to execute.

Why this beats a single pass, in one task

Take the demo Figure led with: two Figure humanoids standing at a kitchen counter, given groceries they have never seen, told to put them away, cooperating without a script. Watch one robot pick up a bag of cookies and hand it to the other, which places it in a drawer.

System 2 on the first robot reads the scene a handful of times a second. It recognizes the cookie bag as a graspable item, decides the intent "pick up the bag, orient it for a handoff," and holds that intent steady while the hand closes; there is no reason to re-reason mid-grasp, and it could not afford to anyway. System 1, meanwhile, is doing the part System 2 never sees. The bag is deformable, so as the fingers close the contact geometry keeps changing, and the 200 Hz loop keeps adjusting finger positions against the wrist-force readings to hold the bag without crushing it. When the second robot's hand arrives to take the bag, the first robot's System 1 feels the load transfer and releases at the right moment. None of that contact choreography went through System 2. It happened on the fast clock, against fresh sensor data, exactly where §14.1 argued it had to happen.

A single-system policy at 7 Hz would have committed to a grasp and then been blind to the bag deforming under its fingers for 140 milliseconds at a stretch, which for a soft object is the difference between holding it and dropping it or flattening it. A single-system policy at 200 Hz could not have afforded the 7B model that recognized the bag as a bag in the first place. Helix runs both, on the same robot, on two onboard GPUs, one carrying System 2 and one carrying System 1, drawing embedded-scale power rather than a datacenter's.

From two systems to three: Helix-02 and System 0

Helix as shipped in early 2025 controlled the upper body and assumed something else kept the robot standing. That something was conventional: a hand-written whole-body controller, on the order of 100,000 lines of C++, doing balance and locomotion the classical way from §4.3, a fast model-based loop tracking references and rejecting disturbances. Helix reasoned and manipulated; hand-coded control kept the legs under it.

Helix-02 (figure.ai/news/helix-02), which Figure detailed in early 2026, folds that layer into the learned stack as well. Figure calls it "System 0," a neural whole-body controller that replaces the hand-written locomotion code with a learned policy running below System 1. The naming is telling: not System 3, sitting above the reasoner, but System 0, sitting below the fast controller, closest to the metal and running at the highest rate of the three. So the stack now reads top to bottom as a slow VLM setting intent, a fast visuomotor policy turning intent into arm-and-hand actions, and a still-faster learned controller keeping the whole body balanced under whatever those actions do to the robot's center of mass. This is worth flagging as a boundary quietly dissolving. Chapter 4 drew a clean line between classical model-based control and learned policies; Helix-02's System 0 erases part of that line by learning the locomotion that used to be derived from a dynamics model. We come back to what that means for the field in §18.1.

The other Helix-02 result worth naming is coordination. Figure showed two robots working a shared task with no designated leader, each running its own Helix stack, negotiating a joint activity through what they observe of each other rather than through a central planner handing out subtasks. A May 2026 demonstration had two of them tidying a bedroom together on this leaderless basis. For the argument of this chapter the significance is narrow but

real: the dual-system design does not just scale down to one robot's two clocks, it composes across robots, because each robot's slow System 2 is already doing the kind of scene-level reasoning that noticing-what-the-other-robot-is-doing requires.

Where Helix sits relative to what's next

Helix is one instantiation of the dual-system idea, and it made specific choices: a continuous latent channel, end-to-end training, an 80M fast head, and eventually a learned System 0 underneath. NVIDIA's GR00T N1 (arXiv:2503.14734) makes the same top-level split for humanoids but fills in the boxes differently, most visibly by using a diffusion-transformer action module in the fast path rather than Helix's plain transformer, and by publishing its weights and training recipe where Figure has kept theirs closed. Comparing those two choice-by-choice is the fastest way to see which parts of the dual-system design are load-bearing and which are one team's taste, so that is exactly what the next section does, walking through GR00T N1 and setting it beside Helix box for box.

14.3 GR00T N1 and its N1.5–N1.7 successors: humanoid-flavored dual systems

Helix showed you one team's dual-system choices, made behind closed weights. GR00T N1 (arXiv:2503.14734), which NVIDIA released in March 2025, makes the same top-level split and then publishes almost all of it: the architecture, the training recipe, the data mixture, and the weights. That openness is what makes it the better teaching object of the two. Where §14.2 had to describe Helix's fast head as "an 80M-parameter transformer" and stop, GR00T lets you read exactly what goes into the seam between the systems, and it fills that seam differently enough from Helix that the contrast tells you which parts of the design are forced and which are one lab's preference.

The same split, drawn NVIDIA's way

GR00T N1 is, at the top level, the picture from §14.1: a slow vision-language module that reasons, and a fast module that produces the motor commands. NVIDIA labels them System 2 and System 1, borrowing the same Kahneman gloss Figure did.

System 2 is a vision-language model, in the released version a variant of NVIDIA's Eagle VLM, running at roughly 10 Hz. It reads the wrist and head cameras and the language instruction and produces a stream of vision-language tokens, the internal representation a VLM builds before it would normally decode text. System 1 is a diffusion transformer, a DiT of the kind you met in §10.2, running much faster and generating continuous action chunks. The fast path does not decode discrete action tokens the way RT-1 or OpenVLA did; it denoises a chunk of continuous actions across a handful of diffusion steps, conditioned on both System 2's tokens and the robot's proprioception. If you have §10.3 in hand, think of the action module as a flow-matching-style generator sitting where Helix put a plain transformer.

That single substitution is the first real design fork between the two systems. Helix's fast head regresses actions directly; GR00T's samples them from a denoiser. The payoff is the one §10.4 laid out: a diffusion or flow head represents multimodal action distributions cleanly, so when there are two reasonable ways to grasp the tote, the head can keep both alive instead of averaging them into a grasp that reaches for neither. The cost is also the one §10.4 named, extra forward passes per action, which is why the number of denoising steps gets tuned down hard for real-time use.

What crosses GR00T's seam

The channel between the systems is where §14.2 told you to look, so look here too. In Helix the channel is one continuous latent vector, deliberately thin. In GR00T it is wider: System 2 passes down its vision-language tokens, and the DiT cross-attends to them at every denoising step. So System 1 does not get a single compressed intent, it gets the reasoner's full token-level representation to attend over while it builds the action.

Neither choice is obviously right, and the difference is instructive. A thin latent forces System 2 to commit to a compact summary of its intent, which keeps the fast loop cheap and makes the handoff easy to reason about, at the price of throwing away whatever nuance did not fit in the vector. A wide token channel keeps that nuance available

to the fast path, at the price of a heavier cross-attention step on every diffusion iteration and a tighter coupling that makes the two networks harder to develop separately. GR00T bets that the extra context is worth the extra cost; Helix bets the opposite. You cannot settle that from first principles, which is exactly the point of having two open-enough systems to compare.

Both networks train together, end to end, and here GR00T's openness pays off again because you can see what "together" means in data terms. NVIDIA trained on a pyramid: real robot teleoperation at the base narrow layer, a large body of human egocentric video in the middle, and synthetic data generated in simulation on top, including trajectories produced by NVIDIA's own neural "dream" generation. Real teleop teaches the exact embodiment; human video teaches manipulation priors that transfer; simulation fills the long tail cheaply. The mixture is the answer to a problem §14.2 skated past: 500 hours of teleoperation is a lot to collect, and video plus sim is how you get the reasoning module the breadth its VLM pretraining promised without teleoperating every scene by hand. Open X-Embodiment (§12.4) sits inside that real-robot layer.

Why humanoid-flavored

Both Helix and GR00T target humanoids, and it is worth being precise about what that adds beyond "the robot has legs." A humanoid is a high-degree-of-freedom, bimanual, balance-critical platform, which stresses exactly the two failures §14.1 built the whole chapter around. The degrees of freedom are numerous enough that a single slow policy cannot close a stable loop over all of them; the balance constraint means a stale chunk is not merely suboptimal but a fall. GR00T is built to drive that class of body, and NVIDIA pairs it with reference humanoid hardware and its Isaac simulation stack, so the sim layer of the training pyramid matches the robots the policy is meant to run on.

The concrete demonstration to picture is a bimanual manipulation task on a humanoid: two arms coordinating to move an object from one hand to the other, or to hold a container steady with one hand while the other loads it. System 2 reads the scene at 10 Hz and settles the intent, "steady the box, place the item"; the DiT closes the fast loop, denoising action chunks that keep both arms coordinated and the grip stable as the item's weight shifts. The handoff between hands is the same class of contact event §14.2 traced through Helix, and it lands on the fast module for the same reason: the reaction window is shorter than System 2's cycle.

The lineage: N1.5 to N1.7

GR00T did not stop at N1, and the successor line is a clean case study in where a dual-system design gets improved once the skeleton is fixed. The changes cluster in two places, and neither is the top-level split, which stays put.

The first place is the reasoning backbone. N1.5 and the later N1.6 swap and upgrade the VLM: N1.6 moves to a Cosmos-family backbone with a substantially larger action module (reported as roughly a 2× DiT). Cosmos matters because it is trained with physics-informed pretraining, NVIDIA's world-foundation-model line, so System 2 arrives already carrying some sense of how objects move and fall rather than only how they look. That is the §9.4 world-model idea leaking into the reasoner of a VLA, and it is worth flagging as a direction §18.3 returns to.

The second place is the data, and this is where N1.7 makes its central bet. N1.7, which reached general availability in mid-2026, is built on a Cosmos-Reason backbone (a Qwen3-VL-architecture VLM with the physics-informed pretraining above) and, more importantly, on a large first-person human-video corpus NVIDIA calls EgoScale, on the order of 20,000 hours of egocentric video feeding the low-level controller. The through-line from N1 to N1.7 is not a new architecture; it is the same dual-system skeleton fed progressively more and better data, especially human video, which is the cheapest way to buy manipulation priors at scale. Read against §14.2, this is GR00T doing openly what Helix's auto-labeled teleop did quietly: turning cheap, abundant video into the breadth the reasoning half was supposed to provide.

A third family: Gemini Robotics-ER

Helix and GR00T are two instances of one idea, close enough that comparing them box by box is the whole exercise. Google DeepMind's Gemini Robotics line (arXiv:2510.03342) is close enough to belong in this chapter and different enough to be worth naming as a distinct family rather than a third clone.

The split is there: Gemini Robotics-ER, the "Embodied Reasoning" model, plays the System 2 role, and a separate action model plays System 1. What Gemini adds is a heavier emphasis on the reasoning half thinking before it acts. DeepMind calls the pattern "Embodied Thinking," and the shape is the embodied chain-of-thought idea we come back to in §18.4: the reasoner does not just emit an intent, it reasons through spatial and temporal steps, produces intermediate representations like points and trajectories on the image, and then hands a plan to the action model. The other capability worth naming is Motion Transfer, the ability to carry a skill learned on one embodiment over to a different one through the shared reasoning layer, which is one concrete attack on the cross-embodiment problem §18.1 takes up in full.

So the family tree of this chapter has three branches. Helix bets on a thin continuous latent and closed end-to-end training. GR00T bets on a wide token channel, a diffusion action head, and an openly published video-heavy data pyramid. Gemini bets on a reasoning half that thinks explicitly, in embodied chains of thought, before the fast half moves. All three keep the two-clock skeleton from §14.1; they disagree about what the slow clock should produce and how richly it should talk to the fast one.

The comparison also exposes the one question all three punt on, which is how you keep the fast loop stable when the slow loop's output updates underneath it, and how much real-time budget the whole stack actually has once you count the diffusion steps and the cross-attention. Naming the families is the easy part; the next section spends that budget line by line, because a dual-system design that misses its latency target is just two networks that fall over together.

14.4 Latency budgets and real-time control

The last three sections argued that a dual-system design earns its complexity by letting reasoning and reacting run on different clocks. That argument only pays off if the fast clock actually keeps time. A design that splits into two networks and then misses its control deadline has not solved the latency problem; it has hidden it inside a more complicated failure. This section is about the accounting that decides whether the split works: where the milliseconds go, what "real-time" means when the ground truth is a robot that can fall, and the tricks that buy a fast loop its deadline without starving the slow one.

What "real-time" actually demands

Start with the word, because it gets used loosely. Real-time control does not mean fast on average. It means a guarantee: every control cycle produces an action within a fixed deadline, and the guarantee holds on the worst cycle, not the median one. A loop that runs at 200 Hz has 5 milliseconds per cycle, and if one cycle in a thousand takes 40 milliseconds, that one cycle is the whole story for a balancing humanoid. Average throughput is a benchmark number. Worst-case latency is what keeps the robot upright.

Where does the deadline come from? From the physics, not the software. Section 14.1 gave the numbers that matter: a bipedal robot is an unstable system whose balance can go from fine to falling in under a hundred milliseconds, and the reaction window when a grasp slips is on the order of tens of milliseconds. Those are not targets someone chose. They are properties of the mechanism, the way a pendulum's fall time is a property of its length. The control loop has to close faster than the fastest disturbance it needs to reject, and everything else in the latency budget is negotiable against that fixed ceiling.

This is why the field talks about a budget at all. You have a fixed amount of time per cycle, set by physics, and a list of things that have to happen inside it: read the sensors, run the fast network, send the command to the motors. Add them up. If the sum fits under the deadline with margin to spare, the loop is real-time. If it does not, no amount of clever architecture upstream will save you, because the body will have moved on before the action arrives.

Tracing the milliseconds through a dual-system stack

Take Helix's shape from §14.2 and walk a single action from photons to torque. There are two loops running at once, so trace them separately.

The fast loop is System 1, the 80M-parameter transformer at 200 Hz. Its per-cycle budget is 5 milliseconds, and into that window it has to fit: grabbing the latest camera frame and proprioception, one forward pass of the small network, and pushing the resulting joint targets out to the motor controllers. An 80M transformer on an embedded GPU runs a forward pass in a couple of milliseconds if the model is well optimized, which leaves a thin margin for the sensor read and the actuator write. That margin is the entire reason System 1 is 80M parameters and not 800M. The size was chosen backward from the deadline.

The slow loop is System 2, the 7B VLM at 7 to 9 Hz. Its budget is generous by comparison, roughly 110 to 140 milliseconds per cycle. A 7B model reading two camera streams and an instruction, then producing a latent vector, spends most of that budget on the forward pass itself, which is fine, because the semantic decision it produces stays valid across many System-1 cycles. The slow loop can afford to be slow precisely because its output ages slowly.

Now notice the thing that makes the dual-system design work and also makes it subtle: the two loops are not synchronized. System 1 does not wait for System 2. It runs every 5 milliseconds against whatever the most recent latent vector happens to be, and System 2 refreshes that vector whenever its own slower cycle completes. Between refreshes, roughly 20 to 40 fast cycles reuse the same standing intent. The latent sitting in the buffer is always a little stale, by up to one System-2 period, and the design bets that staleness is harmless because semantic intent does not change on a 5-millisecond timescale. That bet is the same one π0's chunking made in §13.2, moved from inside one network to the seam between two.

Chunking and asynchronous inference

The staleness bet is worth dwelling on, because it is the general trick that lets slow models drive fast robots, and it shows up in three forms across the models this book has covered.

π0 (arXiv:2410.24164) uses action chunking. One heavy backbone pass produces a whole one-second chunk of actions, and the robot plays that chunk out at high rate while the backbone is busy computing the next one. The backbone runs slowly; the action stream is fast because it is pre-computed in batches. The cost, from §14.1, is that the chunk is open-loop within itself: the fast corrections inside that second are replayed from a decision made at the top of the chunk, not computed against sensor data arriving mid-chunk.

The dual-system models fix that open-loop gap by putting a real closed loop on the fast clock. System 1 is not replaying a pre-computed chunk; it is genuinely reacting to fresh sensors every 5 milliseconds, using the slow latent only as a standing goal. This is more expensive than chunking, because you pay for a live fast network instead of a cheap buffer readout, and it is the reason the design is reserved for tasks dynamic enough to need it.

The third form is asynchronous execution, and it is the piece that keeps the whole thing from stalling. If System 1 ever blocked waiting for System 2 to finish a forward pass, the fast loop would inherit the slow loop's latency, and the split would buy nothing. So the systems run on separate hardware, in Helix's case two onboard GPUs, and communicate through a shared buffer rather than a blocking call. System 2 writes its newest latent into the buffer when ready; System 1 reads whatever is there, right now, without waiting. GR00T's wide token channel from §14.3 changes what sits in that buffer, a full set of vision-language tokens rather than one vector, and it changes the cost of reading it, since the diffusion head must cross-attend to those tokens on every denoising step. The asynchronous shape is the same. The fast path never blocks on the slow path.

The number that actually bites: jitter

Throughput is the easy number and the one every demo reports. The one that decides whether a robot ships is jitter: the variation in cycle time, the gap between the median cycle and the worst one. A loop that averages 5 milliseconds but occasionally spikes to 30 is not a 200 Hz loop in any sense the control engineer cares about, because control stability depends on cycles arriving on a predictable cadence. A controller tuned for a 5-millisecond period behaves badly when handed an action that is 25 milliseconds late, and "behaves badly" for a balancing humanoid means a stumble.

Jitter comes from the messy parts of the stack, not the network math. Garbage collection pauses in a memory-managed runtime. The operating system scheduling something else onto the GPU for a few milliseconds. A camera frame arriving late because the USB bus hiccuped. Memory allocation that usually hits cache and occasionally does

not. None of these show up in a throughput benchmark run on a quiet laptop, and all of them show up on a robot that has been running for six hours in a warehouse. This is why the hands-on exercise for this chapter asks you to measure worst-case jitter and not just mean latency: the mean is the number that looks good in a paper, and the tail is the number that determines whether the robot survives a shift.

Serious real-time robotics fights jitter with the same tools any hard-real-time system uses. Pin the fast loop to a dedicated core so the scheduler cannot preempt it. Pre-allocate every buffer the loop will ever touch so there is no allocation on the hot path. Keep the fast network small and its execution deterministic, no data-dependent branching that makes some cycles longer than others. A learned System 1 is friendlier here than it might seem, because a fixed-size transformer doing a fixed-size forward pass has almost no data-dependent variation in runtime; it does the same arithmetic every cycle regardless of what the input says. That determinism is an underrated reason the fast half of these systems is a plain feedforward network and not something with a variable number of steps.

Where the diffusion head complicates the budget

GR00T's fast path (§14.3) denoises action chunks instead of regressing them directly, and that choice reaches straight into the latency budget. Every denoising step is a forward pass through the DiT, so a fast head that takes ten steps to produce a chunk costs ten times the compute of a single-pass head for the same output. The multimodality payoff from §10.4 is real, but it is not free, and the place it gets paid for is here, in milliseconds per action.

This is why the number of denoising steps is one of the first things tuned down for deployment. A diffusion model trained with fifty steps might run with four or five at inference, trading some sample quality for the latency headroom the control deadline demands. Flow-matching heads of the $\pi 0$ kind push this further, since a well-trained flow model can produce a usable action in a handful of integration steps rather than dozens. The design tension is clean: more steps buy better, more multimodal actions, and fewer steps buy the deadline. A dual-system humanoid does not get to pick both, and the deadline wins, because an elegant action that arrives after the robot has fallen is worth nothing.

Latency budgeting, then, is where the abstract "two clocks" story from §14.1 turns into engineering you can be wrong about. You can pick the right architecture and still miss the deadline by choosing a fast network that is 20M parameters too big, a denoiser that takes three steps too many, or a runtime that garbage-collects on the hot path. The next section moves from the budget to the robots that have met it in the field, looking at Figure's deployed Helix stack and the GR00T-enabled humanoids, to see what these numbers look like when the robot is doing real work for real hours rather than closing a loop on a bench.

14.5 Deployment case studies

The last four sections built the dual-system idea up from an argument to a latency budget. This one turns it around and asks the only question a factory manager cares about: does it hold up over a full shift, on a real robot, doing real work? A demo runs for ninety seconds under good lighting with an engineer's thumb near the stop button. Deployment means eight hours, a thousand repetitions, a floor that gets cluttered, and nobody watching most of the time. The gap between those two things is where most robot-learning results quietly die, so it is worth looking hard at the few systems that have crossed it.

Figure at BMW: the same task, ten thousand times

Figure's most-cited deployment is not the kitchen demo from §14.2. It is the work its humanoids did on a BMW line at the Spartanburg plant in South Carolina, where the robots were put on a real manufacturing station rather than a staged one. The task was deliberately unglamorous: take sheet-metal parts and place them into fixtures, the kind of load-and-position job that sits at the boring end of automotive assembly and the hard end of robot manipulation, because the parts are heavy, the tolerances are tight, and the fixture will not forgive a placement that is two centimeters off.

What makes this a useful case study is the mismatch between how simple the task looks and how much of the chapter it exercises. Placing a part into a fixture is a contact-rich operation with a short reaction window: the moment the part touches the fixture edge, the robot has tens of milliseconds to feel the contact and adjust before it either seats the part or jams it. That is System 1 territory, the fast loop from §14.4 reacting to wrist-force readings, and it is

exactly the kind of event a 7 Hz single-system policy would sail straight through. Meanwhile the decision of which part, which fixture, in which order lives on the slow clock, where the VLM has time to reason about a scene that changes as the line moves. The Spartanburg task needed both clocks for the same reason the chapter has spent four sections insisting most useful tasks do.

The number that matters here is not success rate on a good run; it is consistency across a bad one. Figure's public framing stressed running the same station repeatedly rather than nailing a single flashy attempt, and that emphasis is the right one. A policy that succeeds 95% of the time looks great in a highlight reel and is a disaster on a line that expects one part every few seconds, because a 5% failure rate is a jam every twenty parts, and a jam stops the line. This is why the deployment bar is so much higher than the benchmark bar from Chapter 15's world. LIBERO does not dock you for the twentieth repetition; a production line does nothing but count it.

From upper body to whole body: what Helix-02 changed on the floor

Section 14.2 described Helix-02's System 0, the learned whole-body controller that replaced roughly 100,000 lines of hand-written C++ locomotion. For deployment that change is more than an architecture note, because the old hand-coded controller was the part of the stack an engineer could reason about line by line and certify by hand. Swapping it for a learned policy means the balance layer is now as opaque as the manipulation layer, and it moves right up against the certification problem §17.5 takes up: you cannot read a neural whole-body controller's source to prove it will not step wrong, so you are back to measuring behavior over long runs rather than proving properties.

The payoff Figure claims for absorbing locomotion into the learned stack is that the whole body now coordinates under one set of learned objectives, so the robot can lean, shift weight, and use its legs as part of a manipulation rather than treating them as a separate balancing problem bolted underneath. Whether that coordination survives a full shift is precisely the thing a deployment tests and a demo cannot. The May 2026 leaderless demonstration, two robots tidying a bedroom with no central planner and no designated leader, is the clearest public evidence that the whole-body stack runs long enough and stably enough to let two of them negotiate a shared task in real time. It is still a demonstration, not a shift on a line, and the honest reading is that Figure has shown the coordination works and has not yet shown it works for eight hours straight.

GR00T on many bodies: the reference-hardware bet

NVIDIA's deployment story is shaped differently, and the difference follows from §14.3. Figure builds one robot and one policy and deploys them together. NVIDIA does not build the robot at all; it ships GR00T as a foundation model other people run on their humanoids, which means "deployment" for GR00T means the same weights, adapted, landing on a range of hardware from different manufacturers. Unitree's G1 is the most common target in public examples, and NVIDIA pairs the model with its Isaac simulation stack so that a new robot can be brought up largely in simulation before it touches the physical machine.

By the time N1.7 reached general availability in mid-2026, that portability was the whole pitch. A lab with a humanoid and a modest teleoperation budget could start from N1.7's weights, fine-tune on a few hundred demonstrations of its own task, and get a working policy without training a foundation model from scratch, which no academic lab can afford. The EgoScale corpus from §14.3, on the order of 20,000 hours of first-person human video baked into the low-level controller, is what makes that fine-tuning cheap: most of the manipulation prior is already in the base model, so the lab's teleop data only has to teach the specifics of its embodiment and task. This is a different bet from Figure's. Figure optimizes one stack end to end for one robot; NVIDIA optimizes for a base model that transfers, and accepts that a transferred policy on someone else's robot will rarely match a purpose-built one.

Both bets are defensible, and they answer to different buyers. If you are Figure and you sell a humanoid as a product, the vertically integrated stack is the thing you can guarantee. If you are a research group or a startup that already owns a G1 and cannot spend a year collecting teleoperation data, a transferable foundation model is the only door that opens at all.

The dexterity entrants: a narrower, harder target

Helix and GR00T aim at whole humanoids doing whole tasks. A second wave of foundation models has picked a narrower and in some ways nastier target: the hands. Genesis AI's GENE line (the 2026 GENE-26.5 release is the current public one) and RLWRLD's RLDX-1 are both foundation models built specifically around dexterous manipulation, the fine finger-level control that even the best humanoid stacks still do worse than a distracted human toddler.

Dexterity is a good case study to close on because it is where the deployment gap is widest and most honest. A humanoid can look impressive walking across a room; a hand folding a shirt or seating a small connector exposes every weakness in the fast loop at once. The contact events are denser, the reaction windows shorter, and the consequences of a millisecond-late correction more visible, so a dexterity-first model is under more pressure on exactly the §14.4 budget than a locomotion-first one. Reported results from these models are improving quickly and are still, by their own creators' framing, below reliable human-level dexterity on open-ended tasks. That honesty is worth taking seriously rather than reading as a failure: it is the field telling you where the frontier actually is, which is not "can a robot walk" but "can a robot's hand recover when the object squirts sideways at the last instant."

A related deployment signal comes from the portability end. LingBot-VLA 2.0, released open-source in mid-2026, is reported to run one policy unmodified across twenty robot morphologies from seventeen manufacturers, trained on 60,000 hours of mostly-teleoperation data. Whatever the eventual verdict on its per-task quality, the fact that a single set of weights can be dropped onto that many different bodies at all is the strongest evidence yet that the cross-embodiment problem from §18.1 is starting to soften, and it is the kind of result that only shows up once a design leaves the lab and gets stress-tested on hardware nobody in the original team owns.

What the case studies actually teach

Step back from the individual robots and a pattern falls out. Every deployment that has held up shares the shape this chapter argued for: a slow reasoner that ages gracefully, a fast controller that meets a hard deadline, and an asynchronous seam between them that never lets the fast loop block on the slow one. None of the field results contradict the latency accounting from §14.4; if anything they sharpen it, because the failures that show up over a full shift are jitter failures and contact-timing failures, the ones the budget predicts, not reasoning failures. The slow half was rarely the thing that broke. The fast half, run for hours against sensors that drift and objects that misbehave, is where deployment lives or dies.

The other lesson is quieter and it is about honesty. The teams whose claims have aged well are the ones that reported repetition counts and shift lengths rather than best-of-N successes, and that admitted where dexterity still falls short of a human hand. The next section pulls these threads together into the four things you should now be able to do with a dual-system design, from sizing the two clocks to reading a deployment claim for the number that actually matters.

14.6 Gemini Robotics 2: unified whole-body loco-manipulation

Section 14.3 placed Gemini Robotics-ER as the third branch of the dual-system family, the one that loads its reasoning half hardest and thinks in explicit embodied chains before the fast half moves. That was the 1.5 and 1.6 line. On 30 July 2026, Google DeepMind pushed the line somewhere none of the three families in this chapter had gone, and it is worth a section of its own rather than a footnote to §14.3, because the thing that changed is not the reasoning half at all. It is the scope of the fast half. Gemini Robotics 2 is, by DeepMind's framing, the first VLA to drive a full humanoid — legs, torso, arms, and multi-finger hands — under a single learned policy, and that "legs" is doing more work in the sentence than it looks.

The handoff every prior stack still had

Go back to how the earlier systems in this chapter actually run a humanoid. Helix, until Helix-02, drove the upper body with its learned policy and left locomotion to a separate controller underneath. GR00T N1 (arXiv:2503.14734) produces manipulation actions and assumes a body that can hold itself up while it does. Even Helix-02's System 0

from §14.5, the learned whole-body controller that swallowed roughly 100,000 lines of hand-written C++, is architecturally a distinct layer sitting below the manipulation policy: the two are both learned now, but they are still two, talking across a seam. The pattern across the whole field has been a manipulation brain riding on top of a locomotion system, with a boundary between them that somebody has to design. Most humanoid demos, bluntly, cheat at that boundary by scripting the locomotion so the manipulation policy never has to deal with it.

That boundary is exactly where a lot of useful tasks fall apart. Reaching for something on a low shelf is not a manipulation problem with a locomotion problem stacked beneath it; it is one motion, in which the robot bends, shifts its weight onto one leg, maybe takes a corrective step, and extends an arm, all coupled through a single center of mass. Split that across two controllers and they negotiate at the seam: the manipulation policy commands a reach that the balance controller then has to accommodate, or refuses to, and the coordination that a human does without thinking becomes an interface-design problem between two networks that were trained to care about different things. The heavier the object and the further the reach, the more the two controllers pull against each other.

Gemini Robotics 2's bet is to erase the seam. One policy outputs commands for the whole kinematic chain, from the ankles to the fingertips, so the coupling between stepping and reaching is internal to the network rather than negotiated between two of them. When the robot leans to place a part, the same policy that decides where the hand goes decides how the legs and torso keep the whole system upright, because to it there is no distinction. This is loco-manipulation treated as one problem, which is the phrase to attach to the section, and it is the direction §18.1 flags as blurring the classical-versus-learned locomotion boundary this book drew back in Chapter 4.

Still a dual system, with a wider fast half

It would be easy to read "one unified policy" as the end of the two-clock idea, and that reading is wrong. The split from §14.1 is still there. DeepMind released Gemini Robotics 2 alongside Gemini Robotics-ER 2, the embodied-reasoning model that plays the slow, deliberate System 2 role, the direct descendant of the ER 1.5 and 1.6 you met in §14.3 with the same "Embodied Thinking" habit of reasoning through spatial and temporal steps before committing. What moved is not the existence of the seam between reasoning and acting; it is the reach of the acting half. Where GR00T's System 1 produces arm actions and Helix's fast head drives the upper body, Gemini Robotics 2's fast half now closes the loop over the entire body, legs included.

So the family tree from §14.3 gains a fourth branch, and the axis it varies is a new one. Helix varied the channel between the systems (a thin continuous latent). GR00T varied the action head (a diffusion transformer cross-attending to a wide token stream). Gemini Robotics-ER varied the reasoning half (explicit embodied chains of thought). Gemini Robotics 2 varies the scope of the fast controller, pushing it down to include the balance and locomotion that every prior stack kept separate. All four keep the slow-reasoner, fast-actor skeleton; they disagree, now on four axes rather than three, about what each half should own.

The demonstration platform is Apptronik's Apollo 2, a commercial humanoid fitted with five-fingered SharpaWave hands, which matters for the same reason Figure's BMW deployment mattered in §14.5: a unified whole-body policy is a claim about a real high-degree-of-freedom, balance-critical body, not a simulated one. The public demonstrations are the kind that expose the coupling, packing items into a box, tying off a trash bag, and unscrewing a light bulb, which DeepMind reports at a 92% success rate. Unscrewing a bulb is a good task to sit with, because it looks trivial and is not: it needs a stable stance, a torso that holds a precise reach, and fingers that apply a steady rotating grip, all at once, which is exactly the whole-body coordination a scripted-locomotion stack cannot fake.

The three variants, and the one that points forward

DeepMind shipped the update as a trio, and the third member is the one most relevant to where this book is going. Gemini Robotics 2 is the flagship whole-body policy. Gemini Robotics-ER 2 is the reasoning model that pairs with it, reachable through Google's AI Studio. Gemini Robotics On-Device 2 is the compact variant built to run without a datacenter in the loop, and its reported headline is cross-embodiment adaptation: it can adapt to a new robot body from only a few hours of data.

That number, if it holds up, is a direct hit on the problem §18.1 takes up in full. Getting a policy trained on one embodiment to work on another has been one of the field's stubborn walls, and the responses this chapter has tracked, GR00T's transferable foundation weights from §14.5, LingBot-VLA 2.0 running one policy across twenty

morphologies, Gemini's own Motion Transfer from §14.3, are all attacks on it. An on-device model that re-targets to a new body in an afternoon of data is the same attack pushed toward the practitioner's end of the problem, the person who owns one robot and cannot spend a year teleoperating it. It is the cross-embodiment story from §14.5 arriving in a form you could actually deploy, and Chapter 16's fine-tuning chapter is the discipline you would use to do the re-targeting.

The open counterpart, and the honest caveats

Gemini Robotics 2 is closed, announced through a blog post and shared at first with a hundred-odd trusted testers rather than as a paper with weights, so treat the specifics the way §14.2 asked you to treat Helix: as a credible account of one lab's design, not an inspectable artifact. The research community's open version of the same direction is worth naming alongside it. WholebodyVLA (OpenDriveLab, ICLR 2026), subtitled "Towards Unified Latent VLA for Whole-body Loco-manipulation Control," attacks the identical problem, one latent policy coordinating locomotion and manipulation, in a form you can read and build on. When you want to understand what unified whole-body control actually requires rather than what a launch post claims, WholebodyVLA is the better teaching object, in the same way GR00T was a better teaching object than Helix precisely because it published.

Two caveats close the section, and they are the ones §14.5 taught you to demand. First, this is an announcement and a set of demonstrations, not a shift on a line. A unified policy that leans and reaches gracefully in a launch video has shown the coordination works; it has not shown it works for eight hours against a floor that gets cluttered and sensors that drift, which is the only bar that counts, and a 92% success rate on a light bulb is a jam every twelve or thirteen bulbs on anything resembling a line. Second, folding locomotion into the learned policy sharpens the certification worry §17.5 will press on. When the balance layer was hand-coded C++, an engineer could read it and argue it would not step wrong. Helix-02's System 0 already gave that up for a learned whole-body controller; Gemini Robotics 2 goes further and puts the balance inside the same opaque policy that does the manipulation, so there is no longer any layer of the stack you can certify by reading its source. You are back to measuring behavior over long runs, and the statistics for measuring a whole-body policy, where a failure can mean the robot fell rather than missed a grasp, are exactly the unstandardized combination of §15.5's success intervals and §14.4's jitter budget that this book keeps flagging as unfinished business.

The unified policy also raises the stakes on the very thing the chapter's hands-on exercise measures. A whole-body controller has more degrees of freedom riding on the fast clock than any upper-body policy did, which means the worst-case jitter that §14.4 said decides whether a stack stands is now guarding a larger, balance-critical action space. §14.7 gathers the chapter's threads into the four things you should now be able to do with a dual-system design, and the hands-on exercise in §14.x then hands you the tools to see that latency pressure yourself: you will profile a single-system and a dual-system loop, measure their latency and their worst-case jitter, and watch why the fast half's timing, not the slow half's intelligence, is what a whole-body robot lives or dies by.

14.7 Summary

Chapter 13 handed this chapter a single working model and one built-in limit. π0 (arXiv:2410.24164) runs its heavy backbone once and lets a cheap flow expert iterate over the frozen scene, which is why it closes a control loop where token-decoding VLAs stall. That asymmetry buys a fast action stream, but it does not separate the two jobs a general robot is actually doing, and §14.1 opened by naming them: a slow semantic job that decides what to do, and a fast sensorimotor job that keeps the body upright and the grasp from slipping. Fold both into one forward pass and you are stuck picking one clock for work that wants two clocks more than an order of magnitude apart. Chapter 14 spent six sections on the design that stops picking. Run two networks at two rates at once, a slow reasoner setting intent and a fast controller closing the loop against fresh sensors, and each half runs at the rate its job can actually tolerate. §14.1 built the argument, §14.2 walked Helix end to end, §14.3 set GR00T N1 beside it box by box and named Gemini Robotics-ER as a third family, §14.4 did the latency arithmetic that decides whether the split works, §14.5 checked the whole idea against robots that have run for real hours on real floors, and §14.6 added Gemini Robotics 2 as a fourth family that pushes the fast half all the way down to the legs. The chapter's job was to make the two-clock structure concrete enough that Part 5's fine-tuning and safety questions read as questions about a specific machine rather than about VLAs in the abstract.

The ideas worth carrying forward

Reasoning and reacting run on clocks that differ by more than an order of magnitude, and one forward pass can only offer one clock. §14.1 is the whole chapter in miniature. The semantic decision "pick up the blue mug next" stays valid for a second or two, so re-deriving it two hundred times a second wastes compute to reach the same answer. Balance and contact will not wait: a bipedal robot can go from stable to falling in under a hundred milliseconds, and the window to react to a slipping grasp is tens of milliseconds. A single network forced to serve both ends up too slow to balance or too small to reason, and π0's chunking bends that constraint without breaking it, because the fast corrections inside a chunk are replayed from a decision made at the top of it rather than computed against sensors arriving mid-chunk. The fix is to stop sharing the forward pass. This is also a rediscovery, not an invention: §4.3's classical robots already ran a fast inner control loop inside a slow outer planner, and the dual-system VLA is that layered structure with both layers now learned.

Helix draws the seam between the two systems as plainly as any deployed system does, and the choice that matters most is what crosses it. §14.2 sized the two halves. System 2 is a 7B vision-language model running at roughly 7 to 9 Hz; System 1 is an 80M transformer running at 200 Hz, almost a hundred times smaller and twenty times faster, driving thirty-five-odd degrees of freedom of a humanoid upper body. What passes between them is deliberately thin, one continuous latent vector rather than a language string, because a continuous channel interpolates smoothly and carries no tokenizer latency, which is what lets a 7 Hz reasoner steer a 200 Hz controller without either stuttering. That thin differentiable channel is also what makes the pair trainable as one thing on roughly 500 hours of auto-labeled teleoperation. Helix-02 then folded balance itself into the learned stack as "System 0," a neural whole-body controller replacing about 100,000 lines of hand-written C++, which quietly erases part of the classical boundary §4.3 drew and hands the question to §18.1.

GR00T N1 makes the same top-level split and fills the boxes differently, and the disagreement is the lesson. §14.3 leaned on GR00T (arXiv:2503.14734) because NVIDIA published the architecture, the recipe, the data, and the weights, so you can read what Figure kept closed. Two forks stand out. GR00T's fast path is a diffusion transformer that denoises action chunks rather than regressing them, so it represents multimodal action distributions cleanly at the cost of extra forward passes per action. And its channel is wide where Helix's is thin: the DiT cross-attends to System 2's full vision-language tokens on every denoising step, keeping nuance the reasoner would otherwise have to compress away, at the price of tighter coupling and a heavier fast loop. Neither bet wins from first principles, which is exactly why having two open-enough systems to compare is worth more than either alone. The N1.5-to-N1.7 lineage then makes a second point: across releases the skeleton stayed fixed and the changes clustered in the backbone (Cosmos, physics-informed) and the data (EgoScale, on the order of 20,000 hours of first-person video), the cheapest way to buy manipulation breadth at scale.

The reasoning-first family kept growing, and by the end of the chapter it had gone whole-body. Google DeepMind's Gemini Robotics-ER (arXiv:2510.03342) keeps the two-clock split but loads the reasoning half harder. Its "Embodied Thinking" pattern has System 2 reason through spatial and temporal steps, emit intermediate points and trajectories on the image, and hand a plan down rather than a bare intent, which is the embodied chain-of-thought idea §18.4 develops in full. Its Motion Transfer carries a skill learned on one embodiment to another through the shared reasoning layer, a direct attack on the cross-embodiment problem §18.1 takes up. §14.6 then showed where that line went next: Gemini Robotics 2 (July 2026) keeps the slow-reasoner, fast-actor skeleton but widens the fast half to drive the entire body, legs included, under one policy, erasing the locomotion- versus-manipulation seam every earlier stack still had. So the chapter's family tree has four branches that agree on the skeleton and disagree about what each clock should own: Helix bets thin latent and closed end-to-end training, GR00T bets wide token channel and open video-heavy data, Gemini Robotics-ER bets on a reasoner that thinks explicitly before the fast half moves, and Gemini Robotics 2 bets on a fast half that owns balance and locomotion too.

A dual-system design that misses its deadline is just two networks that fall over together, and the number that decides this is jitter, not mean latency. §14.4 was the reality check. Real-time control is a worst-case guarantee, not an average: a loop that runs 5 milliseconds most cycles and spikes to 30 on one in a thousand is not a 200 Hz loop in any sense the control engineer cares about, because the controller is tuned for a cadence and a late action means a stumble. The fast network's 80M size was chosen backward from that 5-millisecond budget, and its fixed-size feedforward shape is a feature because it runs the same arithmetic every cycle with almost no data-dependent variation. The split only works because the loops run asynchronously on separate hardware through a shared buffer, so System 1 never

blocks on System 2 and instead reads whatever latent is currently there, accepting staleness of up to one slow-clock period as the price of never stalling. GR00T's diffusion head reaches straight into this budget, which is why the denoising-step count is one of the first things tuned down for deployment, and why flow-matching heads that need only a handful of steps have an edge here. A whole-body policy like Gemini Robotics 2 only tightens this, because now the legs ride the fast clock too.

What you should be able to do now

Four things, in the order the rest of the book uses them.

You should be able to *decide whether a robot needs a dual-system design at all.* Given a task, you can say which of the two failures π0 leaves open actually bites: a stale plan, or a missed fast correction on a contact or balance event. A slow tabletop arm sliding a mug needs neither and should not pay for two networks and a communication protocol; a humanoid picking a tote off a moving belt needs both, because the plan ages fine but the grip slip and the shoulder bump land inside a window one clock cannot serve. This is the diagnosis §14.1 built, and it is what keeps you from reaching for a humanoid-grade stack when a single-system policy would do.

You should be able to *draw either Helix or GR00T from memory and say what crosses the seam.* Slow VLM setting intent, fast controller closing the loop, an asynchronous buffer between them. You can name Helix's 7B/80M split and thin continuous latent, contrast it with GR00T's diffusion head and wide token channel, and say what each choice buys and costs. You can also place Helix-02's System 0 below System 1, explain why absorbing locomotion into the learned stack matters for §17.5's certification problem, and see why Gemini Robotics 2 taking that step further, into one whole-body policy, is the same trend reaching its limit.

You should be able to *do the latency accounting that decides whether the split ships.* You can lay out the fast loop's 5-millisecond budget, explain why worst-case jitter and not mean throughput is the number that keeps a humanoid upright, and point to where the milliseconds go: the forward pass, the sensor read, the actuator write, and the denoising steps if the head is a diffusion model. You can say why the two loops must run asynchronously and why the fast network is kept small and deterministic on purpose.

You should be able to *read a deployment claim for the number that actually matters.* Consistency across a full shift, not success rate on a good run. You know that a 95% policy is a jam every twenty parts on a line that counts every repetition, that a 92% light-bulb demo is the same warning in newer clothes, that the failures which show up over hours are jitter and contact-timing failures rather than reasoning failures, and that the teams whose claims aged well reported repetition counts and shift lengths and admitted where dexterity still falls short of a human hand.

Where the chapter has set up the rest of the book

Chapter 14 hands forward more than it resolves. The largest handoff is to Part 5. §14.5 showed that both Helix's vertically integrated stack and GR00T's transferable foundation model exist because fine-tuning a base model onto your own robot is the only affordable door for most labs, which is precisely the decision Chapter 16 takes apart: pick a base model, build a teleop dataset that does not waste your time, and decide how much of the two halves to freeze. GR00T N1.7's EgoScale video prior is why that fine-tuning can be cheap, since most of the manipulation prior is already baked in and the lab's teleop only has to teach the specifics of its embodiment, and Gemini Robotics On-Device 2's few-hours re-targeting from §14.6 is the same idea pushed toward the practitioner.

The safety thread runs straight into Chapter 17. Helix-02's System 0 made the balance layer as opaque as the manipulation layer, and Gemini Robotics 2 put that balance inside the manipulation policy itself, so you can no longer read the locomotion controller's source to prove it will not step wrong, which is the certification gap §17.5 confronts directly. And the deployment lesson from §14.5, that the fast half run for hours against drifting sensors is where a shift lives or dies, is the argument for the runtime monitors and rollback machinery §17.2 and §17.4 build.

The dataset story stacked under §14.3 sets up Chapter 15. GR00T's training pyramid, real teleop at the base, human egocentric video in the middle, simulation on top, is the concrete answer to a question Chapter 15 takes apart in full: what has and has not succeeded the pooled-teleop corpus of Open X-Embodiment (§12.4). LingBot-VLA 2.0's one-policy-across-twenty-morphologies result from §14.5 is the same thread, and the evaluation complaint that a demo is not a shift is the problem Chapter 15 confronts when it gets to real-robot evaluation and its variance.

One thread runs backward, the way it did in Chapters 12 and 13. The dual-system design is a rediscovery of §4.3's layered control, a learned planner over a learned tracking loop, and Helix-02's System 0 and Gemini Robotics 2's whole-body policy push that rediscovery down to the metal by learning the locomotion that used to be derived from a dynamics model. §18.1 is where the book returns to what that dissolving boundary means for the field.

What the chapter has not covered

Two omissions worth naming. The chapter treated the dual-system split as the answer to the two-clock problem and said almost nothing about how you decide the split's internal boundary, how wide the latent channel should be, how many degrees of freedom belong on the fast clock versus the slow one, or whether the two are best trained jointly or separately. §14.3 showed Helix and GR00T disagreeing on the channel width and left the disagreement open on purpose, because the field has not settled it and this book will not pretend to. That is a design frontier, not a solved recipe.

The chapter also stayed almost entirely on humanoids and the manipulation tasks they run. It did not develop the dexterity-first models §14.5 flagged, GENE and RLDX-1, beyond noting that they aim at the hardest end of the fast loop and remain, by their creators' own framing, below reliable human-level dexterity. That frontier gets its own treatment in §18.2. Nor did the chapter touch how a dual-system policy should behave when its slow reasoner produces an intent the fast controller cannot safely execute, which is a safety question §17.2 owns.

Chapter 14's contribution to the book's argument is to answer the ceiling Chapter 13 named. A single forward pass couples reasoning and reacting to one clock, and splitting the policy into a slow vision-language reasoner and a fast sensorimotor controller, joined by a thin asynchronous channel, lets each run at the rate its job requires, which is what a balance-critical dexterous humanoid needs and a single-system VLA cannot give it. Four families, Helix, GR00T, Gemini Robotics-ER, and Gemini Robotics 2, agree on that skeleton and disagree productively about its details, with Gemini Robotics 2 extending the fast half all the way to unified whole-body control. The design earns its complexity only on tasks fast or dynamic enough to need it, its latency budget is decided by worst-case jitter rather than average throughput, and its deployment record says the fast half run for real hours is where success is actually won or lost. With the two-clock structure understood at this depth, Chapter 15 can ask what data trains these systems and how anyone is supposed to measure whether they work.

§14.x closes the chapter with a hands-on exercise, measuring worst-case jitter rather than mean latency on a simulated two-rate control loop so the number §14.4 insisted on becomes something you plot rather than something you take on faith, followed by the chapter's full reading list.

14.x Hands-on exercise + chapter references

§14.4 asked you to accept a claim on the strength of an argument: that a control loop is judged by its worst cycle, not its average one, and that this is the whole reason a dual-system stack exists. This exercise turns the claim into two numbers you compute yourself. You are going to build a stand-in for a single-system policy and a stand-in for a dual-system one, run each inside a simulated control loop, and measure two things the chapter kept insisting on: end-to-end latency per cycle and the worst-case jitter across a run. If §14.4 is right, the single-system loop will show a latency floor no amount of averaging can hide, and the dual-system loop will keep its fast cycle cheap and steady even while a heavy reasoner grinds away beside it. If §14.4 is wrong, the two histograms will look the same and you will have caught the chapter overselling. It does not, but a plot you made yourself is worth more than a paragraph I wrote.

The TOC states the exercise plainly: profile a single-system VLA and a dual-system VLA, measure end-to-end latency and worst-case jitter. You will not download a 7B checkpoint to do it, because the effect has nothing to do with what the networks compute and everything to do with how the loop is shaped. A small matrix multiply standing in for the fast controller and a large one standing in for the reasoner reproduce the timing structure exactly, and they run on a laptop CPU in seconds. The size ratio is the part that carries meaning: pick it to match the Helix figures from §14.2, an 80M fast network beside a 7B slow one, and the arithmetic you plot is the arithmetic Figure ships.

What you need before you start

NumPy and matplotlib. No robot, no simulator, no GPU, no checkpoint. The only thing you are simulating is time, so the networks can be dummies whose sole job is to burn a predictable number of floating-point operations. That is the point. When the jitter appears, you will know it came from the loop structure and the occasional interruption, not from anything the model was thinking about.

Exercise 14.x.1 — Time a single forward pass at two sizes

Start by building the two networks and measuring what one forward pass costs. Make them plain dense layers whose compute scales with a width parameter, so a small width gives you a System-1-sized model and a large width gives you a System-2-sized one.

```
import numpy as np, time

def make_net(width, depth=3, d_in=64):
    layers = [np.random.randn(d_in, width).astype(np.float32)]
    for _ in range(depth - 1):
        layers.append(np.random.randn(width, width).astype(np.float32))
    return layers

def forward(layers, x):
    for W in layers:
        x = np.maximum(x @ W, 0.0)   # dense layer + ReLU
    return x

fast = make_net(width=256)     # stand-in for the 80M System 1
slow = make_net(width=4096)    # stand-in for the 7B System 2
x = np.random.randn(1, 64).astype(np.float32)

for name, net in [("fast", fast), ("slow", slow)]:
    forward(net, x)                        # warm up caches
    t = []
    for _ in range(200):
        s = time.perf_counter(); forward(net, x); t.append(time.perf_counter() - s)
    t = np.array(t) * 1e3                  # milliseconds
    print(f"{name}: mean {t.mean():.3f} ms, p99 {np.percentile(t, 99):.3f} ms")
```

The exact milliseconds depend on your machine, and that does not matter. What matters is the ratio: the wide network costs something like one to two orders of magnitude more per pass than the narrow one, which is the 7B-versus-80M gap of §14.2 showing up as wall-clock time. Note the second number too. The p99 sits above the mean even here, on a dummy net doing nothing but matrix multiplies, because the operating system, memory allocator, and cache all conspire to make some passes slower than others. That gap between mean and p99 is jitter in its simplest form, and it only gets worse from here.

Exercise 14.x.2 — Run the single-system loop

Now put the slow network inside a control loop the way a single-system VLA forces you to. Every cycle has to produce an action, and in a one-network design the only network you have is the big one, so every cycle pays the full reasoner cost.

```
def run_single_system(n_cycles=1000):
    latencies = []
    for _ in range(n_cycles):
        s = time.perf_counter()
        forward(slow, x)          # the ONLY thing that can produce an action
```

```
        latencies.append((time.perf_counter() - s) * 1e3)
    return np.array(latencies)

lat = run_single_system()
print(f"single-system: mean {lat.mean():.2f} ms, "
      f"max {lat.max():.2f} ms, jitter {lat.max() - lat.mean():.2f} ms")
```

Read the numbers against a control deadline. §14.4 put the fast loop's budget near 5 milliseconds, a 200 Hz cadence. If the mean here is already above that, the single-system loop cannot hold 200 Hz at all, which is the structural problem, not a tuning one. And the max will sit well above the mean, so even when the average squeaks under budget, the worst cycle blows through it. On a robot that worst cycle is a late action, and a late action on a balance controller is a stumble. The loop is not "mostly fast enough." It has no fast cycle to fall back on.

Exercise 14.x.3 — Run the dual-system loop and compare jitter

Here is the design the chapter is about. Run the fast network every cycle, and run the slow network only once every `k` cycles, caching its output as the current intent. The fast network reads that cached latent and closes the loop against it.

```
def run_dual_system(n_cycles=1000, k=25):
    latencies = []
    intent = forward(slow, x)               # initial slow pass
    for i in range(n_cycles):
        s = time.perf_counter()
        if i % k == 0:
            intent = forward(slow, x)        # refresh intent occasionally
        forward(fast, x)                     # fast controller, every cycle
        latencies.append((time.perf_counter() - s) * 1e3)
    return np.array(latencies)

dual = run_dual_system()
print(f"dual-system: mean {dual.mean():.2f} ms, "
      f"max {dual.max():.2f} ms, jitter {dual.max() - dual.mean():.2f} ms")
```

Two features should jump out. The mean drops toward the fast network's cost, because 24 of every 25 cycles pay only the cheap forward pass. But the max stays high, spiking on exactly the cycles where `i % k == 0` and the slow pass runs inline. That is the trap §14.4 warned about: naively bolting a slow reasoner onto a fast loop moves the mean but not the worst case, and worst case is what the controller lives or dies by. A histogram makes it obvious.

```
import matplotlib.pyplot as plt
plt.hist(lat,  bins=40, alpha=0.5, label="single-system")
plt.hist(dual, bins=40, alpha=0.5, label="dual-system (inline slow pass)")
plt.axvline(5.0, color="red", linestyle="--", label="5 ms budget")
plt.xlabel("cycle latency (ms)"); plt.legend(); plt.show()
```

You will see the dual-system distribution pile up near the fast cost with a thin tail of tall spikes reaching into single-system territory. The spikes are the unshielded slow passes. Every one of them is a missed deadline waiting to happen.

Exercise 14.x.4 — Decouple the clocks and watch the tail collapse

The fix, and the actual dual-system design, is to stop running the slow pass inside the fast loop. §14.4 described two loops on separate hardware talking through a shared buffer, so the fast controller never blocks on the reasoner and just reads whatever intent is currently sitting in the buffer. You can simulate that decoupling in a single thread by charging the slow pass to a separate clock and never letting it land inside a fast cycle's timing.

```
def run_decoupled(n_cycles=1000, k=25):
    fast_latencies, intent = [], forward(slow, x)
    slow_budget = 0.0                       # "time owed" to the slow clock
    for i in range(n_cycles):
        s = time.perf_counter()
        forward(fast, x)                    # fast loop: ONLY the cheap net
        fast_latencies.append((time.perf_counter() - s) * 1e3)
        # slow pass runs on its own clock, off the fast loop's critical path
        if i % k == 0:
            ss = time.perf_counter(); intent = forward(slow, x)
            slow_budget += (time.perf_counter() - ss) * 1e3
    return np.array(fast_latencies), slow_budget

fast_lat, _ = run_decoupled()
print(f"decoupled fast loop: mean {fast_lat.mean():.2f} ms, "
      f"max {fast_lat.max():.2f} ms, jitter {fast_lat.max() - fast_lat.mean():.2f} ms")
```

The number to stare at is the max, and it should now sit close to the mean, because the only work timed inside the fast loop is the fast network. The tall spikes are gone. The reasoner still runs, still costs what it costs, but its cost no longer shows up on the clock the controller is graded against. That is the entire trick of the dual-system architecture reduced to one loop and a comment: the fast half's worst-case jitter is bounded by the fast network alone, and the slow half is free to be as heavy and as slow as reasoning demands. Plot all three histograms together and the story is complete, with the single-system loop far to the right of the budget line, the inline dual-system loop pulled left but keeping a spiky tail across it, and the decoupled loop packed tight against the fast cost with almost no tail at all.

If you want the piece §14.4 flagged about diffusion heads, add a variable number of denoising steps to the fast network in Exercise 14.x.3, looping the fast forward pass three or five times per cycle. Watch the fast loop's own mean climb and its budget headroom shrink. That is why GR00T's denoising-step count (arXiv:2503.14734) is one of the first knobs tuned down for deployment, and why a flow-matching head that needs only a handful of steps has room to spare where a many-step sampler does not.

Chapter 14 reading list

Cited across §14.1–§14.7, grouped by the job each reference does. Full entries for everything in the book live in Appendix E.2; this is the chapter-local subset.

The two flagship dual-system models

- Figure AI (2025). "Helix: A Vision-Language-Action Model for Generalist Humanoid Control." figure.ai/news/helix. The spine of §14.2. The 7B/80M split, the 7–9 Hz and 200 Hz clocks, the thin continuous latent between the two systems, and the roughly 500 hours of auto-labeled teleoperation all come from here.
- Figure AI (2026). "Helix-02." figure.ai/news/helix-02. §14.2 and §14.5's source for "System 0," the neural whole-body controller that replaced about 100,000 lines of hand-written C++ locomotion, and the leaderless two-robot demonstration. It is also the setup for the certification worry §17.5 inherits.
- Bjorck, J., et al. (2025). "GR00T N1: An Open Foundation Model for Generalist Humanoid Robots." arXiv:2503.14734. §14.3's open counterexample to Helix: the diffusion action head, the wide token channel that cross-attends to full vision-language tokens on every denoising step, and the N1.5–N1.7 lineage where the skeleton held fixed while the backbone (Cosmos) and the data (EgoScale, on the order of 20,000 hours of first-person video) did the changing. Its denoising-step count is the knob Exercise 14.x.4 asks you to feel.

The single-system baseline and its ceiling

- Black, K., et al. (2024). "π0: A Vision-Language-Action Flow Model for General Robot Control." arXiv:2410.24164. §14.1's starting point. π0's one-heavy-pass, cheap-expert asymmetry is the design that bends the two-clock constraint without breaking it, and the limit it leaves open (one forward pass, one clock) is the ceiling this whole chapter answers.

The third and fourth families

- Google DeepMind (2025). "Gemini Robotics-ER 1.5 and the Embodied Reasoning family." arXiv:2510.03342. §14.3's third branch: the two-clock split with a heavier reasoning half, the "Embodied Thinking" pattern that emits intermediate points and trajectories before the fast half moves, and Motion Transfer across embodiments. The embodied-chain-of-thought idea here is developed in full in §18.4, and Motion Transfer feeds the cross-embodiment discussion in §18.1.
- Google DeepMind (2026). "Gemini Robotics 2, Gemini Robotics-ER 2, and Gemini Robotics On-Device 2." deepmind.google/blog (announced 2026-07-30). §14.6's fourth branch: the first VLA to drive a full humanoid, legs included, under one learned policy, demonstrated on Apptronik's Apollo 2 and erasing the locomotion-versus-manipulation seam every earlier stack kept. The On-Device 2 variant's few-hours re-targeting to a new body feeds §18.1, and putting balance inside the manipulation policy sharpens the certification worry of §17.5.
- OpenDriveLab (2026). "WholebodyVLA: Towards Unified Latent VLA for Whole-body Loco-manipulation Control." ICLR 2026. §14.6's open research counterpart to Gemini Robotics 2, the readable version of unified whole-body control.

The backward thread

- Featherstone, R. (2008). *Rigid Body Dynamics Algorithms.* Springer. §14.1 and §14.7's reminder that the fast-inner-loop, slow-outer-loop structure predates learning by decades. The dual-system VLA is §4.3's layered control with both layers now learned, and Helix-02's System 0 is where that rediscovery reaches the locomotion layer classical methods used to own.

Chapter summary

Chapter 14 took the ceiling Chapter 13 named and built the field's answer to it. You can now explain why a single forward pass cannot serve a general robot: the semantic decision of what to do stays valid for a second or two, while balance and contact demand corrections inside tens of milliseconds, so one network is forced to be either too slow to stand or too small to reason. You can draw both Helix and GR00T N1 from memory, name what crosses the seam in each (Helix's thin continuous latent against GR00T's wide token channel, Helix's closed end-to-end training against GR00T's open video-heavy recipe), and place Gemini Robotics-ER as a third family that loads the reasoning half harder. You can do the latency accounting that decides whether a split ships, and after this exercise you can do more than recite it, because you plotted the single-system loop stranded past its budget, watched an inline slow pass leave a spiky tail across the deadline, and saw the tail collapse once the two clocks were decoupled. And you can read a deployment claim for the number that matters, consistency across a full shift rather than success on a lucky run, knowing the failures that show up over hours are jitter and contact-timing failures, not reasoning ones. Chapter 15 picks up the question this chapter kept deferring: what data trains these systems, and how does anyone measure whether they actually work.

Chapter 15 — Datasets, benchmarks, and evaluation

15.1 What a robot dataset looks like, by example

Part 4 kept talking about datasets as if you already knew what one contained. OpenVLA trained on 970,000 trajectories; Octo on 800,000; the whole scale argument of Chapter 12 rested on piles of robot data whose internal shape we never opened up. This chapter opens it. Before you can critique a published evaluation or build your own, you have to know what a single robot episode actually is at the level of bytes and arrays, and the fastest way to learn that is to take one apart.

So take one apart. We will use an episode from BridgeData V2 (Walke et al., 2023, CoRL), a public tabletop-manipulation dataset collected on a low-cost WidowX arm, because it is small enough to inspect by hand and it is one of the larger contributors to Open X-Embodiment. The specifics differ across datasets; the anatomy does not. Once you have seen how Bridge lays out an episode, RT-1's data (arXiv:2212.06817) and the other fifty-nine datasets in the pool read as variations on the same template.

An episode is a list of timesteps

The unit of a robot dataset is not an image or an action. It is an *episode*: one continuous attempt at one task, from the moment the robot starts moving to the moment it stops. "Put the spoon in the pot" is an episode. "Fold the cloth" is another. An episode that ran for six seconds at a 5 Hz control rate is a list of thirty timesteps, and each timestep is a dictionary that pairs what the robot saw with what it did next.

Concretely, one Bridge timestep holds roughly this:

```
timestep = {
    "observation": {
        "image_0":      uint8[256, 256, 3],   # wrist-mounted RGB camera
        "image_1":      uint8[256, 256, 3],   # over-the-shoulder RGB camera
        "state":        float32[7],           # end-effector pose + gripper
    },
    "action":           float32[7],           # what the teleoperator commanded
    "language_instruction": "put the spoon on the towel",
    "is_terminal":      False,
    "is_first":         True,                 # only on timestep 0
    "is_last":          False,
}
```

Read that dictionary slowly, because the entire rest of the book has been feeding on its contents without showing them to you. The `observation` is the policy's input, the thing a VLA's vision encoder and proprioceptive channel actually consume. The `action` is the training target, the label a behavior-cloning loss regresses toward, exactly as §6.2 described. The `language_instruction` is what turns a plain imitation dataset into a *language-conditioned* one,

the ingredient that made everything from RT-1 onward possible. The boolean flags are bookkeeping so the data loader knows where episodes begin and end when it streams thousands of them back to back.

Stack thirty of these dictionaries and you have an episode. Stack sixty thousand episodes and you have BridgeData V2. That is all a robot dataset is: a very large list of lists of these dictionaries.

The observation: more than an image

The word "vision" in vision-language-action hides how much of the observation is not pixels. Bridge gives you two camera streams, and the two-camera setup is not incidental. A wrist camera sees the gripper and whatever it is about to touch, which is where the fine motor action happens; a fixed external camera sees the whole scene, which is where the object you were told to grab actually sits. Drop either one and a class of tasks becomes unsolvable from the images alone. This is why §16.2, when it gets to collecting your own data, spends real time on camera placement rather than treating it as a detail.

The `state` vector is the part beginners skip and then regret skipping. Those seven floats are the robot's proprioception: the (x, y, z) position of the end effector, its orientation, and the gripper's open-or-closed value. Proprioception matters because pixels alone are ambiguous about the robot's own configuration. Two frames can look nearly identical while the arm is in genuinely different poses, and a policy that only sees images has to infer its own joint state from the visual scene, which it does badly. Feeding the state in directly removes the guesswork.

One warning that costs people days. The images are stored as `uint8`, values 0 to 255, because that is how cameras produce them and how you save disk. Almost every model expects `float32` normalized into some range the vision backbone was pretrained on. The conversion happens in the data loader, not in the file, and forgetting it is one of the most common reasons a freshly loaded dataset trains to garbage.

The action: whose convention, in what units

The `action` field is where robot datasets stop resembling image datasets and start being their own hard problem. Seven floats again, but these seven mean something completely different from the seven in `state`. In Bridge they are a *delta*: how much to move the end effector along each axis, how much to rotate it, and whether to change the gripper, all relative to the current pose. The policy is not predicting where the arm should be. It is predicting how it should change over the next timestep.

Three facts about that action vector decide whether two datasets can even be trained together, and none of them is visible until you check:

- **Frame of reference.** Is the action expressed in the robot's base frame or the end-effector frame? Bridge uses end-effector-relative deltas; other datasets use absolute base-frame poses. Mix them without conversion and the policy learns contradictions.
- **Units and scale.** Meters or centimeters, radians or a normalized (-1, 1) range. RT-1 discretizes each action dimension into 256 bins before training; Bridge keeps them continuous. The tokenization choice from §11.3 is a choice made *about this vector*.
- **Control rate.** Bridge runs near 5 Hz, so each action covers a fifth of a second of motion. A dataset recorded at 50 Hz packs ten times as many, much smaller, actions into the same physical motion. Same task, same robot, wildly different-looking action sequences.

This is the concrete face of the MDP-to-robot translation problem from §5.5, and it is why §15.2 can spend an entire section on how Open X-Embodiment harmonized sixty of these action conventions into one loader. Right now the point is narrower: an action is not a number, it is a number plus a convention, and the convention lives in the dataset's documentation rather than in the array.

The language instruction, and how it got there

RT-1 and Bridge both attach a natural-language string to every episode, and that string is doing quiet, heavy work. It is the bridge between the web-scale language pretraining a VLA inherits and the specific motion in front of it.

Without it you have plain behavior cloning, a policy that does one thing; with it you have a policy you can *tell* what to do.

Where the string comes from is less magical than it sounds and worth knowing before you trust it. In most tabletop datasets a human wrote it, either the teleoperator narrating what they were about to do or an annotator labeling the clip afterward. That means the labels are as noisy as human labels always are. "Put the spoon on the towel," "place spoon on cloth," and "move the spoon" might all describe the identical motion in three different episodes, and a few instructions will simply be wrong because someone mislabeled a clip at the end of a long session. Later datasets started auto-generating or templating instructions to cut that noise, which trades human sloppiness for template rigidity. Neither is clean. When a language-conditioned policy behaves oddly on a phrasing that should work, the instruction distribution in its training data is the first place to look.

How it is stored: RLDS and why the format is not a footnote

Sixty datasets from twenty-one labs arrived in sixty formats. The thing that let anyone train across all of them at once was an agreement on storage, and for Open X-Embodiment that agreement is RLDS, the Reinforcement Learning Datasets format, which sits on top of TensorFlow Datasets. RLDS is just a convention for serializing exactly the episode-of-timesteps structure we walked through, plus the metadata a loader needs to shuffle and batch episodes efficiently without reading them all into memory.

Efficiency here is not academic. A single Bridge image is a quarter-megabyte before you even have the second camera or the next twenty-nine timesteps, so a serious pretraining mixture runs into terabytes and cannot live in RAM. The data loader streams episodes from disk, decodes the JPEG-compressed frames on the fly, normalizes the `uint8` images to `float32`, samples a window of timesteps, and hands the model a batch, all while the GPU is busy on the previous batch. Get that pipeline wrong and your expensive accelerator sits idle waiting for data, which is the single most common way a robot-learning run wastes money. §15.2 opens the RLDS plumbing further; Appendix D shows the loader setup end to end.

Simulation datasets look almost the same

Everything above described real robot data, teleoperated by humans on physical arms. A large share of the benchmarks you will meet in §15.4 are simulated instead, and the reassuring news is that a simulated episode has the identical shape: observations, actions, a language instruction, terminal flags. LIBERO (Liu et al., 2023), the benchmark Chapter 2 already ran OpenVLA against, serves episodes that slot into the same data loader as Bridge with barely a change.

The differences are two, and they matter for opposite reasons. Simulated data is cheap and clean: no teleoperator fatigue, no camera calibration drift, perfectly consistent labels, and you can generate ten thousand episodes overnight instead of paying humans for a month. Simulated data is also fake in ways that leak: the physics of contact and friction are approximated, the rendered images carry a visual signature no real camera produces, and a policy that only ever saw simulation frames tends to stumble on the sim-to-real gap we first hit in §7.5. The dataset format is portable across the sim-real boundary. The competence a policy learns from it is not.

What you can do now

You can read the phrase "we pretrained on 800,000 trajectories" and know what was actually on disk: a list of episodes, each a list of timesteps, each timestep a dictionary pairing multi-camera images and a proprioceptive state vector with a commanded action and a language string, serialized in a streaming format like RLDS so a loader can feed a GPU without melting. You can spot the three ways two datasets silently disagree, in action frame, in units, and in control rate. And you know that a simulation episode wears the same clothes as a real one while hiding a different body underneath. The next section takes the one dataset the whole field leans on, Open X-Embodiment, and shows how sixty of these piles were forced into a single trainable corpus.

15.2 Open X-Embodiment in detail — and what has (and has not) succeeded it

Section 15.1 took one episode apart. This section takes sixty datasets and asks the harder question: what does it take to train a single policy across all of them at once, when each one was collected by a different lab, on a different robot, under a different set of conventions? Open X-Embodiment (OXE, arXiv:2310.08864) is the answer the field converged on, and it is worth understanding in detail, because almost every open VLA you met in Part 4 was pretrained on some slice of it. OpenVLA's 970,000 trajectories came from here. Octo's mixture came from here. When a paper says "we pretrained on the OXE mixture," it is leaning on the plumbing this section describes.

What is actually in the pool

OXE is not a dataset in the way BridgeData V2 is a dataset. It is an aggregation: roughly sixty pre-existing datasets, contributed by twenty-one institutions, pooled into one place and re-serialized into a common format. The 2023 release covered twenty-two distinct robot embodiments and more than a million real robot trajectories, spanning something like five hundred skills across the whole collection. The embodiments range from the WidowX arm that recorded Bridge, through Google's mobile manipulators, to Franka Pandas, xArms, and a handful of stranger platforms. No single lab could have collected this. The whole point of pooling was that nobody had to.

The reason anyone bothered pooling is the RT-X result, which §12.4 stated and this section can now explain. Train a policy on one robot's data and it learns that robot. Train the same architecture on the pooled data from twenty-two robots and, on several of the contributing platforms, it does *better* on that platform's own tasks than a policy trained only on that platform's data. The RT-1-X and RT-2-X models in the OXE paper showed roughly a 50% improvement in success rate on the evaluated embodiments over models trained in isolation. That is positive transfer across robots, and before OXE it was not obvious it would happen at all. A skill learned on a Franka should not, on the face of it, help a WidowX; the arms have different kinematics, different cameras, different grippers. It helps anyway, because the visual and semantic structure of "pick up the object the instruction names" is shared even when the hardware is not.

The harmonization problem, concretely

Here is where §15.1's warning about silent disagreements comes due. Sixty datasets meant sixty ways of writing down an action, an observation, and an instruction, and you cannot feed a model contradictions and expect transfer. Someone had to force them into agreement. Three axes of disagreement did most of the damage.

Action spaces first. Some datasets store absolute end-effector poses in the robot's base frame; others, like Bridge, store relative deltas in the end-effector frame. OXE's loader converts these toward a common convention so that the number 0.02 means the same physical thing across datasets. Get this wrong and the policy sees "move right" and "you are here" labeled identically, which teaches it nothing but noise. The gripper dimension is its own small nightmare: some datasets encode open as 1 and closed as 0, others invert it, others use a continuous width in meters. Every one of those has to be mapped to a single scale before training.

Observations second. Bridge gives two cameras; RT-1's data gives one; some datasets carry depth, most do not; a few include multiple wrist views. A model with a fixed input structure cannot consume all of these as-is. OXE's practical answer was to standardize on a primary third-person camera present across most of the pool and treat everything else as optional, which is why models trained on it lean so heavily on a single external view. That decision quietly shaped what the resulting policies can and cannot see.

Control rate and horizon third. A dataset logged at 3 Hz and one logged at 50 Hz describe the same reach with wildly different numbers of timesteps, and the actions at 50 Hz are each a sixteenth the size. The mixture does not fully resolve this; it mostly leaves rates as they are and lets the model average over them, which is one reason OXE-trained policies are smoother on some embodiments than others. Frequency mismatch is the harmonization problem OXE handled least cleanly, and it is worth knowing that going in.

All of this rides on RLDS, the storage format §15.1 introduced. RLDS on top of TensorFlow Datasets is what let sixty piles become one streamable corpus, with per-dataset sampling weights so the loader can decide how often to draw from Bridge versus from a tiny two-hundred-episode contribution. Those weights are not neutral. A naive uniform sample would drown the small datasets, so the OXE mixtures ship with hand-tuned weights that down-weight the

giants and up-weight the rare embodiments. When you read that a model used "the OXE magic soup mixture," those weights are the recipe, and they are as much a modeling choice as the architecture.

A worked slice

Suppose you want to reproduce a smaller version of OpenVLA's pretraining on a laptop-scale budget. You do not pull all sixty datasets; you pull a mixture of maybe eight, chosen for coverage: Bridge and the Google robot data for volume, a Franka dataset or two for a different arm, RT-1's data for the language-conditioned tabletop core. You load them through the RLDS loader with the published sampling weights, and the loader hands you batches where consecutive examples might come from a WidowX in Berkeley and a mobile manipulator in a Google kitchen, already converted into the same action convention and cropped to the same primary camera. Your model never learns that these came from different labs. It sees one distribution. That erasure of provenance, done correctly, is the entire trick.

What has succeeded OXE, and what has not

Here is the honest state of things as of mid-2026: there is no single branded successor to Open X-Embodiment. People expected an "OXE 2.0" with ten million trajectories to appear and anchor the next generation the way OXE anchored 2023 through 2024. It did not. What happened instead is that the center of gravity moved from *aggregating past academic datasets* to *collecting fresh data at fleet scale*, and that data tends to live with the group that collected it rather than in one shared pool.

The growth now comes from large teleoperation fleets. AGIBot's Genie-1 program collected on the order of a million trajectories from a fleet of humanoid and bimanual robots operated in purpose-built data-collection facilities, not scraped from twenty-one labs but produced on an assembly line of human teleoperators. Bimanual rigs like the low-cost YAM arm and platforms like the Unitree G1 humanoid feed similar pipelines. The shape of a single episode is still exactly what §15.1 showed; the difference is industrial scale and central control of quality, which is precisely what a pooled academic corpus could never guarantee.

One concrete example of where this leads. LingBot-VLA 2.0, released in July 2026 by Ant Group's robotics effort, is a 6-billion-parameter open-source VLA trained on roughly sixty thousand hours of data: about fifty thousand hours of real robot teleoperation plus ten thousand hours distilled from first-person human video. A single policy from that training runs unmodified across twenty robot morphologies from seventeen manufacturers, and its authors report beating $\pi 0.5$ on a bimanual benchmark. Notice what that is: not a dataset anyone else can download and remix the way they remixed OXE, but a model whose training corpus is the moat. The data did not become a shared resource; it became a competitive asset.

So OXE succeeded in the way influential infrastructure usually succeeds. Its dataset is now one contributor among many rather than the frontier, but its *conventions* won. RLDS serialization, cross-embodiment sampling weights, the action-harmonization discipline, the assumption that a generalist policy trains on many robots at once: all of that is now just how the field works, baked into loaders that newer fleets and models take for granted. The recent datasets-and-benchmarks survey (arXiv:2604.23001) catalogs dozens of these post-OXE corpora and data engines, and the through-line is that they all speak OXE's dialect even when they do not use OXE's data.

For your own purposes this matters in a specific way. If you are pretraining from scratch, OXE plus a few newer public releases (standardized teleop sets like DROID, for instance) is still the most practical open starting mixture, because it is downloadable and the loader is battle-tested. If you only need to fine-tune, §16.2 will make the case that a few hundred episodes of your own robot, collected cleanly, beats any amount of somebody else's data, and OXE's role shrinks to whatever the base checkpoint already absorbed from it.

What you can do now

You can explain why pooling sixty robot datasets improves performance on each one rather than diluting it, and cite RT-X's roughly 50% transfer gain as the evidence. You can name the three axes on which datasets silently disagree, and say what OXE did about each: convert action frames and gripper encodings, standardize on a primary camera, and mostly punt on control-rate mismatch. You know that RLDS plus tuned sampling weights are the

actual mechanism, not a detail. And you can describe the post-OXE landscape accurately: no branded successor, but a shift to fleet-scale teleoperation whose data stays private while OXE's conventions became universal. The next section turns from the RLDS lineage to the other one, LeRobot, whose format, hub, and low-cost hardware collapse the distance between owning an arm and training a policy on it; §15.4 then leaves real datasets for the simulation benchmarks, from LIBERO to SimplerEnv, that let you compare two policies without owning a robot.

15.3 LeRobot: Hugging Face's dataset format, hub, and community-contributed robot data

Section 15.2 left you with a picture of the field's storage habits circa 2023: sixty academic datasets, RLDS serialization on top of TensorFlow Datasets, sampling weights tuned by hand. That lineage produced Open X-Embodiment and the models trained on it. It also came with a tax. RLDS is a TensorFlow-first format, the tooling assumes you are comfortable in that ecosystem, and contributing a new dataset meant learning a serialization convention that most robotics labs touched exactly once. A second lineage grew up alongside it, aimed at a different user: the person with one low-cost arm on a desk who wants to record a hundred episodes this afternoon and train tonight. That lineage is LeRobot.

LeRobot is Hugging Face's open-source robotics library, released in 2024 and written for PyTorch rather than TensorFlow (Cadene et al., 2024). It bundles four things that the RLDS world kept separate: a dataset format, a hosting hub, a set of pretrained policies you can load in two lines, and driver support for cheap hardware you can build from a parts list. The bet behind it is that robot learning was gated less by ideas than by friction, and that collapsing the distance between "I have an arm" and "I have a trained policy" would matter more than any single architecture. Whether that bet paid off is something you can now check on the Hub yourself: the number of community-uploaded robot datasets there went from a handful at launch to several thousand within about a year.

LeRobotDataset: the same episode, a different on-disk shape

Recall the timestep dictionary from §15.1: images, a proprioceptive state vector, an action, a language instruction, some boolean flags. LeRobotDataset stores exactly that structure, but the encoding decisions differ from RLDS in ways that reveal what the format is optimized for.

The tabular data (states, actions, timestamps, episode indices, task indices) lives in Parquet files, the columnar format from the Apache Arrow world that Hugging Face's `datasets` library already speaks. The camera streams do not sit in those Parquet files as raw arrays. They are encoded as MP4 video, one file per camera per episode (or chunked across episodes in later format versions), and the Parquet rows carry pointers plus timestamps into those videos rather than the pixels themselves. This is the single most consequential design choice in the format, so it is worth dwelling on.

Why video? Go back to §15.1's arithmetic. A single 256×256 RGB frame is a quarter-megabyte uncompressed; a two-camera, 30-timestep episode is fifteen megabytes of raw pixels, and a serious dataset is tens of thousands of episodes. Storing frames as JPEG helps, but consecutive frames in a robot episode are almost identical (the arm moves a centimeter, the background does not move at all) and JPEG throws away none of that inter-frame redundancy. Video codecs exist precisely to exploit it. Encoding an episode's camera stream as H.264 or AV1 routinely shrinks it by an order of magnitude over per-frame JPEG, because the codec stores one keyframe and then only the deltas. The cost is that reading a random timestep now means seeking into a compressed video and decoding, which is more expensive per-frame than reading a JPEG; LeRobot's loader hides this behind the same streaming-and-prefetch machinery §15.1 described, decoding on background workers while the GPU chews the previous batch. For datasets dominated by camera data, which is nearly all of them, the trade lands well on the storage side.

A LeRobotDataset on disk, then, is a directory with three kinds of content: the Parquet files holding the numeric time series, the MP4 files holding the video, and a small pile of JSON metadata. The metadata is where the format earns its keep for anyone loading a dataset they did not create. An `info.json` declares the schema: which features exist, their shapes and dtypes, the frames-per-second, the codec used for the video. Separate files hold the mapping from task-index to natural-language instruction and the per-feature statistics (mean, std, min, max) that a policy needs for input normalization. That last item deserves emphasis: the normalization statistics travel with the dataset. One of

the recurring ways a fine-tuning run in §16 fails is a mismatch between the statistics a checkpoint was trained under and the statistics of the new data, and LeRobot's decision to ship stats as first-class metadata is a direct attempt to make that failure loud instead of silent.

Querying time: the delta-timestamp trick

Here is a problem every robot data loader has to solve and RLDS solves less ergonomically. A policy almost never wants a single timestep. Behavior-cloning models predict a chunk of future actions at once (ACT and Diffusion Policy from Chapter 10 both do), and many architectures condition on a short stack of past frames. So the loader has to hand you windows, not points: "give me the current observation, the previous two frames, and the next sixteen actions."

LeRobotDataset expresses this with a `delta_timestamps` argument. You ask for a feature at a list of time offsets relative to the current frame, in seconds, and the loader assembles the window for you:

```
from lerobot.common.datasets.lerobot_dataset import LeRobotDataset

delta_timestamps = {
    "observation.image": [-0.2, -0.1, 0.0],   # two past frames + current
    "action": [t / 30 for t in range(16)],     # next 16 actions at 30 fps
}
dataset = LeRobotDataset("lerobot/aloha_static_coffee", delta_timestamps=delta_timestamps)
sample = dataset[0]
# sample["observation.image"] -> tensor [3, C, H, W]
# sample["action"]            -> tensor [16, action_dim]
```

The offsets are in seconds, not indices, which is the detail that makes the format robust across recording rates. A dataset logged at 30 fps and one logged at 50 fps both answer "the action 0.1 seconds from now" correctly, because the loader resolves the request against the real timestamps stored in the Parquet rows rather than counting rows. This is the frequency-mismatch headache from §15.2 handled at the query layer instead of being left for the model to average over. It does not make two datasets recorded at different rates identical, but it does mean your code asks for physical time and gets it, regardless of how densely a given dataset sampled that time.

The Hub, and what "community-contributed" actually buys you

RLDS datasets tend to live wherever the lab that made them chose to park them. A LeRobotDataset lives on the Hugging Face Hub by default, addressed by a `repo_id` like `lerobot/aloha_static_coffee`, versioned like a Git repository, and loadable by anyone with the string. `LeRobotDataset("some-user/their_dataset")` pulls it the same way you pull a language-model checkpoint. Uploading your own is a single `push_to_hub()` call after recording.

The effect of putting robot data behind the same one-line pull as a pretrained model is quantitative, not just cosmetic. It dropped the cost of sharing a dataset close to zero, and the Hub filled with community uploads: teleoperation runs on SO-100 and SO-101 arms, ALOHA bimanual data, Koch-arm datasets, task-specific collections someone recorded for a workshop and left public. Several thousand robot datasets now sit there, most of them small, many of them noisy, a few of them excellent. This is a different animal from Open X-Embodiment. OXE is a curated, harmonized corpus assembled by a collaboration; the LeRobot Hub is a bazaar. The upside is coverage and immediacy: if you own an SO-101, someone has probably already published data on a task near yours. The downside is exactly what you would expect from a bazaar: no guarantee of quality, no harmonization across uploads, and camera setups, control rates, and labeling conventions that vary wildly from one `repo_id` to the next. §15.1's warning about silently disagreeing datasets applies with full force here, minus the curation layer OXE provided.

Both storage lineages now coexist, and the field is bilingual rather than converged. Conversion tools move OXE's RLDS datasets into LeRobotDataset form so PyTorch-first practitioners can use them, and the two formats trade influence rather than one replacing the other.

Policies and hardware, in the same box

Two features push LeRobot past being merely a dataset format. First, it ships pretrained and trainable policies as importable modules: ACT, Diffusion Policy, TDMPC, VQ-BeT, and, the reason this matters for a book about foundation models, π0 and SmolVLA. You can load a π0 checkpoint and run inference on a LeRobotDataset without leaving the library, which is why Chapter 16's fine-tuning walkthrough leans on it.

SmolVLA (arXiv:2506.01844) is the clearest expression of the project's thesis. It is a compact VLA, roughly 450 million parameters, trained substantially on community-contributed LeRobot Hub datasets rather than on a proprietary fleet corpus, and designed to run on consumer hardware including a single GPU or even a CPU. It exists to prove that the bazaar produces something usable: that data recorded by hobbyists and small labs on sub-$1,000 arms, pooled through the Hub, can train a VLA that works. Set it beside the fleet-scale models of §15.2, the million-trajectory teleoperation programs behind LingBot-VLA and AGIBot, and you have the two poles of the field's data strategy. One pole pours capital into private industrial-scale collection; the other tries to make the open long tail add up to something. SmolVLA is the evidence that the second pole is not just idealism.

Second, LeRobot supports the low-cost hardware that makes the whole loop reachable. The SO-100 and SO-101 arms, the Koch and Moss designs, and ALOHA-style bimanual setups all have drivers in the library, with parts lists and assembly instructions that put a working teleoperation rig in the low hundreds of dollars. This closes the circuit that Chapter 16 will walk in full: build the arm, teleoperate it to record a LeRobotDataset, push it to the Hub, fine-tune SmolVLA or π0 on it, deploy the policy back to the same arm. The format, the hub, the policy, and the robot are one toolchain, and that integration is the actual product.

What you can do now

You can read `LeRobotDataset("some-user/task")` and know what lands on disk: Parquet files holding the numeric time series, MP4 files holding each camera as compressed video, and JSON metadata carrying the schema, the task strings, and the normalization statistics that keep a fine-tuning run honest. You can explain why the format stores video instead of frames and what that trade costs at read time. You can use `delta_timestamps` to pull past-and-future windows in physical seconds, sidestepping the control-rate mismatch that RLDS left to the model. And you can place the LeRobot Hub correctly against Open X-Embodiment: not a curated corpus but an open bazaar of several thousand community datasets, whose payoff — proof that pooled hobbyist data can train a working VLA — is SmolVLA. The next section leaves datasets for evaluation and walks through the simulation benchmarks, LIBERO, CALVIN, RoboCasa, and SimplerEnv, that let you compare two policies without owning any of this hardware.

15.4 Sim benchmarks (LIBERO, CALVIN, RoboCasa, SimplerEnv) and red-team-style suites such as RoboArena

You cannot borrow a fleet of real robots to test your policy, and even if you could, you could not afford to run a thousand trials on each one. Simulation is how the field gets around that. A sim benchmark is a fixed set of tasks, a fixed simulator, and a fixed scoring rule, packaged so that two labs on two continents can report numbers that mean the same thing. That last clause is the whole point and also the whole problem, because a benchmark is only as honest as the gap between its simulated world and the messy one your robot actually lives in. This section walks the four benchmarks the VLA literature quotes most, explains what each one measures and what it quietly ignores, and then turns to a newer style of evaluation that tries to close the sim-to-real credibility gap by giving up on sim entirely.

LIBERO: the fine-tuning yardstick

If you have read an OpenVLA or π0 paper, you have seen LIBERO numbers. It has become the default place to report how well a VLA adapts to a small suite of manipulation tasks, and you already met it in §2.2 when you stood up its simulator to watch OpenVLA run. LIBERO (Liu et al., 2023) was built to study lifelong learning, the question of whether a robot can pick up new skills without forgetting old ones, but the community adopted it for a simpler purpose: it is a clean, fast, tabletop manipulation testbed with a Franka Panda arm and a language instruction attached to every task.

The design worth remembering is how LIBERO factors difficulty. It splits its 130 tasks into four suites, each isolating one kind of generalization. LIBERO-Spatial keeps the objects fixed but moves their arrangement, so the policy has to key on spatial language ("the bowl on the left"). LIBERO-Object holds the layout and swaps the objects. LIBERO-Goal fixes objects and layout and changes only the instruction, testing whether the policy actually reads the goal instead of running one memorized motion. LIBERO-Long, sometimes called LIBERO-100, chains multiple subtasks into long-horizon episodes that punish compounding error of the kind you met in §6.3. Reporting a single LIBERO average hides which of these a policy is good at, so read the per-suite breakdown; a model can post a strong mean while quietly failing every LIBERO-Long episode.

LIBERO is cheap to run, which is most of why it won. A modern VLA fine-tune plus evaluation on all four suites fits on a single GPU in an afternoon, and that low cost is exactly what makes it a fine-tuning benchmark rather than a pretraining one. Its weakness follows from the same fact: the scenes are simple, the assets are a small fixed set, and a policy that tops the LIBERO leaderboard has told you it can adapt to a narrow tabletop distribution, nothing more.

CALVIN: long horizons and language

CALVIN (Mees et al., 2022) attacks the part LIBERO treats as one suite among four. It is built around long-horizon, language-conditioned manipulation, and its signature metric is chaining: the robot is given a sequence of five instructions in a row ("open the drawer," then "pick up the blue block," then "push it left," and so on), and it only gets credit for task k+1 if it completed tasks 1 through k. The headline number everyone quotes is "average successful sequence length," a value between 0 and 5. A policy that nails single tasks but drifts over a minute of continuous control gets exposed here in a way LIBERO's shorter episodes forgive.

The second thing CALVIN did was formalize environment generalization through its train/test splits, labeled A, B, C, and D, which are four tabletop environments differing in textures and object positions. The hard setting, written ABC→D, trains on three environments and evaluates on a fourth the policy never saw. That single arrow is doing real work: it separates a model that learned the task from one that overfit the training rooms. When a paper reports CALVIN ABC→D, it is claiming its policy transfers across scenes, and that is a stronger claim than a same-environment number.

RoboCasa: scale and generated assets

LIBERO and CALVIN share a limitation. Their asset libraries are small and hand-authored, so a policy can, in principle, memorize every mug it will ever see. RoboCasa (Nasiriany et al., 2024) was built to break that. It drops the robot into fully modeled kitchens, built on the RoboSuite/MuJoCo stack, with thousands of 3D objects (many generated by text-to-3D models) spread across roughly 100 tasks that range from atomic skills like turning a faucet to composite activities like making coffee.

RoboCasa matters here for two reasons. First, its scale makes memorization a losing strategy, which pushes evaluation closer to the generalization we actually care about. Second, it is designed as a data engine as much as a benchmark: it ships large volumes of machine-generated demonstrations, so you can study how policy performance scales with synthetic data volume, a question that connects directly to the scaling arguments of §11.5. The catch is cost. A kitchen full of articulated objects is heavier to simulate than a LIBERO tabletop, so RoboCasa evaluations run slower and demand more compute, and that friction is why you see it quoted less often than LIBERO despite being the more demanding test.

SimplerEnv: making sim numbers predict real numbers

The three benchmarks above all share one unexamined assumption: that success in their simulator tells you something about success on hardware. Usually it does not, at least not quantitatively. A policy trained on real Google robot data and evaluated in a generic sim will post numbers that correlate weakly, if at all, with its real-world success rate, because the rendering, the physics, and the camera placement are all subtly wrong. SimplerEnv (Li et al., 2024) is the benchmark built to fix precisely that mismatch, and it is the one the chapter's hands-on exercise uses.

The idea is narrow and clever. Rather than build a new simulated world and hope policies transfer to it, SimplerEnv reconstructs the specific real-world setups that already have published real-robot evaluations, the Google robot

(RT-1, RT-2) tabletop and the WidowX/BridgeData setup, matched carefully enough in visuals and dynamics that a policy's success rate in the sim tracks its success rate on the physical rig. The measure of merit for SimplerEnv is not "is this task hard" but "does my sim score predict my real score," and the paper reports the correlation directly. When that correlation is high, you have earned the right to iterate in simulation and trust that improvements will survive contact with the real robot. That is a different and more useful promise than a raw leaderboard, and it is why SimplerEnv became the standard way to compare published VLAs like OpenVLA (arXiv:2406.09246) and Octo (arXiv:2405.12213) without every lab needing to rebuild the original hardware.

Two cautions come with it. SimplerEnv only covers the handful of real setups it was tuned against, so it evaluates your policy on those embodiments and tasks, not yours. And the correlation it reports is a property of the policies tested when it was built; a new architecture that games the visual match without solving the task can, in principle, break it. Treat a strong SimplerEnv number as good evidence, not proof.

RoboArena: when you stop trusting sim

Every benchmark so far buys reproducibility with a fixed world, and a fixed world is a world you can overfit. If a benchmark's tasks, objects, and initial states are frozen and public, a well-funded team can tune against them until the leaderboard number stops meaning anything, the same treadmill that eventually discredited saturated benchmarks in vision and language. The response taking shape in robotics is to move evaluation back onto real robots but distribute it across many labs, and RoboArena (Atreya, Pertsch et al., 2025) is the clearest example.

RoboArena borrows its structure from the way large language models get ranked in Chatbot Arena. Instead of a fixed task suite with a success threshold, it runs pairwise comparisons: two policies attempt the same free-form, evaluator-chosen task on real hardware at whichever institution is hosting, and a human judges which did better. Aggregate those pairwise verdicts across many tasks, many labs, and many robot instances, and you get a relative ranking that no single team controls or can quietly train against. The tasks are not published in advance because there is no fixed list; each evaluator improvises, which is where the "red-team" flavor comes from. You are not measured on how well you memorized a benchmark. You are measured on whether you hold up when a stranger picks the task.

The price is everything reproducibility gave you. Results depend on which labs participated, which robots they own, and how the human judges felt that day, so RoboArena trades the clean, rerunnable number of a sim benchmark for a noisier signal that is much harder to game. Both kinds of evaluation are load-bearing, and a serious policy report in 2026 tends to carry both: a LIBERO or SimplerEnv table for cheap, repeatable iteration, and a distributed real-robot comparison for the claim that the thing actually works off the bench. §15.5 takes the real-robot side of that pairing seriously and works through how to measure success rate, variance, and time-to-completion when every trial costs you a physical robot and a human minder.

15.5 Real-robot evaluation: variance, success rate, time-to-completion

A sim benchmark hands you a number and a promise that anyone can reproduce it. A real robot hands you a number and a shrug. Run the same policy on the same task fifty times and you will not get the same answer, because the lighting drifted, the gripper picked up a little grease, the object landed two centimeters left of where it did yesterday, and the tape on the table has started to peel. This is not sloppiness you can engineer away. Physical evaluation is a noisy measurement of a quantity you cannot observe directly, and the whole job of this section is to treat it that way: how to estimate a success rate you can defend, how to say honestly how uncertain that estimate is, and what to record beyond the pass/fail bit so that "it works" means something.

Success rate is an estimate, not a score

Start with the number everyone reports. You run the policy N times, it succeeds k times, and you write down k/N as the success rate. That fraction is not the policy's success rate. It is a sample estimate of an unknown probability p, and the gap between the two is where most published robot numbers quietly mislead.

Each trial is close enough to a coin flip with bias p that the binomial model applies: k successes out of N draws. The thing you actually care about, p, is hidden, and k/N is your best guess at it. The trouble is that guesses from small

N are wide. Suppose you run 20 trials and get 15 successes. You report 75%. A 95% confidence interval around that number, using the Wilson method that behaves well for proportions near 0 and 1, runs roughly from 53% to 89%. So "75%" is compatible with the true policy being a coin flip on a good day or nearly reliable, and your 20 trials cannot tell those apart. That is not a rounding concern. A difference between 53% and 89% is the difference between a demo and a product.

The lesson is blunt: report the interval, not just the point. A success rate without an N and an uncertainty band is a marketing figure. When you read a paper that says "our policy achieves 90% on the pick-and-place task" with no trial count, assume the worst about N until proven otherwise, because the authors who ran 200 trials almost always tell you so.

How many trials do you need? Enough that the interval is narrow enough to support the claim you want to make. The width of a binomial interval shrinks with the square root of N, which is unforgiving arithmetic. Going from 20 to 80 trials halves your error bar; going from 80 to 320 halves it again. If you want to distinguish a 70% policy from an 80% policy with any confidence, you are looking at well over a hundred trials per policy, and that is why honest real-robot comparisons are expensive and why so many papers dodge them with sim. There is no trick that gets you a tight interval from ten rollouts.

Where the variance actually comes from

Treating trials as independent coin flips is a useful fiction, and like all useful fictions it leaks. The reasons it leaks are worth naming, because each one is a source of variance you can either control or at least record.

The first is initial-condition variance. Where the object starts, at what angle, under what light, on what surface: change any of these and you have changed the task. A policy can hit 95% when the mug always starts in the same 5-centimeter box and collapse to 40% when you scatter the mug across the whole table. Neither number is wrong; they measure different distributions of starting states. So the initial-state distribution is part of the benchmark, and if you do not fix it and describe it, your success rate is uninterpretable. Good protocols mark starting positions with a grid or randomize them from a written, reproducible scheme, and they report which.

The second is temporal drift, the slow kind. Over a long evaluation session the hardware changes underneath you. Motors warm up and their friction shifts, the gripper's rubber wears, cameras auto-adjust exposure as the afternoon sun moves, and a cable you bumped at trial 30 subtly changes the arm's zero. None of this is in your policy, all of it is in your numbers, and the fix is to interleave rather than batch: if you are comparing policy A against policy B, alternate them trial by trial instead of running all of A in the morning and all of B after lunch. Batching lets the time-of-day confound ride directly on top of the comparison you care about.

The third is the human in the loop. Someone resets the scene between trials, and that someone makes judgment calls: was that a success or did the block wobble off at the last second, is this starting pose close enough to the intended one, should we retry because the object rolled off before the policy even started. Those calls move the number. RoboArena (Atreya, Pertsch et al., 2025) leans into this by making a human's pairwise preference the unit of measurement, which sidesteps the "what counts as success" argument by asking "which of these two was better" instead. For a single-policy success rate you do not get that luxury, so you write the success criterion down before you run anything: contact does not count, the object must be inside the target region for two seconds, a dropped object is a failure even if it lands in the right place. Decide it in advance, because deciding it while watching a marginal trial is how bias walks in.

Not every failure is the same failure

The pass/fail bit throws away information you paid for. A policy that fails by knocking the mug over is telling you something different from one that fails by freezing, and a policy that fails by grabbing the wrong object is different again. When you log failures, log the mode: no-grasp, wrong-object, dropped-in-transit, missed-placement, collision, timeout. A confusion table of failure modes turns a flat 75% into a diagnosis. The OpenVLA evaluations (arXiv:2406.09246) and the $\pi 0$ report (arXiv:2410.24164) both break results down by task category rather than dumping one aggregate, and that breakdown is what lets a reader see whether a policy's weakness is perception, grasping, or the long-horizon glue between steps.

This connects straight back to §6.3. Compounding error means failures cluster late in an episode, so a policy that succeeds on the first two subtasks of a five-step chain and dies on the third is not "60% good"; it has a specific failure surface at step three. Aggregate success rate hides that. Per-step or per-subtask success rates expose it.

Time-to-completion, and why it is not a tiebreaker

Two policies both hit 80% on the same task. One finishes the average successful trial in 14 seconds, the other takes 41. They are not equally good, and success rate alone cannot see the difference. Time-to-completion is the second axis of real-robot evaluation, and it matters for reasons beyond throughput: a policy that takes three times as long is usually taking a more roundabout, hesitant path, which means more chances to knock something over and less headroom when the next task is waiting.

Report time-to-completion only over successful trials, and report its spread, not just its mean. A policy that finishes in 14 seconds give or take 2 is a different animal from one that averages 14 but ranges from 6 to 55; the second one is fast on the easy starts and grinding through the hard ones, and that variance predicts how it will behave when the task gets harder. Because completion times are usually skewed with a long right tail, the median and the interquartile range describe them more honestly than mean and standard deviation. And watch the failure-truncation trap: if your hard trials tend to fail and get excluded, your average completion time is computed only over the easy successes and looks better than the policy deserves. Always read time-to-completion next to success rate, never instead of it.

There is a third quantity worth logging that sits between the two: intervention rate, if your setup allows a human to nudge or reset a stuck policy mid-episode. How often a supervisor has to step in, and for what, is often the number a deployment team cares about more than raw success, because it maps directly onto how many robots one person can babysit. §16 and §17 return to this when we talk about fielding a policy for real work rather than a paper.

A reporting protocol you can actually defend

Pulling it together, here is what a real-robot evaluation should carry so that someone else can read it and know what they are looking at.

Fix and describe the initial-state distribution before running, whether gridded or randomized from a stated scheme. Choose N large enough that your confidence interval supports your claim, and report N alongside the rate. Interleave policies trial by trial when comparing, never batch. Write the success criterion down in advance and apply it mechanically. Log the failure mode of every failed trial, not just the count. Report success rate with a Wilson (or equivalent) interval, and report time-to-completion over successes as a median with its spread.

That is more bookkeeping than a sim benchmark demands, and it is the price of a number that survives contact with someone else's robot. A distributed effort like RoboArena exists precisely because no single lab's protocol is fully trusted by the rest; spreading evaluation across institutions and using relative comparisons is a way of averaging out one lab's peeling tape and warm motors. You will not run RoboArena at home. But you can run its underlying discipline, which is to treat every reported success rate as an estimate with an error bar and a paper trail, and that discipline is what §15.6 turns into a checklist for building an evaluation of your own from scratch.

15.6 Building your own evaluation

By now you can read someone else's evaluation and tell whether to trust it. The harder job is designing one from scratch for a policy nobody has benchmarked yet, which is exactly what you face the first time you fine-tune a model for a robot that no public suite covers. There is no leaderboard for "my WidowX arm sorting my specific bin of parts." You have to build the ruler before you can measure anything, and a badly built ruler will lie to you for weeks before you notice. This section is the procedure: what to decide, in what order, and where the traps sit.

The framing that saves you the most pain is to treat an evaluation as a design artifact with the same care you would give a piece of the policy. It has requirements, it has failure modes, and it can be wrong. A good one answers a question you actually have. A bad one produces a number that feels informative and is not.

Start from the decision, not the task

Before you pick tasks or count trials, write down the sentence that this evaluation is supposed to let you finish. "I will ship policy B instead of policy A if ." "I will keep collecting data if ." "I will trust this on the real robot if ___." The blank is your target: a threshold, a comparison, a go/no-go gate. Everything downstream, how many trials, which tasks, what you log, is set by that sentence, and skipping it is the most expensive mistake in the whole chapter because it produces evaluations that measure something real but useless.

An example of the trap. You want to know whether adding wrist-camera data helped, so you run both policies on your standard ten tasks and policy B scores three points higher overall. Did wrist data help? You cannot tell, because the tasks where a wrist camera should matter, the tight-clearance grasps, are four of your ten, and averaging them with six tasks the wrist never sees washes the signal out. The evaluation measured aggregate competence when your question was about a specific capability. Had you started from the decision, "keep the wrist camera if it improves tight grasps," you would have built a different task set.

So: one evaluation, one question. If you have three questions, you may need three task sets, and pretending one number answers all of them is how teams talk themselves into shipping the wrong model.

Choose tasks that span the difficulty you care about

Two failure shapes bracket a bad task set. Too easy and every policy scores near 100%, so the evaluation cannot separate them; this is the ceiling effect, and it is common when you design tasks around the demo you already know works. Too hard and everything scores near zero, which is a floor effect and just as blind. The tasks that carry information are the ones where good policies pull apart from bad ones, which means you want a spread: a few your current policy mostly handles, a few it half-handles, a couple it fails. The half-handled ones do most of the discriminating.

Borrow structure rather than inventing it. LIBERO (arXiv:2306.03310) organizes its tasks along axes that are worth stealing even if you never run LIBERO itself: variation in objects, in spatial layout, in goal, and in long-horizon composition. Those axes give you a checklist for coverage. If every task in your set varies the object but none vary the spatial layout, you will not learn whether your policy generalizes across placements, and placement is where §15.5's initial-condition variance bites hardest. Pick tasks so that each axis you care about moves in at least a couple of them.

One more coverage rule that people skip: include a task the policy should refuse or cannot do, if safety matters to you at all. A policy that confidently attempts an impossible grasp and drags the arm into the table is worse than one that stalls, and an evaluation made only of solvable tasks never sees that difference. Chapter 17 builds on this when it treats safety as its own evaluation axis rather than a footnote to success rate.

Fix the protocol before the first trial

Everything §15.5 argued about noise becomes a set of decisions you write down now, before a single rollout, because deciding any of them mid-evaluation lets bias in through the side door.

The success criterion, first and most important. Write the pass condition for each task as something a stranger could apply without watching you: "the block is fully inside the printed square for two continuous seconds, gripper open." Not "it looks placed." If you cannot state it mechanically, you will adjudicate marginal trials by mood, and your mood correlates with which policy you are rooting for.

The initial-state distribution, second. Decide whether object start poses are gridded or randomized, and if randomized, from what scheme, and write it in a form you can reproduce next month. Mark the positions with tape or a printed template. This is the single biggest lever on the absolute success number, so a rate reported without it is uninterpretable, exactly as in §15.5.

Then the trial count. Work backwards from your decision sentence. If you need to tell a 70% policy from an 85% one, a back-of-envelope power calculation using the binomial spread from §15.5 puts you north of a hundred trials per policy, and no amount of wishing shrinks that. If you only need to know whether a policy clears 50% at all, far fewer

will do. Decide N from the gap you must resolve, not from how many rollouts you have patience for, and if patience and statistics disagree, shrink the claim rather than the sample.

Interleaving and blinding round it out. Alternate policies trial by trial so warm motors and drifting light hit both equally. If a human scores the trials, have them score without knowing which policy produced the rollout when you can manage it, because knowing leaks into the borderline calls. RoboArena (Atreya, Pertsch et al., 2025) takes both of these to their logical end with distributed, preference-based scoring across labs; you are running a one-room version of the same idea.

Log the episode, not the outcome

Here is the move that pays off for months: record every trial in the same shape as a training episode from §15.1, not as a row in a pass/fail spreadsheet. Save the observations, the actions the policy emitted, the language instruction, timestamps, the outcome, and the failure mode when it failed. Two reasons this matters beyond tidiness.

First, a failure you logged fully is a failure you can replay. When policy B misses a grasp on trial 47, a bare "fail" tells you nothing, but the recorded observation stream lets you watch what the policy saw and often see immediately that the object was half out of frame, or the instruction was one you never trained on. The OpenVLA evaluations (arXiv:2406.09246) break results down by task and failure category precisely because the aggregate hides the diagnosis; you get that breakdown for free if you logged the modes as you went.

Second, evaluation rollouts are expensive to collect and identical in format to training data. A hard task your policy fails on is a candidate for your next data-collection session. Throwing that rollout away because your eval harness only kept a boolean is discarding the most valuable data you produce, the data from exactly the states where the policy is weak. Log episodes and your evaluation doubles as a failure-mining pipeline that feeds §16.

A minimal record per trial looks like this:

```
trial = {
    "policy_id":     "B_wristcam_v3",
    "task_id":       "tight_grasp_bin",
    "seed":          47,             # or gridded init index
    "instruction":   "pick up the hex bolt",
    "episode":       [...],          # obs/action timesteps, §15.1 shape
    "outcome":       "fail",
    "failure_mode":  "no-grasp",     # from a fixed vocabulary
    "duration_s":    None,           # filled only on success
}
```

The `failure_mode` field draws from a small fixed list you settle before running: no-grasp, wrong-object, dropped-in-transit, missed-placement, collision, timeout. A fixed vocabulary is what turns a pile of failures into a confusion table, and a confusion table is what turns "75%" into "it grasps fine but places short," which is a sentence you can act on.

Sim first, then real, and know what the sim number is worth

If a simulator covers anything close to your task, evaluate there first, because sim buys you the two things real hardware refuses: reproducibility and volume. You can run five hundred seeded trials overnight and get a confidence interval so tight it is almost a point estimate. The chapter's own hands-on exercise leans on this, reproducing an OpenVLA evaluation in SimplerEnv across five seeds and computing intervals over them, and that is the right first move for any new policy.

The catch is the sim-to-real gap from §7.5. A sim success rate is an honest measurement of behavior in the simulator and an optimistic proxy for the real robot, sometimes wildly so. Treat the sim number as a screen, not a verdict: it is excellent for killing bad candidates cheaply and for catching regressions between training runs, and it is not a substitute for the smaller, noisier, more expensive real-robot number you report when it actually matters. A useful habit is to run a handful of real trials on a couple of sim tasks and note the gap; if sim says 90% and hardware says 55% on the same task, you have calibrated how much to discount every other sim figure.

The checklist

Strip away the reasoning and a defensible home-grown evaluation is this. Write the decision sentence the evaluation must let you finish. Choose tasks that spread across the difficulty and the variation axes you care about, avoiding the ceiling and the floor. Fix the success criterion, the initial-state distribution, the trial count, and the interleaving before the first rollout, all in writing. Log each trial as a full episode with a failure mode from a fixed list, not as a boolean. Screen in sim for volume, confirm on real hardware for truth, and report success rate with a Wilson interval and time-to-completion as a median with spread.

None of this is exotic, and all of it is skipped constantly, which is why so many robot demos evaporate the moment someone else runs them. The discipline is the deliverable. A policy is only as trustworthy as the ruler you measured it with, and once that ruler holds, §15.7 closes the chapter by pulling the dataset, benchmark, and evaluation threads back into a single picture before Part 5 turns to building.

15.7 Summary

Part 4 spent six chapters treating data and evaluation as background: OpenVLA trained on 970,000 trajectories, Octo on 800,000, and the numbers that ranked one VLA above another came pre-computed from some benchmark you were asked to trust. Chapter 15 stopped taking either for granted. It opened the pile of trajectories to see what one episode actually is, followed the corpus up from a single WidowX arm to sixty pooled datasets, and then asked the question every one of those model comparisons quietly assumed had an answer: when a paper says a policy "works," what was measured, and would it survive someone else running it? §15.1 took an episode apart array by array, §15.2 scaled that to Open X-Embodiment and its fleet-scale successors, §15.3 set LeRobot beside it as the second storage lineage, §15.4 walked the sim benchmarks the field quotes, §15.5 turned real-robot success into a statistics problem, and §15.6 turned all of it into a procedure you can run on a robot no leaderboard covers. The chapter's job was to make sure that when Part 5 fine-tunes a model and asks whether it improved, you know what an honest answer to that question costs.

The ideas worth carrying forward

A robot episode is a list of timestamped dictionaries, and the anatomy is the same everywhere even when the conventions are not. §15.1 used one BridgeData V2 episode (Walke et al., 2023) because it is small enough to inspect by hand. Each timestep carries an observation that is more than an image, an action written in some lab's private convention of units and frames, a language instruction that arrived by a route worth knowing, and a scalar or two of proprioception. The lesson that carries is that the shape is universal and the semantics are not: two datasets can both store a seven-vector action where one means end-effector deltas in meters and the other means joint velocities in radians per second, and nothing in the file tells you which. That gap is the seed of every problem in §15.2, and it is why the episode format you log your own evaluations in should match this shape, so your eval data is reusable as training data later.

Open X-Embodiment is plumbing, not a dataset, and the plumbing is the hard part. §15.2 scaled one pile of dictionaries up to sixty. OXE (arXiv:2310.08864) pooled roughly sixty datasets from twenty-one institutions, covering twenty-two robot embodiments and more than a million real trajectories, and its contribution was less the data than the harmonization: re-serializing everyone's incompatible conventions into one format a single policy could train across. That harmonization is exactly the MDP-to-robot translation problem from §5.5 wearing work clothes, and it is imperfect on purpose, because forcing every embodiment into one action space throws away detail no matter how you do it. What has succeeded OXE is not a bigger branded pool but large teleoperation fleets, AGIBot's Genie-1 collection, bimanual YAM rigs, Unitree G1 data, feeding newer foundation models directly, and LingBot-VLA 2.0's one-policy-across-twenty-morphologies result is the clearest sign that the pooled corpus was a phase, not an endpoint.

LeRobot is the same episode with a different address and a different audience. §15.3 drew the contrast that matters. Where OXE grew out of TensorFlow, RLDS, and academic labs comfortable in that ecosystem, LeRobot (Cadene et al., 2024) is PyTorch-first and built for the person with one low-cost arm who wants to record a hundred episodes this afternoon and train tonight. It bundles a dataset format, a hosting hub, pretrained policies you load in two lines, and driver support for hardware you can build from a parts list. The delta-timestamp query trick, asking for a window

of past and future frames by time rather than index, is the piece of engineering worth remembering, because it is what lets the same on-disk episode feed a diffusion head that wants a sixteen-step action horizon and a single-step policy that wants one. The community-dataset count going from a handful to several thousand in about a year is the evidence that lowering friction moved more than any architecture did.

A sim benchmark is a fixed world sold as a fair comparison, and each one trades honesty for reproducibility differently. §15.4 walked four. LIBERO (arXiv:2306.03310) is the fine-tuning yardstick, organized along axes of object, spatial, goal, and long-horizon variation that are worth stealing even if you never run it. CALVIN pushes on long horizons and language chaining; RoboCasa buys scale through generated assets; SimplerEnv exists for one purpose, to make a sim number predict the real number, by rebuilding real evaluation scenes closely enough that the ranking transfers. The through-line is that reproducibility and realism pull against each other, and every benchmark picks a point on that line and hides the cost of its choice in the tasks it happens not to include. RoboArena (Atreya, Pertsch et al., 2025) is the reaction: give up on sim, run distributed preference-scored evaluation across many labs on real hardware, and accept noise as the price of credibility.

A real-robot success rate is a sample estimate of a probability you cannot observe, and reporting it without an interval is the most common quiet lie in the literature. §15.5 was the statistics chapter. You run a policy N times, it succeeds k times, and k/N is not the policy's success rate; it is one draw from a binomial whose spread is wide at the sample sizes robotics can afford. Twenty trials at 15 successes is consistent with a true rate anywhere from about 53% to 88% at 95% confidence, which is why a Wilson interval belongs on every reported number and why two policies that differ by one or two successes out of twenty differ by nothing you can defend. The variance is real and physical, drifting light and grease and a two-centimeter shift in object pose, and the fixes are procedural: interleave policies trial by trial, log failure modes from a fixed vocabulary rather than a bare boolean, and report time-to-completion as a median with spread rather than a mean that one timeout destroys.

What you should be able to do now

Four things, in the order Part 5 uses them.

You should be able to *read a dataset off disk and know what every field means and does not mean.* Given an unfamiliar episode, you can name the observation, action, instruction, and proprioception channels, and you know to check the action convention before trusting it, because a seven-vector that looks like end-effector deltas might be joint velocities and the file will not warn you. This is the inspection §15.1 walked, and it is what keeps you from feeding a policy actions in the wrong units and blaming the architecture.

You should be able to *explain what pooling robot data costs, and what replaced pooling.* You can say why Open X-Embodiment's real contribution was harmonization rather than collection, why that harmonization loses embodiment-specific detail, and why the field moved from one branded corpus toward large teleop fleets and one-policy-across-many-morphologies results. You can also place LeRobot as the second lineage and say who each format is actually for.

You should be able to *tell an honest sim benchmark result from a misleading one.* You know what LIBERO, CALVIN, RoboCasa, and SimplerEnv each measure and each ignore, you know that a sim number is a screen and not a verdict because of the sim-to-real gap from §7.5, and you know why RoboArena gave up on sim to buy credibility. Handed a table of benchmark scores, you can say which comparisons the benchmark actually supports.

You should be able to *design and run an evaluation for a policy nobody has benchmarked.* You start from the decision sentence the evaluation must let you finish, choose tasks that spread across difficulty rather than clustering at the ceiling, fix the success criterion and initial-state distribution and trial count in writing before the first rollout, log each trial as a full episode with a failure mode, screen in sim for volume, and confirm on hardware for truth. This is the §15.6 procedure, and it is the deliverable Chapter 16 will lean on the moment it asks whether fine-tuning helped.

Where the chapter has set up the rest of the book

Chapter 15 is the hinge into Part 5, and it hands forward a specific pairing. The whole point of Chapter 16 is to fine-tune a base VLA onto your own robot, and every step of that depends on this chapter: you need to build a teleop dataset in a format (§15.1, §15.3) that a training loop can consume, and you need an evaluation (§15.5, §15.6)

honest enough to tell you whether the fine-tune moved the number or just moved the noise. §16 will say "pick a base model and collect data that does not waste your time," and the reason that sentence is not empty is that this chapter told you what good data and a defensible measurement look like.

The safety thread runs into Chapter 17. §15.6's rule about including a task the policy should refuse, and §15.5's insistence on logging failure modes rather than a boolean, are the seeds of treating safety as its own evaluation axis rather than a footnote to success rate. When §17.3 builds A/B evaluation on hardware, it is running the interleaved, blinded protocol §15.5 argued for, on higher stakes. When §17.4 builds logging and rollback, it is extending the episode-logging discipline from §15.6 into a production system that watches a running robot instead of a test harness.

One thread points at data collection specifically. §15.6's move, logging every evaluation rollout as a full episode so a failure becomes replayable and a hard task becomes a candidate for the next collection session, turns evaluation into a failure-mining pipeline that feeds §16.2's data-collection loop. Evaluation and data collection are not two activities in this book; they are one loop, and the rollouts where your policy is weakest are the most valuable data you will ever record.

What the chapter has not covered

Two omissions worth naming. The chapter treated evaluation almost entirely as measuring task success, and said little about evaluating the properties that do not reduce to a success rate: how smooth the motion is, how hard the policy is on the hardware, whether it fails safely or violently, how it degrades as the scene drifts from the training distribution. Some of those get picked up as safety in Chapter 17, but the general problem of scoring a policy on axes other than "did it finish" is one the field has barely standardized, and this book does not pretend otherwise.

The chapter also stayed on manipulation and mostly on single-arm tabletop tasks, which is where the datasets and benchmarks are richest. It touched bimanual and whole-body data through the teleop-fleet successors in §15.2 but did not develop how you evaluate a loco-manipulation policy, where a "failure" can mean the robot fell over rather than missed a grasp, and where the statistics of §15.5 have to share a page with the real-time jitter budget of §14.4. That combined evaluation problem is still an open one, and §18 returns to why the humanoid case makes it harder.

Chapter 15's contribution to the book's argument is to put a floor under every number in Part 4 and every claim in Part 5. A model comparison is only as good as the dataset conventions underneath it and the evaluation protocol on top of it, and this chapter showed both: what a trajectory contains and where its conventions can mislead, how the field pools and formats data and what that costs, what the sim benchmarks measure and hide, and why a real success rate is an estimate that needs a confidence interval before it means anything. With data and evaluation understood at this depth, Part 5 can build, because now you can tell whether what you built actually works.

§15.x closes the chapter with a hands-on exercise, reproducing an OpenVLA evaluation in SimplerEnv across five seeds and computing a confidence interval over the runs, so the gap between one sim number and a defensible one becomes something you measure rather than something you take on trust, followed by the chapter's full reading list.

15.x Hands-on exercise + chapter references

§15.5 made a claim you were asked to take on faith: that a single success rate, reported without an interval, hides how little you actually know. Twenty trials, fifteen successes, "75%," and the true rate sitting anywhere from the low fifties to the high eighties. This exercise turns that claim into an interval you compute from runs you did yourself. The TOC states the target plainly: reproduce a published OpenVLA evaluation in SimplerEnv, then compute confidence intervals over five seeds. You are going to run the same policy on the same task five times, changing only the random seed, and watch the "one number" the leaderboard prints fan out into a spread wide enough to change what you would conclude. If §15.5 was overselling, the five seeds will land on top of each other and the interval will be a point. They will not, and the gap between the one number and the honest one is the whole lesson.

There is a practical fork here, and it is worth naming before you start. Running OpenVLA in SimplerEnv for real wants a GPU and a checkpoint download, and it will keep a machine busy for an afternoon. The statistical payoff does not need any of that. So the exercise splits: 15.x.1 and 15.x.2 are the full reproduction for readers with the hardware, and 15.x.3 onward, the part that actually teaches the confidence interval, runs on a laptop CPU from the

per-seed counts, whether those counts came from your own GPU run or from the recorded numbers below. Do the full run if you can. If you cannot, skip to 15.x.3 with the recorded counts and you lose nothing of the point.

What you need

For the full path: a CUDA GPU with enough memory for the 7B OpenVLA checkpoint, the SimplerEnv package (Li et al., 2024), and the OpenVLA weights (arXiv:2406.09246) from the Hugging Face hub. For the statistics path: Python, NumPy, SciPy, and nothing else. No robot either way, because SimplerEnv is exactly the sim-for-real substitute §15.4 built the case for; the physical WidowX and Google robot setups it mirrors are the ones whose real numbers OpenVLA already published against.

Exercise 15.x.1 — Reproduce one seed

Start by running a single evaluation and checking that your number lands near the paper's. SimplerEnv ships the Google-robot and BridgeData/WidowX task suites already matched to their real-world counterparts, so the setup is mostly wiring a policy to an environment and looping.

```
import numpy as np
import simpler_env
from simpler_env.policies.openvla import OpenVLAInference  # package's wrapper

def evaluate(env_name, policy, n_episodes=25, seed=0):
    env = simpler_env.make(env_name)
    rng = np.random.default_rng(seed)
    successes = 0
    for _ in range(n_episodes):
        obs, _ = env.reset(seed=int(rng.integers(1 << 31)))
        policy.reset()
        done = False
        while not done:
            action = policy.step(obs["image"], obs["instruction"])
            obs, _, terminated, truncated, info = env.step(action)
            done = terminated or truncated
        successes += int(info.get("success", False))
    return successes, n_episodes

policy = OpenVLAInference(checkpoint="openvla/openvla-7b")
k, n = evaluate("google_robot_pick_coke_can", policy, n_episodes=25, seed=0)
print(f"seed 0: {k}/{n} = {k / n:.0%}")
```

Two things to check before you trust anything. Your single-seed number should land in the neighborhood of OpenVLA's published SimplerEnv result for that task, not on top of it; a few points off is normal and is itself the first hint of what 15.x.2 is about. And the task string must match the setup the paper reported, because SimplerEnv's whole promise from §15.4 is scene-by-scene fidelity, and a slightly different variant is a different number.

Exercise 15.x.2 — Run five seeds

Now change one thing, the seed, and hold everything else fixed. Same checkpoint, same task, same episode count. The seed moves object initial poses, and nothing else, which is precisely the initial-condition variance §15.5 said bites hardest.

```
results = {}
for seed in range(5):
    k, n = evaluate("google_robot_pick_coke_can", policy, n_episodes=25, seed=seed)
    results[seed] = (k, n)
    print(f"seed {seed}: {k}/{n} = {k / n:.0%}")
```

Write the five fractions down. If your run produced something like 19/25, 21/25, 16/25, 20/25, 18/25, you are looking at a range from 64% to 84% from nothing but the seed. That spread is not a bug in SimplerEnv and it is not your policy being unstable; it is what a success rate is, a draw from a distribution, and §15.5 told you to expect exactly this. Readers on the statistics-only path: use those five counts as your recorded numbers and carry on.

Exercise 15.x.3 — Put an interval on each seed

Here is where the chapter's argument gets settled. For each seed you have k successes out of n. Compute the Wilson interval §15.5 recommended, the one that behaves near 0 and 1 where the textbook normal approximation falls apart.

```
from scipy.stats import norm

def wilson(k, n, conf=0.95):
    if n == 0:
        return (0.0, 1.0)
    z = norm.ppf(1 - (1 - conf) / 2)
    p = k / n
    denom = 1 + z**2 / n
    center = (p + z**2 / (2 * n)) / denom
    half = (z / denom) * np.sqrt(p * (1 - p) / n + z**2 / (4 * n**2))
    return center - half, center + half

recorded = {0: (19, 25), 1: (21, 25), 2: (16, 25), 3: (20, 25), 4: (18, 25)}
for seed, (k, n) in recorded.items():
    lo, hi = wilson(k, n)
    print(f"seed {seed}: {k/n:.0%}  95% CI [{lo:.0%}, {hi:.0%}]")
```

Every one of those intervals is roughly twenty points wide, because n is 25 and there is no arithmetic that makes 25 trials narrow. Read two seeds against each other: seed 2 at 64% and seed 1 at 84% look like a real difference until you notice their intervals overlap heavily, which means the "difference" is the seed talking, not the policy. This is the same trap as comparing two policies that differ by a trial or two, drawn here from one policy against itself. If a single policy against itself can swing twenty points on the seed, a two-point gap between two different policies is worth nothing.

Exercise 15.x.4 — Pool the seeds

The fix for a wide interval is more trials, and five seeds of 25 is 125 trials you can pool, provided the seeds were independent draws from the same setup, which by construction they were. Sum the successes, sum the episodes, and put one interval on the pooled count.

```
K = sum(k for k, _ in recorded.values())
N = sum(n for _, n in recorded.values())
lo, hi = wilson(K, N)
print(f"pooled: {K}/{N} = {K/N:.0%}  95% CI [{lo:.0%}, {hi:.0%}]")
```

The pooled point estimate sits near the mean of the five, around 75%, but the interval has tightened from twenty-odd points to roughly thirteen, because n went from 25 to 125. That shrinkage, proportional to the square root of the sample size, is the only lever you have; there is no clever estimator that gets you a tight number from few trials. If you want the interval to halve, you run four times as many episodes, and now you understand in your fingers why §15.5 called sample size the thing robotics cannot afford and why RoboArena (Atreya, Pertsch et al., 2025) distributes the cost across labs instead of asking one lab to run a thousand rollouts alone.

Exercise 15.x.5 — Compare two policies honestly

The payoff. Repeat 15.x.2 for a second policy on the same task, Octo (arXiv:2405.12213) is the natural partner since SimplerEnv ships it too, pool each policy's five seeds, and ask whether their intervals overlap.

```
octo_recorded = {0: (14, 25), 1: (13, 25), 2: (17, 25), 3: (12, 25), 4: (15, 25)}
Ko = sum(k for k, _ in octo_recorded.values()); No = sum(n for _, n in octo_recorded.values())
print("OpenVLA:", wilson(K, N))
print("Octo:   ", wilson(Ko, No))
```

If the two pooled intervals are disjoint, you can claim one policy beats the other on this task and defend it. If they overlap, you cannot, no matter how far apart the point estimates look, and reporting the raw gap anyway is the quiet lie §15.5 named. Run the comparison the honest way from §15.6 when it is on real hardware: interleave the two policies trial by trial rather than batching, so warm motors and drifting light hit both equally. In sim the seed does that work for you, which is one more reason SimplerEnv is where you iterate and hardware is where you confirm.

Chapter 15 reading list

Cited across §15.1–§15.7, grouped by the job each reference does. Full entries for everything in the book live in Appendix E.2; this is the chapter-local subset.

The datasets

- Open X-Embodiment Collaboration, Padalkar, A., et al. (2023). "Open X-Embodiment: Robotic Learning Datasets and RT-X Models." arXiv:2310.08864. The spine of §15.2: roughly sixty datasets, twenty-one institutions, twenty-two embodiments, and the point that the contribution was harmonization, not collection.
- Walke, H., et al. (2023). "BridgeData V2: A Dataset for Robot Learning at Scale." CoRL 2023. §15.1's worked example, the single WidowX episode small enough to inspect array by array.
- Cadene, R., Alibert, S., Soare, A., et al. (2024). "LeRobot: State-of-the-art machine learning for real-world robotics in PyTorch." github.com/huggingface/lerobot. §15.3's second storage lineage, PyTorch-first and aimed at the person with one cheap arm; the delta-timestamp query is the piece of engineering to remember.
- Shukor, M., et al. (2025). "SmolVLA." arXiv:2506.01844. §15.3's example of a policy that ships inside the LeRobot ecosystem and loads in two lines.

The benchmarks

- Liu, B., Zhu, Y., Gao, C., et al. (2023). "LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning." arXiv:2306.03310. §15.4 and §15.6's source for the four-axis factoring of difficulty, object, spatial, goal, long-horizon, worth stealing as a coverage checklist even if you never run it.
- Mees, O., Hermann, L., Rosete-Beas, E., Burgard, W. (2022). "CALVIN: A Benchmark for Language-Conditioned Policy Learning for Long-Horizon Robot Manipulation." IEEE RA-L. §15.4's long-horizon chaining benchmark and the ABC→D split that separates a policy that learned the task from one that memorized the rooms.
- Nasiriany, S., Maddukuri, A., Zhang, L., et al. (2024). "RoboCasa: Large-Scale Simulation of Everyday Tasks for Generalist Robots." RSS. §15.4's scale-through-generated-assets benchmark, where memorization stops being a winning strategy.
- Li, X., Hsu, K., Fu, J., et al. (2024). "Evaluating Real-World Robot Manipulation Policies in Simulation (SimplerEnv)." CoRL. §15.4's sim-that-predicts-real benchmark and the environment this exercise runs in; its measure of merit is correlation with the physical rig, not raw difficulty.

The policies under the microscope

- Kim, M. J., Pertsch, K., Karamcheti, S., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246. The policy you reproduce in 15.x.1–15.x.2 and the 970,000-trajectory training figure §15.7 uses to frame the whole chapter.
- Ghosh, D., Walke, H., Pertsch, K., et al. (2024). "Octo: An Open-Source Generalist Robot Policy." arXiv:2405.12213. 15.x.5's comparison partner, and §15.7's second data point for what pretraining scale buys.

Real-robot evaluation

- Atreya, P., Pertsch, K., et al. (2025). "RoboArena: Distributed Real-World Evaluation of Generalist Robot Policies." §15.4, §15.5, and §15.6's recurring reference: the Chatbot-Arena-style pairwise, distributed protocol that gives up sim's reproducibility to buy a signal no single lab can quietly train against.

The survey that frames the space

- Zhang, J., et al. (2026). "Vision-Language-Action in Robotics: A Survey of Datasets, Benchmarks, and Data Engines." arXiv:2604.23001. §15.1–§15.3's framing reference for how the field's data and benchmarks fit together.

Chapter summary

Chapter 15 put a floor under every number Part 4 quoted and every claim Part 5 will make. You can now open a robot episode and name what each field means, knowing a seven-vector action might be end-effector deltas or joint velocities and the file will not tell you which. You can explain what pooling data cost Open X-Embodiment, that its real work was harmonizing incompatible conventions, and why the field drifted from one branded corpus toward teleop fleets and LeRobot's low-friction path for a person with one arm. You can read a table of LIBERO, CALVIN, RoboCasa, and SimplerEnv scores and say which comparisons it actually supports and which it only decorates, and after this exercise you can do more than recite §15.5's warning about intervals, because you watched one policy against itself swing twenty points on the seed alone, tightened the interval by pooling, and learned to call two overlapping intervals a tie no matter how far apart their point estimates print. That last skill is the one Chapter 16 leans on the moment it fine-tunes a model and asks whether the number moved or the noise did.

Chapter 16 — Fine-tuning a VLA for your robot

16.1 Picking a base model: OpenVLA, Octo, or a smaller distilled checkpoint

Part 4 handed you a shelf of trained VLAs and explained how each works. Part 5 asks a blunter question: which one do you download and fine-tune tonight, on the robot you actually own? The honest answer is that the choice is mostly made for you by four constraints you cannot wish away, and only after those have done their filtering does taste enter. Those constraints are the control rate your robot needs, the GPU you have to serve the model on, the shape of the data you can collect, and the license you are allowed to build on. Get those straight and the field of candidates usually collapses to one or two.

Start with the constraint that kills the most options first, because it is the one people ignore until their robot is jerking around: the control rate. §12.2 gave OpenVLA's honest latency, about 6 Hz on an RTX 4090 in bfloat16, and §14.4 explained why that number decides everything downstream. A tabletop pick-and-place that tolerates a deliberate, one-action-per-sixth-of-a-second cadence is fine on a 7B model. A robot that has to react to contact, correct a slipping grasp, or hold a balance loop is not, and no amount of fine-tuning fixes a base model whose forward pass is slower than the loop it has to close. So before comparing checkpoints on quality, write down the frequency your task demands, and treat any model that cannot hit it as disqualified rather than as a fixer-upper.

The three tiers on the shelf

The open checkpoints worth starting from in 2026 sort into three rough tiers by size, and the tiers matter more than the individual names because they trade the same things against each other.

The 7B generalists are OpenVLA (arXiv:2406.09246) and $\pi 0$ (arXiv:2410.24164). These are the models with the most competence per download: trained on the largest pools, best at absorbing a new task from few demonstrations, and the ones a paper will compare against. They also want a real GPU to serve, land in the single-digit-hertz range for the token-head ones, and take the longest to fine-tune. OpenVLA is the safe default here, and most of this chapter uses it, because its fine-tuning story is the most thoroughly documented open recipe in existence and its decision table (§12.2) is something you can act on directly.

The mid-size generalists are Octo, at 27M and 93M parameters (arXiv:2405.12213). Octo is an order of magnitude smaller than OpenVLA, carries a diffusion action head instead of a token classifier, and was built from the start to be fine-tuned onto new observation and action spaces. It runs faster, fits on smaller hardware, and its diffusion head gives you smoother, more multimodal trajectories than OpenVLA's discrete bins can (the §10.5 argument, made concrete). The cost is raw generalization: a 93M model has not memorized the visual world the way a 7B backbone has, so it leans harder on your fine-tuning data to make up the difference.

The small, deployment-oriented models are SmolVLA (arXiv:2506.01844) and TinyVLA (arXiv:2409.12514). SmolVLA is a roughly 450M-parameter VLA built inside the LeRobot ecosystem (§15.3) and trained largely on community-contributed datasets, explicitly aimed at the person with one low-cost arm and a single consumer GPU. TinyVLA pushes the same direction, small backbone, fast inference, data-efficient fine-tuning. Neither will top a leaderboard against a 7B model on hard generalization, and neither is trying to. They exist so that the fine-tune-deploy loop fits on

hardware a hobbyist or a small lab already has, and they close the loop fast enough that a few hundred milliseconds of latency stops being the wall it is for the big models.

What actually separates them

Four axes decide which tier and which checkpoint you land on. None of them is "which scored highest," because the highest-scoring model on someone else's benchmark can be the wrong tool for your robot.

The first is the action representation, and it is the one with the longest downstream shadow. OpenVLA emits discrete action tokens; Octo, π0, and SmolVLA carry continuous heads (diffusion or flow-matching). If your task needs smooth, high-frequency, multimodal motion, a continuous head starts you closer to the goal and a token head fights you the whole way, because 256 bins per dimension (§11.3) put a hard resolution floor under how fine a motion the policy can even represent. If your task is slow and mostly unimodal, the token head's simplicity is a feature, and OpenVLA's discrete recipe means every part of the training stack is boring and well-trodden.

The second is fine-tuning economics. §12.2's table is the number to internalize: OpenVLA with LoRA reaches about 68% where full fine-tuning of just the last layer collapses to 30%, and LoRA does it with roughly an 8x compute reduction on a single A100. That is what makes the 7B tier reachable at all for a normal lab. Octo and the small models are cheaper still per step simply because they are smaller, so the calculus is less about clever adaptation and more about whether you can afford to touch the whole network. If your compute budget is one consumer GPU and a weekend, that budget alone may push you from the 7B tier down to SmolVLA regardless of what you would prefer on quality.

The third is the observation and action interface, meaning how much surgery it takes to make the model's inputs and outputs match your robot. OpenVLA expects a single third-person image and a language string, and emits a 7-DoF end-effector action; it has no slot for a wrist camera, a proprioceptive state vector, or a bimanual action space without you modifying the model. Octo was designed for exactly this flexibility, with input tokenizers you can add or swap and an action head you can re-target to a new action dimension, which is why it is often the better starting point for an embodiment that does not look like a single 7-DoF arm. Check this early. A model whose interface is close to your robot's saves you the most error-prone part of the whole job.

The fourth is licensing and provenance, the constraint people discover last and regret most. Weights, training code, and the datasets underneath all carry licenses, and a permissive research checkpoint is not automatically something you can ship in a product. OpenVLA and Octo released weights and code openly and are widely used in commercial-adjacent settings, but the datasets pooled into Open X-Embodiment (§15.2) carry their own terms, and a few contributing datasets are research-only. If your fine-tune is going into anything you sell, read the license on the base weights and on the pretraining corpus before you invest a month of data collection on top of them.

A short decision procedure

Put the four axes in order and the choice usually makes itself. Run through them like a filter.

First, does the model meet your control rate on the GPU you can deploy on? If not, drop to a smaller tier; a fast, slightly-less-capable policy beats a brilliant one that misses its deadline. Second, does its action head match the motion your task needs, smooth and multimodal versus slow and simple? Third, can you afford to fine-tune it, given LoRA for the 7B tier and full fine-tuning for the small tier? Fourth, does its input/output interface sit close to your robot's cameras and action space, or will you be doing model surgery before you even start? Fifth, does the license permit what you actually intend to do with the result?

For most readers of this book, the default that survives that filter is OpenVLA fine-tuned with LoRA, and that is the setup the rest of the chapter and the hands-on exercise are built around. It is the best-documented recipe, it runs on hardware a lab owns, and its failure modes are the ones the literature has mapped. If your robot needs smoother or faster control than a 6 Hz token head can give, Octo is the reach for its diffusion head and flexible interface. If your entire budget is one cheap arm and one consumer card, SmolVLA is the model that was built for your exact situation and will not punish you for it.

There is a temptation, once you have picked, to treat the base model as sacred and pour all your effort into the fine-tune. Resist it. The base model is a starting distribution, not a finished product, and the single largest lever on your

final success rate is not which of these checkpoints you chose but the quality of the data you fine-tune it on. That is where a project quietly succeeds or wastes a month, and it is what §16.2 is about: building a teleoperation dataset that earns its collection time instead of burning it.

16.2 Building a teleop dataset that does not waste your time

The most expensive mistake in this whole chapter is collecting the wrong 200 episodes. Teleoperation is slow, it needs a human at the controls, and every session drifts the setup a little, so a dataset that turns out to be unusable is not just wasted GPU time, it is a wasted afternoon of your life you cannot get back by re-running a script. This section is about spending that afternoon well. The governing idea is simple to state and hard to obey: your fine-tuning data has to look like the situations the deployed policy will face, including the ones where things go wrong, and most first datasets fail because they contain only the situations where things went right.

Why clean demonstrations make bad data

The instinct when you sit down to teleoperate is to perform. You reset the scene neatly, you drive the arm along a smooth confident path, the gripper closes at exactly the right moment, and you log a beautiful success. Do that two hundred times and you have built a dataset that teaches the policy one thing: how to be perfect from a starting state it will never see twice. The trouble is behavior cloning, and §6.3 already named it. A policy trained only on expert trajectories has never seen the states that follow a small mistake, so the first time it drifts even slightly off the demonstrated path, it is in unfamiliar territory and the errors compound. Your gorgeous demonstrations are the exact reason it falls apart.

The fix is not to demonstrate badly. It is to demonstrate recovery. When the grasp misses, do not reset and pretend it did not happen; keep teleoperating, drive the arm back, re-approach, and complete the task. Now the dataset contains the state "gripper is near the object but not aligned" together with the action "back off and re-approach," which is precisely the correction the policy needs when it makes that mistake on its own. This is the human-in-the-loop intuition behind DAgger (Ross et al., 2011) applied at collection time rather than as a separate correction loop: you are deliberately seeding the data with the off-distribution states and their fixes, so the policy learns to get back on track instead of only how to stay on it.

Coverage beats volume

The second failure mode is a dataset that is large but narrow. Two hundred episodes of the same mug in the same spot under the same lighting is two hundred near-duplicates, and the policy that trains on it will fail the moment the mug moves five centimeters or a cloud passes the window. §15.5 called this initial-condition variance and showed it is where real-robot success rates go to die. Your data is the only place the policy learns to tolerate it, so the variance has to be in the data on purpose.

Vary the things the deployed robot will see vary. Object position and orientation across the reachable workspace, not clustered in one comfortable zone. Lighting, if your deployment site's lighting is not controlled. Distractor objects in the scene, so the policy learns which object the instruction refers to rather than "the only thing on the table." The specific instruction phrasing, if you want language robustness, because a policy trained only on "pick up the red block" can be surprisingly brittle to "grab the red one." A useful discipline is to grid the workspace: mentally divide the reachable area into a handful of zones and make sure each zone gets its share of episodes, rather than letting your hand default to the center where teleoperation is easiest. Fifty well-spread episodes routinely beat two hundred clustered ones, and they take a quarter of the time to collect.

Match the format to the model, from the first episode

Before you record anything, settle the format, because reformatting a dataset after the fact is the kind of tedious, bug-prone work that eats a day and introduces silent errors. §15.1 laid out the episode anatomy: a list of timesteps, each a dictionary pairing an observation with the action that followed. Your recording has to produce that shape, and it has to produce it in the units and frames your base model expects, which §16.1 told you to check early. This is where the single most common silent failure in the whole pipeline lives, and it deserves its own paragraph.

The action-convention mismatch is the bug that trains a policy that looks fine and behaves like it is drunk. Your teleoperation system logs actions in some convention, absolute end-effector pose, or delta pose, in meters or millimeters, in the base frame or the camera frame, with the gripper as a binary or a continuous width. OpenVLA (arXiv:2406.09246) expects 7-DoF end-effector deltas normalized to its training statistics. If you log absolute poses and feed them to a model expecting deltas, nothing errors out; the numbers are the right shape and the wrong meaning, and the loss will even go down as the policy learns to predict your (wrong-convention) actions. You will discover the problem only on the robot, when it moves confidently to the wrong place. Fix the convention before collection and verify it by replaying a logged episode open-loop on the robot: if the replay reproduces the demonstrated motion, your convention is right, and if it does not, you just saved yourself the fine-tune.

The LeRobot format (§15.3, Cadene et al., 2024) is worth adopting here for a concrete reason beyond tidiness: it is what several base models load natively, its dataset tooling handles the timestamp bookkeeping for you, and its delta-timestamp query lets the same recorded episode feed a policy that wants a single action and one that wants a sixteen-step horizon. Recording straight into that format means your data is loadable by the training loop with no conversion step, which removes an entire class of the reformatting bugs above.

How much data, and how to tell when you have enough

There is no universal number, but there is a shape to the answer. For fine-tuning a 7B generalist that already knows the visual world, the surprising finding across OpenVLA-style fine-tunes is that a few tens to low hundreds of episodes per task often move the success rate substantially, because the base model is not learning to see, only learning to map what it already sees onto your robot's actions. For a small model that has not memorized as much, you will need more, because it is doing more of the learning from your data alone. The chapter's exercise ablates dataset size directly so you can watch the curve for yourself rather than trust a number I quote.

The practical way to find your enough is to collect in rounds and let evaluation tell you when to stop. Collect a first batch, fine-tune, evaluate with the honest protocol from §15.5 and §15.6, and look at where it fails. Then, and this is the move that makes the whole chapter cohere, collect your next batch targeting those failures specifically. If the policy fails when the object is at the left edge, spend the next session putting the object at the left edge. Evaluation and data collection are one loop, not two activities, and the rollouts where your policy is weakest are the most valuable episodes you will ever record. A dataset built this way grows toward the policy's actual weaknesses instead of padding its existing strengths.

A collection checklist

Before the session, decide and write down: the exact action convention and units, verified by open-loop replay; the observation the model expects, cameras and any state vector, at the right resolution; the instruction phrasing set; and the workspace zones you will cover. During the session, include recovery behavior rather than resetting after every fumble, spread episodes across the zones instead of the comfortable center, and vary the conditions you know the deployment will vary. After the session, replay a random few episodes to confirm the logs are sane, and check the dataset loads in your training pipeline before you collect the next batch, not after you have collected all of them.

Do that and the two hundred episodes you collect are two hundred you will actually use. Now they need to go into the model, and the question of which parts of a 7B network you update, and at what cost, is the one that decides whether your single GPU is enough. That is §16.3: LoRA against full fine-tuning against touching only the action head.

16.3 LoRA vs. full fine-tuning vs. action-head-only

You have a 7B model and a dataset of 150 episodes. Now the question is what, exactly, gets updated when you train, because a 7B model has more parameters than your 150 episodes have supervision to move responsibly, and the three standard answers, update everything, update a small low-rank slice, or update only the action head, trade the same three things against each other: how much compute you spend, how much of the base model's competence you keep, and how far you can push the policy toward a task that looks nothing like its pretraining.

Full fine-tuning: the most capable and the most dangerous

Full fine-tuning updates every weight in the network, backbone included. It is the most expressive option, because nothing is held fixed, and on a large, diverse fine-tuning set it gives the best ceiling. It is also the option most likely to hurt you on a small one, for two reasons that both come down to having too much capacity chasing too little data.

The first is cost. Updating all 7B parameters of OpenVLA means storing gradients and optimizer state for all of them, which pushes memory well past a single consumer GPU and into multi-GPU or high-memory-datacenter-card territory. §12.2 put the LoRA-versus-full gap at roughly 8x compute, and that multiplier is the difference between a weekend on one card and a week on a cluster. For most readers that alone settles it.

The second is catastrophic forgetting, the failure mode where the model, in learning your task, overwrites the general competence that made it worth starting from. Train all 7B parameters hard on 150 episodes of one mug on one table and the model can forget how to generalize across objects it used to handle, because gradient descent has no reason to preserve capabilities your loss function never rewards. You end up with a model that aces your exact setup and has quietly become worse than the checkpoint you started from at anything else. On a large and varied fine-tuning corpus this is not a concern; on a small specialized one it is the default outcome unless you guard against it.

LoRA: the recommended default, and why it works

LoRA, low-rank adaptation (Hu et al., 2021, arXiv:2106.09685), is the method that makes fine-tuning a 7B VLA sane on a single GPU, and it is OpenVLA's own recommendation for good reason. The idea rests on one empirical observation: the update a fine-tune applies to a big weight matrix is usually low-rank, meaning it can be well approximated by the product of two much smaller matrices. So instead of updating the weight matrix W directly, you freeze it and learn a small additive correction, W + BA, where B and A are skinny matrices whose inner dimension r (the rank) is tiny, often 8, 16, or 32. During training only B and A get gradients; W never moves.

The consequences are worth spelling out because they explain every practical advantage. The number of trainable parameters drops by orders of magnitude, since two rank-16 matrices are a rounding error next to a full weight matrix, which is what collapses the memory and compute cost. The frozen W keeps the base model's competence intact by construction, because you literally cannot overwrite it, which is why LoRA resists the catastrophic forgetting that stalks full fine-tuning. And the learned adapter is small enough to store and swap, so you can keep one base model and a folder of task-specific adapters, loading whichever one you need, rather than a full 7B copy per task.

```
# LoRA applied to OpenVLA's backbone, using the PEFT library
from peft import LoraConfig, get_peft_model

config = LoraConfig(
    r=16,                       # rank: the one knob that matters most
    lora_alpha=16,              # scaling; commonly set equal to r
    target_modules=["q_proj", "v_proj"],   # which projections get adapters
    lora_dropout=0.05,
)
model = get_peft_model(base_openvla, config)
model.print_trainable_parameters()   # e.g. 0.1% of 7B is trainable
```

The rank r is the knob to understand, and the chapter exercise ablates it directly. A higher rank gives the adapter more capacity to fit your task, at more memory and more risk of overfitting a small dataset; a lower rank is cheaper and more conservative. The useful mental model is that r sets how much you are allowed to change the base model, and the right value scales with how far your task is from pretraining and how much data you have. A task close to the pretraining distribution with little data wants a small rank; a task far from it with a lot of data can use a larger one. §12.2's headline holds across a wide band of ranks, though: LoRA at a sensible rank matched full-quality sandwich fine-tuning for a fraction of the cost, which is why it is the default this book recommends and the exercise walks you through.

One detail from §12.2 that interacts with LoRA and is easy to get wrong: OpenVLA found the vision encoder had to adapt during fine-tuning, not stay frozen, because internet-pretrained visual features lack the fine spatial geometry a manipulation policy needs. So when you configure LoRA, make sure the adapters actually reach the visual pathway

or that the encoder is otherwise trainable, rather than adapting only the language backbone and leaving the eyes locked. A LoRA config that targets only the LLM's attention projections and skips the encoder reproduces the weak "frozen vision" result from the decision table, near 47%, and you will wonder why your well-collected data is underperforming.

Action-head-only: the cheapest, for the narrowest case

The third option updates only the action head and freezes the entire backbone. For OpenVLA this means touching just the repurposed output vocabulary slots; for a model like Octo (arXiv:2405.12213) with a separate diffusion head, it means training that head while the transformer trunk stays fixed. This is the cheapest thing you can do, trains in minutes, and needs almost no memory.

It is also the most limited, and §12.2 quantified how limited: OpenVLA's last-layer-only fine-tune landed around 30%, the worst option in the table. The reason is intuitive. If you only touch the head, the representation feeding it, everything the backbone computes about the scene, is frozen at its pretraining state, so the head can only re-slice features the model already extracts. That is enough when your task is genuinely just a re-mapping of the base model's existing competence onto a slightly different action space, and it is nowhere near enough when your robot's cameras, objects, or dynamics differ from pretraining in ways the frozen backbone never learned to represent.

Where action-head-only earns its place is a specific, common situation: adapting a model to a new action space or action dimension without changing what it perceives. Octo was designed with exactly this in mind, letting you attach a new action head for a robot with a different number of degrees of freedom and train just that head, precisely because the flexible-interface design of §16.1 assumed you would. If your robot sees roughly what the base model was trained to see but acts in a different space, head-only adaptation is a fast, honest first try. If it sees something new, you will need LoRA or more.

Choosing, in one paragraph

Default to LoRA. It is the setup that fits a single GPU, resists forgetting, produces swappable adapters, and matched full-quality fine-tuning in OpenVLA's own experiments, which is why the rest of this chapter uses it. Reach for full fine-tuning only when you have a large, varied dataset (thousands of episodes across many conditions), the compute to run it, and evidence that LoRA's capacity is actually the bottleneck rather than your data. Reach for action-head-only when you are re-targeting the action space of a model whose perception already matches your robot, and you want an answer in minutes. And whichever you pick, remember the failure the next section is about, because these methods do not fail loudly: a fine-tune can drive its training loss to the floor and still produce a policy that is worse than the base model on everything except a memorized demo. §16.4 puts that inside the sim-to-real loop, where the gap between a low training loss and a working robot is widest.

16.4 Sim-to-real fine-tuning loops

Real robot time is the scarcest resource in this whole enterprise. Every real episode costs a human minder, a physical reset, and a slice of wear on the hardware, and §15.5 showed that the statistics you need for an honest success rate demand more episodes than you will want to run. Simulation is the obvious relief valve: it is cheap, parallelizable, and resettable, and you can run five hundred seeded trials overnight. The catch is the sim-to-real gap from §7.5, the fact that a policy that works in simulation often does not survive contact with a real robot, because the rendering, the physics, and the sensor noise are all subtly wrong. This section is about building a fine-tuning loop that uses sim for what it is good at, volume, and hardware for what only it can give, truth, without letting the gap between them fool you.

Two jobs sim can do, and one it cannot

Simulation plays two distinct roles in a fine-tuning pipeline, and conflating them is where people go wrong. The first role is data augmentation: generating additional training episodes in sim to supplement a small real dataset, usually with domain randomization (Tobin et al., 2017) that varies textures, lighting, object poses, and physics parameters so the policy cannot lean on any single simulator artifact. The bet, the same one §7.5 laid out, is that if the policy

sees enough randomized variation, the real world looks like just another sample from the distribution, and the gap shrinks to something the fine-tune can cross. The second role is evaluation, the SimplerEnv (§15.4) use: running the fine-tuned policy in a sim built to correlate with real success, as a cheap screen before you spend hardware time.

The job sim cannot do is certify. A high sim number is evidence, not proof, and §15.4 was blunt that even SimplerEnv's correlation is a property of the policies it was validated against, breakable by a new architecture that games the visual match. So the loop below uses sim to decide what to test on hardware and to cheaply kill obviously-broken fine-tunes, and it uses hardware to decide what actually works. It never lets a sim number stand as the final answer.

The loop

The loop is the §15.6 procedure turned into a training cycle. One turn of it looks like this. You fine-tune on your current dataset, real plus any sim augmentation. You screen the result in sim across several seeds, computing the confidence interval from §15.x rather than trusting one number, and if it is clearly broken you fix it before touching the robot. You then run a small, honest real evaluation on the survivors, logging every trial as a full episode with a failure mode. You read the failures, and, the move that closes the loop, you collect or generate the next batch of data targeting exactly those failures, whether that means more real recovery episodes (§16.2) or more randomized sim episodes in the failing condition. Then you turn the crank again.

Two properties make this work. It spends real robot time only on policies that already survived the cheap screen, so hardware evaluation is confirming, not exploring. And it grows the dataset toward the policy's demonstrated weaknesses, which is the DAgger intuition (Ross et al., 2011) from §16.2 operating at the level of whole training rounds: the states where the policy fails are the states you add data for. A fine-tune that stalls is almost always one where this loop was skipped, where someone trained once, tested once, and concluded the method does not work.

Three failure modes, and how to tell them apart

Fine-tuning fails quietly. The loss goes down, the training curves look healthy, and the robot still does not work, because a low training loss measures how well the policy fits your data, not whether it does your task. Three failure modes account for most of what goes wrong, and the reason to name them is that they look similar on the robot and demand opposite fixes.

Over-specialization is the fine-tune that learned your dataset too literally and nothing around it. On the robot it presents as a policy that works beautifully in the exact conditions you collected, the same object, the same spot, the same light, and falls apart the instant anything shifts. This is catastrophic forgetting (§16.3) and narrow data (§16.2) showing up together: the model overwrote its general competence and your data did not cover the variation to replace it. The tell is a large gap between success on in-distribution starts and success on shifted ones. The fix is on the data and the method side, not the training-longer side: widen the coverage of the dataset and lower the LoRA rank or reduce training steps so you change the base model less. Training harder makes over-specialization worse, which is the counterintuitive part people get wrong.

Mode collapse is the policy that has stopped representing the multimodality the task actually has. Many manipulation tasks admit several valid solutions, grasp the mug from the left or the right, and behavior cloning on a dataset containing both, trained with a loss that rewards matching the average, can produce a policy that commits to the mean of the two: reaching straight down the middle and grasping nothing. On the robot it shows as a policy that freezes, jitters between options, or takes a blended action that satisfies no mode. This is exactly the failure §10.5 said continuous multimodal heads (diffusion, flow-matching) exist to prevent, so one fix is architectural, use or fine-tune a model whose head can represent multiple modes. The data-side fix is to reduce the ambiguity the policy has to resolve, by conditioning more precisely, sharper language or an added goal, so that each state maps to one clear action rather than an average of several.

Action-token mismatch is the convention bug from §16.2 wearing its consequences. If your logged actions are in a different convention, units, or frame than the base model expects, or if OpenVLA's normalization statistics do not match your robot's action range, the model trains happily on numbers that mean the wrong thing, and the loss drops because it is predicting your (mislabeled) actions accurately. On the robot it presents as confident, smooth motion to the wrong place, or motion at the wrong scale, over- or under-shooting consistently. It is the easiest of the three to prevent and the most embarrassing to diagnose late, because the fix is the open-loop replay check from §16.2

that you should have run before collecting the full dataset. If a fine-tune produces coordinated-but-wrong behavior, suspect the convention before you suspect the model.

The diagnostic that separates them is cheap and worth building into the loop. Evaluate on three sets: the exact training conditions, mildly shifted conditions, and a replay sanity check. A policy that fails the replay check has a token mismatch. A policy that passes training conditions but fails shifted ones has over-specialized. A policy that fails even the training conditions with frozen, blended, or hesitant motion has collapsed a mode. Same symptom on the surface, three different root causes, and reading the pattern saves you from applying the wrong fix and making things worse.

Co-training and the ratio that matters

When you do use sim data alongside real, the mixing ratio is a knob, and the safe default is to keep real data well represented rather than letting a flood of cheap sim episodes drown it out. A common pattern is co-training, sampling each training batch from both sources with real weighted up relative to its raw count, so the policy grounds on real observations while borrowing the sim data's coverage. Too much sim and the policy optimizes for simulator artifacts the domain randomization did not cover; too little and you have not bought any of the volume sim was supposed to give you. The chapter exercise does not go deep on this because a single low-cost arm rarely has a matching high-fidelity sim, but for setups that do, notably the SimplerEnv-covered embodiments, co-training with real weighted up is the setup that most reliably transfers.

You now have the pieces: a base model, a dataset built to cover and to recover, a fine-tuning method that fits your hardware, a loop that uses sim honestly, and a diagnosis for the three ways it fails. §16.5 compresses all of it into something you can pin above your desk, a one-page recipe card for pointing this whole process at a robot you have never fine-tuned for before.

16.5 A recipe card for new embodiments

The chapter's last learning objective is the most practical: write a one-page recipe card you could hand to someone starting a fine-tune on a robot neither of you has worked with before. This section is that card, followed by the reasoning that keeps it from being a cargo cult. The card is ordered, and the order is load-bearing, because the expensive mistakes in this chapter all come from doing a later step before an earlier one, collecting data before fixing the action convention, or training hard before checking coverage.

The card

Step 0 — Write the decision sentence. Before anything, write the sentence the whole project has to let you finish: "this policy picks up the bin and places it on the shelf, at least 80% of the time, across the six starting positions we care about." Everything downstream, how much data, which base model, what counts as success, is set by this sentence. Skipping it is how projects drift into collecting data for a task nobody defined.

Step 1 — Pick the base model by constraint, not by leaderboard (§16.1). Filter in this order: control rate on your deployable GPU, action-head match to your motion (smooth/multimodal versus slow/simple), fine-tuning cost against your compute, interface fit to your cameras and action space, and license fit to your intended use. For most new embodiments the survivor is OpenVLA with LoRA; reach for Octo if you need a diffusion head or a non-standard action space, and SmolVLA if your whole budget is one cheap arm and one consumer card.

Step 2 — Fix and verify the action convention (§16.2). Settle units, frame, delta-versus-absolute, and gripper encoding to match the base model's expectation. Then verify by open-loop replay: log one teleoperated episode, replay the logged actions on the robot, and confirm the motion reproduces. Do not proceed until the replay matches. This single check prevents the action-token mismatch failure (§16.4) that otherwise surfaces only on the robot, weeks later.

Step 3 — Collect a first dataset built to cover and to recover (§16.2). Grid the workspace and spread episodes across zones rather than the comfortable center. Vary object pose, lighting, distractors, and instruction phrasing to match what deployment will vary. Include recovery: when a grasp misses, drive back and re-approach on the same episode

rather than resetting. Aim for a few tens to low hundreds of episodes per task for a 7B base model, more for a small one. Record straight into the format the model loads (LeRobot, typically) so there is no reformatting step.

Step 4 — Fine-tune with LoRA (§16.3). Start at rank 16. Make sure the adapters reach the vision pathway or the encoder is otherwise trainable, because a frozen encoder reproduces OpenVLA's weak ~47% result. Keep the base weights frozen so you cannot catastrophically forget. Save the adapter, not a full model copy.

Step 5 — Screen in sim, then confirm on hardware (§16.4). If a correlated sim covers your task, run the fine-tune across several seeds and compute a confidence interval (§15.x) before spending robot time. Then run a small honest real evaluation: fixed success criterion written in advance, initial states from a stated distribution, every trial logged as a full episode with a failure mode from a fixed list, success rate reported with a Wilson interval (§15.5).

Step 6 — Diagnose failures with the three-set test (§16.4). Evaluate on exact training conditions, mildly shifted conditions, and an open-loop replay check. Fails replay: token mismatch, go back to Step 2. Passes training but fails shifted: over-specialization, widen data and lower rank or steps, do not train longer. Fails even training with frozen or blended motion: mode collapse, sharpen conditioning or move to a multimodal head.

Step 7 — Collect the next batch against the failures, and repeat (§16.4). The rollouts where the policy is weakest are the most valuable data you can record. Target the next collection session at exactly those conditions, fine-tune again, and turn the crank until the decision sentence from Step 0 is satisfied or you have evidence the base model cannot get there.

Why the order is the way it is

A recipe card is dangerous if it is followed without understanding, so here is what each ordering choice is defending against.

Step 0 comes first because the most common way a fine-tuning project wastes a month is not a technical failure at all; it is collecting a beautiful dataset for a task whose success criterion nobody pinned down, so the evaluation at the end cannot say whether it worked. The decision sentence is what makes every later number interpretable.

Steps 1 and 2 come before any data collection because both are cheap to get right early and ruinous to fix late. Choosing the base model after you have collected data risks discovering your data does not match its interface. Verifying the convention after you have collected two hundred episodes risks discovering all two hundred are in the wrong frame. The replay check in Step 2 takes ten minutes and saves the single most demoralizing failure in the chapter, the policy that trained perfectly and moves confidently to the wrong place.

Step 3 puts coverage and recovery ahead of volume because §16.2's whole argument is that fifty well-spread episodes with recovery behavior beat two hundred clean clustered ones. The card's instruction to grid the workspace is not fussiness; it is the direct countermeasure to the initial-condition variance (§15.5) that otherwise silently caps your success rate.

Step 4 defaults to LoRA at rank 16 with a trainable vision path because that is the configuration OpenVLA's own experiments found best-per-dollar, matching full-quality fine-tuning at roughly 8x less compute (§12.2). The rank-16 starting point is a starting point, not a law, which is why the exercise ablates it; but starting elsewhere without a reason is how people end up either overfitting a tiny dataset with too much rank or starving a hard task with too little.

Steps 5 and 6 separate screening from confirming and separate the three failure modes because the same on-robot symptom, "it does not work", has three different root causes that demand opposite fixes. The three-set test is the cheapest way to tell over-specialization (fix: less training, more data variety) from mode collapse (fix: sharper conditioning or a multimodal head) from token mismatch (fix: go back to Step 2). Apply the wrong fix and you make it worse, which is why the diagnosis is a step in its own right rather than an afterthought.

Step 7 makes the whole thing a loop rather than a line because a single pass almost never hits the target, and a project that treats fine-tuning as train-once-test-once concludes, wrongly, that the method failed. The loop is what converts a stalled fine-tune into a converging one, and it is the same evaluation-feeds-collection cycle §15.6 and §16.4 built.

What the card does not cover

The card assumes manipulation on a robot roughly shaped like an arm, because that is where the base models, the datasets, and the tooling are richest. It does not cover fine-tuning a whole-body or loco-manipulation policy, where a failure can mean the robot fell over and the real-time jitter budget of §14.4 shares a page with the success statistics of §15.5. It also stops at the point where the policy works in the lab; getting it to survive a real deployment, with monitoring, logging, and rollback, is a different discipline and the subject of Chapter 17. And it says nothing about certifying that a working policy is safe, because, as §17.5 will argue, nobody yet can.

With the card in hand you have the whole chapter as a procedure. §16.6 steps back from the procedure to the ideas, and names the four things you should now be able to do with a base model, a robot, and an afternoon.

16.6 Summary

Part 4 left you able to read the papers. Chapter 16 was the first chapter of the book about doing the work: taking a foundation VLA you did not train, and a robot the model has never seen, and closing the distance between them on hardware a normal lab owns. The chapter's argument runs against a common instinct. The instinct is that fine-tuning is mostly a training problem, solved by the right hyperparameters and enough epochs. The chapter's claim is that fine-tuning is mostly a data-and-procedure problem, and that the training itself, once you use LoRA, is the easy part. Most of what decides whether your policy works happens before the first gradient step, in which base model you picked and what your dataset contains, and after it, in whether you ran an honest loop or trained once and gave up.

The ideas worth carrying forward

The base model is chosen by constraints, not by scores. §16.1 put four filters in order: control rate on your deployable GPU, action-head match to your motion, fine-tuning cost against your compute, and interface and license fit. A model that tops a benchmark but cannot hit your control rate on your card is disqualified, not a fixer-upper, because §14.4's latency wall is structural. For most new embodiments the survivor is OpenVLA (arXiv:2406.09246) with LoRA, with Octo (arXiv:2405.12213) as the reach for a diffusion head or an odd action space and SmolVLA as the one built for a single cheap arm. The lesson that carries is that the leaderboard number is the last thing to consult, not the first.

Clean demonstrations make bad training data, and coverage beats volume. §16.2 was the chapter's most counterintuitive section. A dataset of flawless expert trajectories teaches a policy to be perfect from states it will never re-enter, and the moment it drifts, the compounding error of §6.3 takes over. The fix is to demonstrate recovery, driving back and re-approaching after a fumble on the same episode, which seeds the data with the off-distribution states and their corrections, the DAgger intuition (Ross et al., 2011) applied at collection time. Paired with that is coverage: fifty episodes spread across the workspace, lighting, and distractors beat two hundred clustered in the comfortable center, because the variance you fail to collect is the variance §15.5 said your success rate will die on.

LoRA makes a 7B fine-tune sane, and it works because the update is low-rank. §16.3 explained the method and the decision behind it. Freezing the big weight matrix and learning a small additive correction (Hu et al., 2021) cuts trainable parameters by orders of magnitude, which collapses the compute cost to a single GPU, and it preserves the base model's competence by construction, because you cannot overwrite what you never touch. OpenVLA's own table is the number to keep: LoRA matched full-quality fine-tuning at roughly 8x less compute, while last-layer-only collapsed to 30% and a frozen vision encoder capped out near 47%. That last figure is the trap, since it means a LoRA config that skips the visual pathway silently reproduces the weak result.

Sim buys volume, hardware buys truth, and neither certifies. §16.4 turned §15.6's screen-then-confirm procedure into a training loop: fine-tune, screen in sim across seeds with a confidence interval, confirm on a small honest real evaluation, read the failures, collect against them, repeat. Simulation earns its place as a cheap way to kill broken fine-tunes and as randomized data augmentation (the domain-randomization bet from §7.5), but a high sim number is evidence, not proof, and the loop never lets it stand as the final answer.

The same on-robot symptom has three root causes that demand opposite fixes. §16.4's diagnostic is the piece practitioners get most wrong. A fine-tune drives its loss to the floor and the robot still fails, and "it does not work" can mean

over-specialization (works in exact training conditions, fails when anything shifts; fix by widening data and training less, not more), mode collapse (frozen or blended motion because the policy averaged two valid solutions into one that satisfies neither; fix with sharper conditioning or a multimodal head), or action-token mismatch (confident motion to the wrong place because the action convention is wrong; fix by the open-loop replay check that should have happened before collection). The three-set test, exact conditions, shifted conditions, replay check, tells them apart, and applying the wrong fix makes things worse.

What you should be able to do now

Four things, in the order the recipe card uses them.

You should be able to *pick a base model for a specific robot and defend the choice.* Given a robot's control-rate requirement, its cameras and action space, your GPU, and your licensing needs, you can run §16.1's filter and say why OpenVLA, Octo, or SmolVLA is the right starting point, and why the highest-scoring checkpoint might not be.

You should be able to *collect a teleop dataset that a fine-tune can actually use.* You know to fix and replay-verify the action convention before recording anything, to spread episodes across the workspace and conditions rather than clustering them, to include recovery behavior instead of resetting after every fumble, and to record in the format your model loads so there is no reformatting step to introduce silent bugs.

You should be able to *fine-tune OpenVLA with LoRA and full fine-tuning and know when each is right.* You can configure a LoRA adapter that reaches the vision pathway, explain why it resists catastrophic forgetting, and say when the larger cost of full fine-tuning is justified (large varied data, evidence LoRA's capacity is the bottleneck) versus when action-head-only suffices (re-targeting an action space the model's perception already matches).

You should be able to *diagnose the three failure modes and write a recipe card for a new embodiment.* Handed a fine-tune that fails on the robot, you can run the three-set test, name whether it over-specialized, collapsed a mode, or has a token mismatch, and apply the matching fix. And you can compress the whole process into §16.5's ordered card, understanding why each step precedes the next.

Where the chapter has set up the rest of the book

Chapter 16 gets a policy working in the lab. Chapter 17 asks the harder question of whether it should be allowed to run outside one. The screen-then-confirm loop here becomes §17.3's A/B evaluation on hardware, the same interleaved, blinded protocol at higher stakes. The episode-logging discipline, every trial recorded as a full episode with a failure mode, becomes §17.4's production logging and rollback, watching a running robot instead of a test harness. And the three failure modes get a fourth, adversarial sibling in §17.5, where the question is not whether the policy fails on its own but whether someone can make it fail on purpose.

There is also a thread the chapter deliberately left open. §16.5's card assumed a robot shaped like an arm, and said nothing about fine-tuning a whole-body or loco-manipulation policy, where a failure can mean the robot fell over and the real-time budget of §14.4 collides with the success statistics of §15.5. That combined problem is one the field has barely standardized, and §18 returns to why the humanoid case makes it harder.

Chapter 16's contribution to the book's argument is to convert everything Part 4 explained into something you can do on Monday. A foundation VLA is not a finished product; it is a starting distribution, and the work of pointing it at your robot is mostly the unglamorous discipline of good data and an honest loop, with the clever low-rank training trick doing far less of the lifting than its reputation suggests. §16.x makes this concrete: you will fine-tune OpenVLA with LoRA on a small dataset and then ablate two knobs, the dataset size and the adapter rank, so the curves the chapter described in words become curves you plotted, and the diminishing returns of both become something you measured rather than took on trust.

16.x Hands-on exercise + chapter references

§16.3 asked you to accept two claims about the knobs on a LoRA fine-tune: that dataset size buys success with diminishing returns, and that the adapter rank sets how much you are allowed to change the base model, with too little starving a hard task and too much overfitting a small one. This exercise turns both claims into curves you plot.

The TOC states the target: fine-tune OpenVLA with LoRA on a small dataset, then ablate dataset size and adapter rank. You will run the same fine-tune several times, changing one knob at a time, and watch where each curve bends. If §16.3 was right, the dataset-size curve rises fast and then flattens, and the rank curve is flat across a wide middle band with a fall-off at both ends. If it was wrong, the curves are lines and the chapter oversold its knobs. They are not lines, and the bends are the lesson.

The same practical fork as §15.x applies. Running the OpenVLA LoRA fine-tune for real wants a GPU with room for the 7B checkpoint (a single A100, or a 24 GB consumer card with 4-bit quantization) and a couple of hours per training run. The curve-reading payoff, the part that teaches the diminishing returns, needs none of that. So 16.x.1 and the training runs are the full path; 16.x.2 onward computes and reads the ablation curves on a laptop CPU, from your own numbers if you ran the fine-tunes or from the recorded numbers below if you did not.

What you need

Full path: a CUDA GPU, the OpenVLA weights (arXiv:2406.09246) from Hugging Face, the PEFT library for LoRA (Hu et al., 2021), and a small teleop dataset in LeRobot format, either one you collected following §16.2 or a public LeRobot dataset of a single manipulation task. Analysis path: Python, NumPy, SciPy, and matplotlib.

Exercise 16.x.1 — Fine-tune once with LoRA

Wire up a single LoRA fine-tune and confirm it trains before you spend afternoons ablating it. The configuration below is the §16.5 recipe-card default: rank 16, adapters on the attention projections, and the vision encoder left trainable so you do not reproduce OpenVLA's weak frozen-encoder result.

```
from peft import LoraConfig, get_peft_model
from transformers import AutoModelForVision2Seq

base = AutoModelForVision2Seq.from_pretrained("openvla/openvla-7b")
cfg = LoraConfig(r=16, lora_alpha=16, lora_dropout=0.05,
                 target_modules=["q_proj", "v_proj", "k_proj", "o_proj"])
model = get_peft_model(base, cfg)
model.print_trainable_parameters()   # confirm it is a fraction of a percent

# ... standard supervised loop over your LeRobot dataset:
# batch -> model(pixels, instruction, action_labels) -> loss.backward() -> step
```

Two checks before trusting anything. The trainable-parameter fraction should print well under 1% of 7B, or your config is touching too much. And the training loss should fall; if it does not fall at all, suspect the action-convention mismatch from §16.2 before you suspect the learning rate, and run the open-loop replay check. A loss that falls is necessary, not sufficient, which is the whole point of §16.4.

Exercise 16.x.2 — Ablate dataset size

Hold the rank fixed at 16 and fine-tune four times on nested subsets of your data: 25, 50, 100, and 200 episodes, each subset a superset of the last so you are adding data, not swapping it. Evaluate each fine-tune with the honest protocol from §15.5, and put a Wilson interval (§15.x) on every success rate so the curve has error bars.

```
import numpy as np, matplotlib.pyplot as plt
from scipy.stats import norm

def wilson(k, n, conf=0.95):
    z = norm.ppf(1 - (1 - conf) / 2); p = k / n; d = 1 + z*z/n
    c = (p + z*z/(2*n)) / d
    h = (z/d) * np.sqrt(p*(1-p)/n + z*z/(4*n*n))
    return c - h, c + h
```

```
# recorded: (episodes, successes out of 25 eval trials)
size_runs = {25: 9, 50: 14, 100: 18, 200: 19}
xs = sorted(size_runs); ys = [size_runs[n]/25 for n in xs]
errs = [ (size_runs[n]/25 - wilson(size_runs[n],25)[0]) for n in xs ]
plt.errorbar(xs, ys, yerr=errs, marker="o"); plt.xscale("log")
plt.xlabel("training episodes"); plt.ylabel("success rate"); plt.show()
```

The shape to read is the bend. Going from 25 to 50 to 100 episodes buys big jumps, and 100 to 200 buys almost nothing that the error bars do not swallow. That flattening is §16.2's coverage-beats-volume claim in one plot: once the dataset covers the task's variation, more episodes of the same variation are near-duplicates, and the base model already knew how to see. The practical reading is that the 200-episode run was mostly wasted collection time, and those hours would have bought more if spent widening coverage instead of deepening it. Note also that the error bars at 25 trials are wide enough that the 100 and 200 points overlap, which is §15.x's warning: do not over-read a gap smaller than your interval.

Exercise 16.x.3 — Ablate adapter rank

Now hold the dataset fixed at your largest size and sweep the rank: 4, 8, 16, 32, 64. Same evaluation, same intervals.

```
rank_runs = {4: 15, 8: 18, 16: 19, 32: 19, 64: 17}   # successes / 25
xs = sorted(rank_runs); ys = [rank_runs[r]/25 for r in xs]
plt.plot(xs, ys, marker="s"); plt.xscale("log", base=2)
plt.xlabel("LoRA rank r"); plt.ylabel("success rate"); plt.show()
```

Two features. The curve climbs from rank 4, where the adapter is too small to fit the task, into a flat plateau across 8 through 32, which is the wide band §16.3 promised where the rank stops mattering. And at rank 64 it dips, because on a fixed small dataset the extra adapter capacity starts fitting noise instead of signal. That dip is over-specialization (§16.4) arriving through the rank knob rather than the training-length knob, and it is why the recipe card defaults to 16 rather than "as high as fits in memory." More capacity is not more skill when the data cannot fill it.

Exercise 16.x.4 — Reproduce over-specialization on purpose

Combine the two knobs to manufacture a failure and then diagnose it with the three-set test from §16.4. Take the smallest dataset, 25 episodes, and the highest rank, 64, and fine-tune. Then evaluate three ways: on the exact training starting conditions, on mildly shifted conditions (object moved a few centimeters, light changed), and with an open-loop replay check.

```
diagnosis = {"train_conditions": 22, "shifted_conditions": 6, "replay_ok": True}
# 22/25 in-distribution, 6/25 shifted, replay reproduces the demo motion
```

Read the pattern. High success on training conditions, a collapse on shifted ones, and a clean replay is the signature of over-specialization: the model memorized 25 episodes with more adapter capacity than they could justify, and it has no coverage to fall back on when the scene moves. The replay passing rules out a token mismatch, and the coordinated (not frozen or blended) motion rules out mode collapse. The fix the chapter prescribed is the counter-intuitive one: not more training, but more data variety and less rank, exactly the opposite of what the low training loss tempts you toward. Re-run at rank 16 on the 100-episode set and watch the shifted-condition number recover. That single before-and-after is the entire chapter compressed into one experiment.

Chapter 16 reading list

Cited across §16.1–§16.6, grouped by the job each reference does. Full entries for everything in the book live in Appendix E.2; this is the chapter-local subset.

The base models you choose between

- Kim, M. J., Pertsch, K., Karamcheti, S., et al. (2024). "OpenVLA: An Open-Source Vision-Language-Action Model." arXiv:2406.09246. The chapter's default base model and the source of its fine-tuning decision table

(last-layer 30%, frozen-encoder 47%, LoRA 68% at ~8x less compute) and the finding that the vision encoder must stay trainable.
- Ghosh, D., Walke, H., Pertsch, K., et al. (2024). "Octo: An Open-Source Generalist Robot Policy." arXiv:2405.12213. §16.1's diffusion-head, flexible-interface alternative, and §16.3's example of action-head-only re-targeting to a new action dimension.
- Shukor, M., et al. (2025). "SmolVLA: A Vision-Language-Action Model for Affordable and Efficient Robotics." arXiv:2506.01844. §16.1's small, LeRobot-native model built for a single cheap arm and one consumer GPU.
- Wen, J., et al. (2024). "TinyVLA: Towards Fast, Data-Efficient Vision-Language-Action Models." arXiv:2409.12514. §16.1's other small, data-efficient entrant in the deployment-oriented tier.

The fine-tuning method

- Hu, E., Shen, Y., Wallis, P., et al. (2021). "LoRA: Low-Rank Adaptation of Large Language Models." arXiv:2106.09685. §16.3's core method: freeze the big matrix, learn a low-rank additive correction, and get single-GPU fine-tuning that resists catastrophic forgetting by construction. The rank is the knob 16.x.3 ablates.

The data and evaluation discipline

- Ross, S., Gordon, G., Bagnell, D. (2011). "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning (DAgger)." AISTATS. §16.2 and §16.4's backbone: the argument that a policy must see off-distribution states and their corrections, which the chapter applies at collection time as "demonstrate recovery" and across rounds as "collect against the failures."
- Cadene, R., Alibert, S., Soare, A., et al. (2024). "LeRobot." github.com/huggingface/lerobot. §16.2's recording format, adopted so the dataset loads into training with no conversion step.
- Li, X., Hsu, K., Fu, J., et al. (2024). "Evaluating Real-World Robot Manipulation Policies in Simulation (SimplerEnv)." CoRL. §16.4's cheap screen, the sim built to correlate with real success.
- Tobin, J., et al. (2017). "Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World." IROS. §16.4's basis for using randomized sim episodes as data augmentation.

Chapter summary

Chapter 16 turned a shelf of trained VLAs into a procedure you can run on your own robot. You can now pick a base model by the constraints that actually decide the job, control rate, action head, compute, interface, and license, rather than by whichever number is largest, and defend why OpenVLA-with-LoRA, Octo, or SmolVLA fits a given robot. You can build a teleop dataset that earns its collection time, one that verifies its action convention by replay before recording, spreads across the workspace instead of clustering, and demonstrates recovery so the policy learns to get back on track. You can fine-tune with LoRA on a single GPU, say why it resists forgetting, and, after this exercise, read the dataset-size and rank curves you plotted to see coverage's diminishing returns and the rank sweet spot with your own eyes. And you can diagnose a fine-tune that fails on the robot, telling over-specialization from mode collapse from a token mismatch with the three-set test, and applying the fix that helps rather than the one the falling loss tempts you toward. Chapter 17 takes the working policy and asks the question this chapter set aside: not whether it works on a good day, but whether it is safe enough to run on a bad one.

Chapter 17 — Evaluation, safety, and deployment

17.1 Safety as a layer, not a property

Chapter 16 got a policy working in the lab. This chapter asks the harder question, the one a demo never has to answer: should it be allowed to run outside one, near people, for hours, when no engineer is watching? The instinct is to treat safety as something you train into the model, a property it acquires from good data and enough fine-tuning. That instinct is wrong, and getting it wrong is how robots hurt people. Safety is not a property a learned policy has. It is a layer you build around one, out of code whose worst-case behavior you can state and check, precisely because the policy's worst-case behavior you cannot.

Why you cannot train safety in

Go back to what a VLA actually is. §13.5 was blunt about it: a foundation action model is an imitation learner, a function that maps pixels and a sentence to an action, with no value function, no explicit model of consequences, and no guarantee about the motion it emits. A flow-matching head is a learned velocity field integrated by an ODE solver, and nothing in that machinery certifies that the resulting trajectory stays inside a workspace, keeps its force under a threshold, or refuses an instruction it should refuse. You can fine-tune it on a thousand careful demonstrations and it will still, on the input you did not anticipate, do something you did not train, because interpolating outside the data is exactly what these models do and exactly where their behavior is undefined.

This is why "we trained it to be safe" is not a safety argument. Training shifts the average. It does not bound the tail, and safety lives entirely in the tail. §1.5 drew the line that matters and it is worth repeating in full: two policies can post identical 95% success rates and be completely different products. One fails by dropping the object on the table, a productivity nuisance. The other fails by closing its gripper on a human hand, a recall notice. Their summary statistics are the same. The failure distribution is not, and no average-case metric, no benchmark from Chapter 15, distinguishes them. A safety layer is what you build because the model's own competence, however high, says nothing about what its 5% looks like.

What a layer is

The architecture is old and it is the same one classical robotics used before any of this was learned: the policy proposes, a separate mechanism disposes. The VLA suggests an action; a layer between the model's output and the motor command checks that action against constraints that were never learned from data, and clips, halts, or overrides it when it violates one. The layer is dumb on purpose. Its whole value is that you can read it, state what it will do in the worst case, and trust that statement, none of which is true of the network upstream.

The constraints that layer enforces are the boring, checkable ones. Force and torque limits: if the wrist sensor reads above a threshold, stop, because a rising force with no expected contact is the signature of the gripper meeting something it should not, a fixture, a wall, a hand. Workspace bounds: a geometric box, or a set of joint limits, that the end-effector may not leave, so a hallucinated action that would drive the arm off the table is rejected before it reaches an actuator. Velocity and acceleration ceilings: a cap on how fast the robot may move, because kinetic energy

is what turns a mistake into an injury, and a slow wrong motion is recoverable where a fast one is not. Collision-checked replanning: before a commanded motion executes, a fast geometric check against a model of the scene, with the motion refused or rerouted if it would intersect known geometry. None of these needs a network. All of them have worst-case behavior you can write on a whiteboard.

The hardware layer: compliance, not just checks

Software checks sit above the controller. There is a second safety layer below it, in how the robot is controlled at all, and §4.3 already introduced it: impedance control. A position-controlled arm commanded to a point will drive toward that point with whatever force it takes, which is exactly the behavior you do not want when the point is wrong and a person is in the way. An impedance-controlled arm behaves like a spring: commanded to a point, it applies a force proportional to how far it is from that point, so when it meets unexpected contact it deflects rather than pushing through. The Franka Panda exposes this mode by default, and it is what makes physical human-robot interaction survivable, because the arm yields to contact as a property of the controller rather than as a reaction the policy has to compute in time.

Compliance is a good example of the layered principle because it is a safety behavior that does not depend on the policy being right, or even on the monitor catching the error in time. A stiff position controller relies on nothing touching the arm unexpectedly; a compliant one degrades gracefully when something does. You want as much of your safety as possible to have that shape: behavior that holds even when the layer above it has already failed.

Defense in depth, and its honest limit

One layer is a single point of failure. The discipline, borrowed from every other safety-critical field, is defense in depth: several independent layers, each catching what the others miss, so that a hazard has to pass through every one of them to reach a person. The force limit catches the checks the workspace bound misses; the compliant controller catches the contact the force threshold was too slow to stop; a physical light curtain or an emergency stop catches what the software layers all missed. The layers are independent on purpose, because correlated failures are how defense in depth collapses: two checks that both rely on the same camera both fail when the camera does.

Here is the limit, stated plainly because the rest of the chapter depends on it. Layering reduces risk; it does not certify safety. Every layer you add is another chance to catch a failure, and none of them, together or apart, proves the system is safe, because the hazards you layered against are the ones you thought of, and the policy's undefined tail includes the ones you did not. §17.5 is where this book admits what that means: that nobody can currently certify a foundation-model-driven robot safe the way you can certify a hand-coded controller, and that the honest posture is risk reduction plus relentless measurement, not a safety guarantee. The layers in this chapter are how you reduce the risk. They are not how you sign off that there is none.

Safety is its own evaluation axis

There is a measurement consequence, and §15.6 already planted it. If safety lives in the failure distribution rather than the success rate, then an evaluation made only of tasks the policy can do never measures safety at all. You have to test the failures on purpose. Include a task the policy should refuse or physically cannot complete, and watch what it does: a policy that stalls when it cannot grasp is safe, and one that confidently drives the arm into the table attempting the impossible is not, and a success-only benchmark cannot tell them apart because neither scores a success. Log what the failures look like, not just that they happened, because "failed by stopping" and "failed by exceeding force" are the same row in a success table and opposite outcomes on a factory floor.

This reframes what the chapter is doing. Safety is not a section you bolt on after evaluation; it is an axis of evaluation, sitting alongside success rate and time-to-completion, measuring the shape of the 5% that success rate throws away. There is a research line trying to make some of this checking itself intelligent, using a vision-language model to watch the scene and flag constraint violations the geometric checks cannot express, which the Code-as-Monitor work (Zhou et al., 2025) is a good entry point to. That approach is powerful and it reintroduces the exact problem this section started with, because a learned monitor has the same undefined tail as the learned policy it watches. §17.2 takes the layer apart in detail: what a runtime monitor actually checks, where a VLM-as-monitor helps and

where it just moves the opacity around, and how the predicate drills you wrote back in §3.x become the kernel of a monitor running on a real robot.

17.2 Runtime monitors and shielding

§17.1 argued that safety is a layer of non-learned checks wrapped around the policy. This section builds the layer. The component that does the work has a name in the safe-control literature, a shield (Alshiekh et al., 2018): a piece of code that sits between the policy's output and the robot's actuators, watches every proposed action against a set of constraints, and overrides any action that would violate one before it executes. The policy proposes; the shield has veto power; the veto is the safety.

The monitor is a set of predicates

Strip a runtime monitor down and it is a list of predicates, boolean functions of the robot's state and the proposed action, each of which must be true for the action to pass. You already wrote a first sketch of these back in §3.x, the drills that checked whether a state satisfied some condition; here those same predicates become the kernel of a thing running on a real robot at control rate. The shield evaluates them every cycle, and when one fails it does not ask the policy to try again, it substitutes a safe action of its own.

```
def shield(state, proposed_action, limits):
    # each predicate returns True if the action is SAFE on that axis
    checks = {
        "force":     state.wrist_force < limits.max_force,
        "workspace": in_box(forward_kinematics(state, proposed_action), limits.box),
        "velocity":  norm(proposed_action.delta) < limits.max_step,
        "collision": not collides(planned_path(state, proposed_action), limits.scene),
    }
    if all(checks.values()):
        return proposed_action           # policy's action is allowed through
    log_violation(checks, state, proposed_action)
    return safe_fallback(state)           # e.g. hold position, or decelerate to stop
```

Two things about that sketch matter more than its details. The fallback is not "do nothing", it is a specific, defined safe action, usually decelerate-and-hold, because on a moving robot "do nothing" is ambiguous and "hold position under compliant control" is not. And the violation is logged, every time, with the state and action that triggered it, because those logs are the raw material §17.4 turns into alerting and the failure-mining loop from §15.6. A shield that silently clips is a shield you cannot debug.

What the monitor catches that testing did not

A monitor earns its place on exactly the failures your evaluation could not enumerate. Recall the three silent bugs from §2.4: a flipped image, a wrong prompt template, a wrong `unnorm_key` feeding the detokenizer the wrong per-embodiment statistics. Each produced confident, wrong actions and raised no exception. A monitor is the deployment-time answer to that whole class, because it does not care why the action is wrong, only that it violates a constraint. A detokenizer fed the wrong normalization statistics emits actions whose magnitudes are systematically off, and a velocity ceiling catches them at the first oversized step, long before you have diagnosed the `unnorm_key`. The monitor converts a silent failure into a loud, logged intervention, which is the whole point.

The same logic covers drift. §6.3's compounding error says a policy that wanders off its training distribution produces states it has never seen and actions that get worse from there. A monitor cannot fix the drift, but it can detect it: an action-statistics check that flags when the policy's outputs leave the distribution it produced in evaluation, or a stall detector that notices the robot is making no progress, gives you a trigger to stop and hand off rather than letting a confidently-diverging policy run to the point of damage. Stopping safely on a detected anomaly is almost always better than continuing, because the failures that show up in deployment, as §14.5 found, are the ones long runs expose and short demos hide.

The latency the monitor has to fit inside

A monitor runs inside the control loop, so it spends the budget §14.4 was strict about. A predicate shield is cheap by construction: a force comparison, a forward-kinematics call, a bounds check, a fast collision query, all of which run in microseconds and none of which involves a network. That cheapness is not incidental, it is why the safety layer can be non-learned and still keep up with a 200 Hz fast loop. The moment a check needs a neural forward pass, it can no longer live on the fast clock, which is the structural reason the intelligent monitors below run on the slow clock instead.

This is also where §14.7's problem lands. A dual-system policy's slow reasoner can propose an intent the fast controller should not execute, a reach that would exceed a force limit, a path through a person. The shield is the arbiter: it sits on the fast clock, it does not care that the intent came from a 7B model that reasoned carefully, and it vetoes the action on the same cheap predicates it applies to everything else. The reasoner's intelligence buys it no exemption from the force limit, which is exactly the property you want.

VLM-as-monitor: power and the opacity it reintroduces

Predicate shields have a ceiling: they can only check constraints you can write as cheap code. "Do not exceed 20 N" is easy; "do not put the knife down with the blade toward the person" is not a geometric box. A research line attacks this by using a vision-language model as the monitor, translating a free-form constraint into something checkable and watching the scene for violations. Code-as-Monitor (Zhou et al., 2025) is the clearest instance: it uses a VLM to compile constraints into small visual-programming checks, and it distinguishes reactive detection, catching a failure as it happens, from proactive detection, flagging that the current plan is heading toward one. That is genuinely more expressive than a predicate list, and it moves the opacity problem rather than solving it, because a learned monitor has the same undefined tail §17.1 warned about in the learned policy. You have added a second network that can also be wrong on the input nobody anticipated.

The honest way to use a VLM monitor is as an outer, slow-clock layer over an inner, fast, non-learned shield, never as a replacement for it. The cheap predicates guarantee the worst-case behavior you can state; the VLM catches the semantic violations the predicates cannot express; and when the two disagree, the cheap layer wins, because it is the one whose failure modes you understand. Defense in depth (§17.1) with a learned outer layer is fine. A learned layer alone is just the original problem with an extra step.

Authorization and learned failure signals

Two research directions are worth naming because they are where this subfield is moving. The first is runtime action authorization, surveyed in arXiv:2606.00090, which frames the shield as an authorization gate: every action must be authorized against an explicit policy before it reaches an actuator, and the survey organizes the design space of what does the authorizing and against what specification. It is the same shield idea given a security vocabulary, and the vocabulary helps, because it makes you name the authority and the policy rather than scattering `if` statements through a control loop.

The second is learning the monitor itself. Hide-and-Seek in Trajectories (arXiv:2605.30834) trains a model to detect failure signals in a policy's trajectory for runtime monitoring, and its central contribution is honest about the hard part: step-level failure annotation is noisy, because the moment a trajectory "went wrong" is rarely a clean single frame, and a monitor trained on noisy step labels inherits that noise. This is the same measurement-is-hard theme from Chapter 15 reappearing inside the safety layer, and it is why a learned failure detector, like a VLM monitor, belongs on top of a predicate shield rather than in place of one.

You now have the layer §17.1 promised, built from cheap predicates on the fast clock and expressive learned checks on the slow one. A monitor tells you when to intervene on a single robot in real time. The next question is a different one: across many runs, is version B of your policy actually better and safer than version A, or does it just look that way? §17.3 builds the A/B evaluation on hardware that answers it, and it runs into every statistical trap Chapter 15 warned about, now with a safety column added.

17.3 A/B evaluation on hardware

You fine-tuned a new version of your policy. Call it B, and the one already running A. B scores higher on your sim screen. The question this section answers is whether B is actually better on the real robot, or whether it just looks better, and the reason the question is hard is that a real robot is a noisy measuring instrument that drifts under you while you use it. The TOC's phrasing for the goal is exact: an A/B evaluation pipeline that does not lie to you. Most naive comparisons lie, and they lie in ways that look like results.

The comparison that lies

Here is the tempting, wrong protocol: run A for a day, collect its success rate, run B the next day, compare. It is wrong because everything §15.5 listed as a source of variance moves between the two days. The tape on the gripper wears, the lab lights shift, the motors run warmer in the afternoon, someone nudges a camera. Any of those can swing a success rate by more than the difference between two policies, so a day-over-day comparison measures the day at least as much as the policy. You will conclude B is better, or worse, and you will have measured the weather.

The fix is the one §15.5 already argued for in the single-policy case, now load-bearing: interleave. Alternate A and B trial by trial, so that whatever drifts hits both policies equally. If the light dims halfway through, it dims on A's trials and B's trials in the same proportion, and the comparison survives it. Batching is the enemy; interleaving is the whole trick. RoboArena (Atreya, Pertsch et al., 2025) scaled exactly this intuition across labs, using pairwise comparisons rather than absolute scores precisely because a relative, interleaved judgment is robust to the per-lab conditions an absolute number is not.

Pair the trials, and blind the judge

Interleaving buys more if you pair it. Where you can reset to the same initial state twice, run A and B from that same state back to back, and you have a paired trial: same object pose, same lighting, same everything, differing only in the policy. Paired comparisons are more powerful than unpaired ones because they cancel the per-trial variance that initial conditions inject, so you detect a real difference with fewer trials. On a robot where trials are expensive and §15.5 already showed intervals are wide at the sample sizes you can afford, that efficiency is worth the extra setup discipline of reproducing initial states.

Blinding matters as much as pairing. The person who resets the scene and scores the trial should not know whether A or B produced the rollout, because knowing leaks into the borderline calls, and manipulation is full of borderline calls: did the block settle in the target or wobble out, was that contact a success or a near-miss. §15.5 named the human in the loop as a source of bias; blinding is how you close it. Randomize the A/B order so it is not predictable, hide the identity from the scorer, and record which was which separately for analysis. If the scorer can guess the policy, the score is contaminated.

The statistic is the difference, not two intervals

Here is the trap §15.x set up and this section springs. You run the interleaved comparison, you compute A's success rate with its Wilson interval and B's with its own, you see the intervals overlap, and you are tempted to conclude "no difference." That conclusion is wrong, or at least unsupported, because two overlapping marginal intervals do not test a difference. The quantity you care about is the difference itself, and with paired trials it has its own, tighter test.

```
# paired A/B on the same initial states: each trial gives (a_success, b_success)
import numpy as np
from scipy.stats import binomtest

pairs = [(1, 1), (0, 1), (1, 0), (1, 1), (0, 1), ...]   # 1 = success
b_wins = sum(1 for a, b in pairs if b > a)   # B succeeded, A failed
a_wins = sum(1 for a, b in pairs if a > b)   # A succeeded, B failed
# McNemar: only the discordant pairs carry information about the difference
```

```
p = binomtest(b_wins, b_wins + a_wins, 0.5).pvalue
print(f"B>A on {b_wins} pairs, A>B on {a_wins}, discordant p={p:.3f}")
```

The point of that snippet is what it throws away. The pairs where A and B both succeed or both fail carry no information about which is better; only the discordant pairs, where one succeeded and the other did not, do. A McNemar-style test looks at exactly those, which is why a paired design is more sensitive than comparing two marginal rates. Report the difference with an interval, say the same thing in words, and never present two overlapping marginal intervals as if their overlap were the answer.

And decide N before you start. The cardinal sin of expensive-trial evaluation is peeking: running until the result looks significant and then stopping, which inflates your false-positive rate because you gave yourself many chances to cross the line by noise. Pre-register the trial count and the decision rule, run to it, and report what you find. If you must look early, use a sequential test built for it rather than a fixed test you peek at, because the fixed test's p-value means nothing once you have optionally stopped.

Do not drop the safety column

Everything so far compared success rates, and §17.1 already warned that success rate is the wrong sole axis. A/B evaluation inherits that warning with force, because the most dangerous outcome is a B that improves the mean and regresses the tail. Suppose B lifts success from 82% to 88% and, in doing so, changes its failure mode from "stops when uncertain" to "pushes harder and occasionally exceeds the force limit." On a success table B is the clear winner. On a factory floor B is the recall notice from §1.5, and shipping it because the mean improved is exactly the mistake this chapter exists to prevent.

So the comparison carries a safety column alongside the success column. Log every trial as a full episode with its failure mode, the §15.6 discipline, and count the safety-layer interventions from §17.2 per policy: how often did the shield have to veto B's action versus A's, and did B's failures move toward the benign end of the distribution or the dangerous one. A policy that wins on success and loses on shield-intervention rate has not been shown to be better; it has been shown to be a different risk profile, and the decision to ship it is a decision about that risk, not about the mean. Borrow LIBERO's axis idea (arXiv:2306.03310) here too: break the comparison out by the kinds of variation that matter, because B might beat A on the easy tasks and lose on the ones near a person.

From bench to production

The A/B protocol so far assumes a test station. Deployment adds two moves worth naming. Shadow mode runs B alongside A on the live robot without executing B's actions, computing what B would have done and comparing it to what A did, which gets you a large, cheap, real-world comparison with no risk because B never touches an actuator. Canary deployment runs B for real on a small fraction of the workload while A handles the rest, so a regression shows up on a slice you can roll back rather than the whole fleet. Both are the confirm step of §16.4's screen-then-confirm loop, extended past the lab into a running deployment, and both feed the logging and rollback machinery §17.4 builds next. An A/B result that says B is better on the bench is a hypothesis; shadow and canary are how you test it where it actually has to hold.

A comparison that interleaves, pairs, blinds, tests the difference rather than two intervals, fixes its sample size in advance, and refuses to drop the safety column is a comparison that does not lie to you. It is more bookkeeping than a leaderboard submission, and it is the price of knowing that the new policy you are about to trust near a person is actually the better one. §17.4 turns the logs this evaluation generates into the production machinery that watches the policy after you have shipped it: what to record, what to alert on, and what to roll back when the number moves the wrong way.

17.4 Logging, alerting, and rollback

A crash is the easy failure. It stops the robot, raises an exception, and points at its own cause. §2.4 named the failure that actually costs you: the bug that does not crash, the flipped image or wrong normalization that produces confident, wrong behavior and raises nothing. On a laptop that bug wasted an afternoon. In production, on a robot running unattended for a shift, the same class of silent degradation runs until something breaks or someone notices,

and "someone notices" is not a monitoring strategy. This section is about the machinery that catches the silent kind: what to log so a failure is reconstructable, what to alert on so you learn about it before the damage, and what to roll back so learning about it is survivable.

Log the whole episode, not the outcome

The unit of a production log is the same unit Chapter 15 used for a dataset: the episode. §15.6 told you to log every evaluation rollout as a full episode with a failure mode; production is that discipline turned always-on. For every run the robot makes, you record the stream of observations, the actions the policy proposed, the actions the shield actually let through, the proprioception, the timestamps, and the shield-intervention log from §17.2. Record it in the episode format from §15.1, the one LeRobot (Cadene et al., 2024) standardizes, and you get two things at once: a log you can replay to reconstruct exactly what happened, and, because it is already in training format, data you can fold straight back into the next fine-tune.

```
record = {
    "episode_id":   uuid(),
    "model_version": "openvla-ft-2026-08-03-a",   # WHICH policy produced this
    "metadata": {                                  # the stuff §2.4 said travels silently
        "unnorm_key": "my_robot", "prompt_template": "openvla-v1",
        "camera_calib_hash": "3f9a...",
    },
    "timesteps": [ ... ],          # obs, proposed_action, executed_action, force, dt
    "shield_events": [ ... ],      # every veto, with the predicate that fired
    "outcome": {"success": None, "failure_mode": None},   # filled in later
}
```

The two fields people leave out are the two that matter most in an incident. The model version, because when a robot misbehaves the first question is always "which policy was running," and a log that cannot answer it makes every incident an investigation from scratch. And the metadata that §2.4 warned travels implicitly, the `unnorm_key`, the prompt template, the camera calibration, because the silent bugs are precisely mismatches in that metadata, and a log that records it lets you diff a bad run against a good one and find the flipped bit. A log that stores only "success: false" has thrown away everything you need to know why.

Alert on leading indicators, not on crashes

Alerting on crashes is alerting too late. The signal you want is the drift that precedes the failure, and you already have the sensors for it from §17.2. The shield-intervention rate is the best single leading indicator: if the fraction of actions the shield has to veto is climbing, the policy is proposing more unsafe actions than it used to, which means it is drifting off distribution before any single failure is visible. Action-statistics drift is another, the §6.3 compounding-error signal, flagged when the policy's outputs move away from the distribution it produced in evaluation. Rising time-to-completion, the §15.5 metric, catches a policy that is still succeeding but working harder, which often precedes a policy that stops succeeding. Force readings creeping toward the limit, success rate sliding on a canary slice, all of these move before the robot actually fails.

The discipline is to pick thresholds that fire early enough to act and rarely enough to trust. An alert that fires constantly is an alert everyone ignores, which is worse than no alert because it launders inattention into process. So set thresholds off the evaluation baseline, alert on a sustained shift rather than a single noisy sample, and tier the response: a soft threshold that logs and notifies, a hard threshold that stops the robot and pages a human. The hard threshold is the one that has to fire faster than the damage, which means it lives close to the shield, on the fast clock, not in a dashboard someone checks hourly.

Rollback: versioning is the prerequisite

Rollback is only possible if you can name what you are rolling back to, so model versioning is the unglamorous prerequisite that makes the rest work. Every deployed policy is a named, stored checkpoint, and every log records which one ran, so that when B regresses you can revert to A without a rebuild. This is the same blue-green and

canary machinery §17.3 used to compare policies, now used to retreat: keep the previous version loaded and ready, and make switching back a single deliberate action rather than a redeploy under pressure.

Define the rollback triggers before you deploy, not during the incident. A rollback trigger is a written rule, "if the shield-intervention rate on the canary exceeds twice baseline for five minutes, revert to the previous version," decided in advance and applied mechanically, because deciding it while watching a robot misbehave is how a two-minute rollback becomes a twenty-minute argument. The trigger has to beat the damage: a rollback that takes an hour is no defense against a policy that hurts something in a minute, which is why the fast, hard stop from the alerting tier and the slower version-revert are two different mechanisms. The stop buys you the time; the revert fixes the cause.

The loop closes here

There is a payoff to logging every run as a replayable episode that goes beyond incident response, and it is the loop this book has been building toward since Chapter 15. The runs where your deployed policy is weakest, the ones that tripped the shield, drifted the statistics, or failed outright, are the most valuable training data you own, exactly as §15.6 and §16.4 argued. Because you logged them in training format, they flow straight back into the next fine-tune, so the failures your deployment surfaces become the data that fixes them. Evaluation, deployment, and data collection are not three activities; they are one loop, and production logging is the arc that closes it. A robot that runs, logs its failures, and feeds them back is a robot that gets better at the tasks it is worst at, which is the only kind of improvement that compounds.

This also reframes what a deployed VLA is. It is not a finished artifact you install and forget; it is a running process you watch, measure, and revise, more like a service in production than a shipped appliance. The monitoring, alerting, and rollback stack is the operational half of that, and it is why deployment is a discipline distinct from fine-tuning: §16 got the policy working, and this chapter keeps it working, and safe, once it leaves the lab. The techniques so far, layers, monitors, honest A/B, logging and rollback, all reduce risk and catch failures. None of them proves the policy safe. §17.5 confronts that directly, and names the residual risks, adversarial and otherwise, that no amount of this machinery currently lets anyone certify away.

17.5 What we still cannot certify

Everything in this chapter so far reduces risk. Layers catch violations, monitors flag drift, honest A/B keeps a regression off the floor, logging and rollback let you retreat. None of it certifies safety, and the gap between "we reduced the risk a lot" and "we certified it safe" is the subject of this section, because pretending the gap is closed is its own hazard. The honest position, stated once and defended below, is that nobody can currently certify a foundation-model-driven robot safe the way a hand-coded controller can be certified, and deploying one is a decision about managed risk, not a guarantee.

The guarantee you gave up

Chapter 4 is the contrast that makes this concrete. A symbolic planner (§4.1) operated on an explicit world model, and inside that model it could prove things: that the plan achieved the goal, that it respected the preconditions, that a new constraint added to the problem file would be honored. §4.4 named the trade you make leaving that world for a learned one, and this is where the bill comes due. A VLA offers no such proof. There is no world model to check the plan against, no precondition it verifiably respects, and no property you can establish by inspection, because the thing producing the action is an opaque forward pass over a 7B network. §9.1 made the same point from the world-model side: a system that emits actions from an inscrutable function gives you no handle to ask what future it predicted or whether that future was safe.

Formal verification, the tool that certifies safety-critical software in avionics and medical devices, does not scale here, and it is worth being precise about why rather than hand-waving. Verification proves a property holds over all inputs. A VLA's input is every image the camera might produce crossed with every instruction a person might phrase, an unbounded space, and the function mapping that space to actions is a deep network with hundreds of layers and no analyzable structure. There is no proof that the network never exceeds a force, never leaves a workspace, never closes on a hand, because constructing such a proof would mean bounding the behavior of the whole network over

the whole input space, which no current method can do. This is exactly why §17.1 put the guarantees in the non-learned layer: you certify the force limit because it is ten lines of code, and you cannot certify the policy because it is not.

The whole-body models sharpened this, as §14.5 through §14.7 traced. When the balance controller was hand-written C++, an engineer could read it and argue it would not step wrong; certification-by-inspection was at least possible for that layer. Helix-02's System 0 replaced it with a learned controller, and Gemini Robotics 2 folded balance into the manipulation policy itself, so the one part of the humanoid stack you used to be able to certify by reading is now as opaque as the rest. The trend is toward more capability and less analyzability at the same time, which is the uncomfortable direction for anyone who has to sign off that a robot is safe.

The adversarial surface

§16.4 gave you three failure modes a policy hits on its own. There is a fourth, and it is not accidental: the failures an attacker induces on purpose. §13.5 flagged the mechanism, that a continuous action head is a smooth, differentiable surface, and a smooth surface is one an optimizer can push on. The result is a research literature, growing fast through 2026, that this chapter has to represent honestly because it bears directly on what you can and cannot certify.

The canonical attack is the adversarial patch. TRAP (arXiv:2603.23117) showed that a printed pattern placed in the scene can hijack a VLA's chain-of-thought reasoning, steering the action the model produces, without touching the model or its inputs digitally, just by putting a crafted object in the camera's view. The property that makes this more than a lab curiosity is transferability. VLA-Hijack (arXiv:2605.28083) demonstrated patch attacks that transfer black-box across architectures, hijacking OpenVLA, UniVLA, and CronusVLA with a single pattern and no access to the target's weights, which means an attacker does not need your model to attack it, only one like it. Later work extended the threat to the partially observable case (arXiv:2606.03556), where the patch does not even have to stay in full view to work. For a deployed robot, a patch attack is a physical, reproducible way to make a policy misbehave that the success-rate evaluation of Chapter 15 would never surface, because the benchmark scene contains no adversary.

The fragility is not only adversarial. Lost in Fog (arXiv:2605.21446) showed that ordinary sensor perturbations, fog on a camera in a driving VLA, degrade the model's reasoning sharply, which means the same brittleness an attacker exploits also shows up under natural distribution shift that no one engineered. That is the more sobering result in some ways, because you can try to keep adversaries out of a workcell, and you cannot keep weather off a camera. The community has started to organize around measuring this: the CVPR 2026 AdvML workshop ran a challenge on adversarial attacks against driving VLAs (arXiv:2607.11560), and ForesightSafety-VLA (arXiv:2606.27079) is an early diagnostic benchmark that tries to score a policy's safety failures rather than its task successes. The Safety of VLA Models survey (arXiv:2604.23775) is the current map of the whole area, and the reason to cite a survey here rather than pile up primaries is that the field is moving fast enough that any single result is provisional; the survey is where you go to see the shape.

The honest posture

Put the two halves together and the conclusion is uncomfortable but clear. You cannot verify the policy, and the policy has an attack surface, both adversarial and natural, that your task-success evaluation does not touch. So certification in the strong sense, a proof that the robot is safe, is not available, and any vendor claiming it should be met with the question of what exactly was proven and over what input space. What is available is managed risk, and the management has three moves worth naming.

Keep the non-learned layers load-bearing, and never let a learned component be the last line of defense. The force limit, the workspace bound, the compliant controller, the physical emergency stop: these are what you can reason about, so they are what stands between a hijacked or fogged-out policy and a person. A VLM monitor (§17.2) can sit on top, but the thing that actually stops the arm has to be simple enough to certify.

Constrain the operating domain, because you often can certify the environment even when you cannot certify the policy. A caged workcell with no humans inside, a speed limit low enough that the worst-case kinetic energy is survivable, a task where the failure is a dropped part rather than a crushed hand: these are choices that shrink the space of hazards to one you can actually bound. The uncomfortable corollary is that the open-ended home robot,

operating near people with no cage and no speed cap, is precisely the deployment we are least able to certify, and the demos that look most impressive are often the ones running in the domain where the residual risk is highest.

Keep measuring, forever, because measurement is the only handle you have when proof is unavailable. The logging, alerting, and rollback of §17.4 are not just operations hygiene; on an uncertifiable system they are the substitute for a safety proof, the running evidence that the behavior has stayed inside the envelope you tested. A foundation-model robot is safe the way a bridge with sensors on it is safe: not because you proved it will never fail, but because you are watching it closely enough to catch the failure before it hurts anyone, and honest enough to admit that is what you are doing.

That admission is where the chapter has to end its argument, and it is the right note to hand to Chapter 18. The residual risks here, the ones no layer certifies away, are not a footnote to an otherwise solved deployment story; they are open problems, and the reasoning-plus-action and cross-embodiment threads §18 takes up are in part the field's attempt to build models whose behavior is analyzable enough to eventually close some of this gap. §17.6 gathers what you can now do, and it is a real and useful list, provided you hold it next to an honest account of what you still cannot.

17.6 Summary

Chapter 16 got a policy working in the lab. Chapter 17 was about the distance between working and deployable, which is larger than a demo ever lets on and is measured almost entirely in failures rather than successes. The chapter's spine is a single claim with an uncomfortable corollary. The claim: safety is not a property a learned policy has, it is a layer you build around one, out of code whose worst case you can state. The corollary, which §17.5 refused to soften: those layers reduce risk and none of them certifies the policy safe, so a deployed foundation-model robot is a managed risk watched closely, not a guarantee signed off. Everything between §17.1 and §17.4 is the machinery of managing that risk well; §17.5 is the honesty about what the machinery cannot do.

The ideas worth carrying forward

Safety is a layer, not a property, because you cannot bound the tail of a learned policy by training it. §17.1 is the chapter in one line. A VLA is an imitation learner with no value function and no guarantee about the motion it emits, so "we trained it to be safe" shifts the average and says nothing about the 5% where safety actually lives. Two policies with identical success rates can be a productivity tool and a recall notice, differing only in whether they fail by dropping the object or by closing on a hand. The response is a layer of non-learned checks, force and workspace and velocity limits, collision-checked replanning, compliant impedance control underneath, whose worst-case behavior you can write on a whiteboard precisely because the network's you cannot. Defense in depth stacks several such layers so a hazard has to pass all of them, and even then it reduces risk rather than proving its absence.

A runtime monitor is a shield of cheap predicates, and it earns its place on the failures your evaluation could not enumerate. §17.2 built the layer. A shield sits between the policy and the actuators, evaluates a list of predicates every cycle, the same predicate drills you first wrote in §3.x, and substitutes a defined safe action when one fails. It catches the silent bugs from §2.4 without diagnosing them, because a wrong `unnorm_key` produces oversized actions and a velocity ceiling stops them at the first step, and it converts the compounding drift of §6.3 into a logged, loud intervention. The predicates are cheap on purpose so they fit the §14.4 latency budget on the fast clock; a VLM-as-monitor like Code-as-Monitor (Zhou et al., 2025) adds semantic checks the predicates cannot express but reintroduces the opacity of the thing it watches, so it belongs on top of the cheap shield, never in place of it.

Hardware A/B that does not lie to you interleaves, pairs, blinds, and tests the difference rather than two intervals. §17.3 turned §15.5's single-policy protocol into a comparison. A day-over-day test measures the day; interleaving A and B trial by trial cancels the drift, pairing on shared initial states cancels the initial-condition variance, and blinding the scorer cancels the human bias. The statistic is the difference itself, and with paired trials only the discordant pairs carry information, which a McNemar-style test uses to beat comparing two overlapping marginal intervals, the §15.x trap. Fix the sample size in advance so peeking cannot inflate your false positives, and never drop the safety column, because a B that lifts the mean while moving its failures toward the dangerous end has been shown to be a worse risk, not a better policy.

Production is watching, not shipping, and the expensive failures are the ones that do not crash. §17.4 built the operational stack. Log every run as a full episode in the §15.1 format, including the model version and the metadata §2.4 said travels silently, so an incident is reconstructable and the log doubles as training data. Alert on leading indicators, a rising shield-intervention rate, action-statistics drift, creeping time-to-completion, rather than on crashes that fire too late, with thresholds tiered from a soft notify to a hard stop that beats the damage. Version every policy so rollback is possible, define the rollback triggers before deployment rather than during the incident, and recognize that the runs where the policy is weakest are the most valuable data you own, which is how deployment, evaluation, and data collection collapse into one loop.

Nobody can currently certify a foundation-model robot safe, and the honest posture is managed risk plus relentless measurement. §17.5 was the chapter's hard truth. A symbolic planner could prove properties inside its world model; a VLA offers no such proof, and formal verification does not scale to a 7B network over the unbounded space of images and instructions. The whole-body models made it worse by folding the one certifiable-by-reading layer, hand-coded balance, into the opaque policy. And the policy has an attack surface the success-rate benchmarks never touch: adversarial patches that hijack VLA reasoning and transfer black-box across architectures (TRAP, VLA-Hijack), fragility to ordinary sensor perturbation like fog, and the beginnings of diagnostic safety benchmarks trying to measure any of it. The three defensible moves are to keep the non-learned layers load-bearing, constrain the operating domain so you certify the environment even when you cannot certify the policy, and keep measuring forever, because measurement is the substitute for a proof you cannot construct.

What you should be able to do now

Four things, matching the chapter's objectives.

You should be able to *wrap a VLA in a runtime safety layer.* Given a policy and a robot, you can specify the force limits, workspace bounds, velocity ceilings, and collision-checked replanning that make a shield, implement it as cheap predicates that run inside the control loop, and define the safe fallback action the shield substitutes on a violation. You know why the layer must not be learned and why compliance belongs underneath it.

You should be able to *set up an A/B evaluation that does not lie to you.* You can interleave and pair trials, blind the scorer, test the difference with a paired statistic instead of two marginal intervals, pre-register the sample size to defeat peeking, and carry a safety column so a tail regression cannot hide behind a better mean. You know when to move the comparison into shadow mode or a canary slice.

You should be able to *decide what to log, alert on, and roll back.* You can design a production log that records the whole episode plus the model version and the silent-bug metadata, choose leading indicators to alert on with tiered thresholds, and write rollback triggers in advance that beat the damage. You understand why the logs close the failure-mining loop back into fine-tuning.

You should be able to *articulate the residual risks honestly.* You can explain why a learned policy cannot be formally verified, name the adversarial and natural-perturbation attack surfaces that task-success evaluation misses, and state plainly what managed risk buys and what it does not, which is the difference between an honest deployment and an oversold one.

Where the chapter has set up the rest of the book

Chapter 17 hands one clean thread to Chapter 18. The residual risks §17.5 named are not closed problems with known answers; they are open problems, and several of §18's threads are the field's attempts to reduce them. Reasoning-plus-action (§18.4) is in part a bet that a model which exposes its intermediate reasoning is more analyzable, and therefore closer to something you could eventually monitor or certify, than one that emits actions from a single opaque pass. Cross-embodiment generalization (§18.1) carries a safety cost the chapter only gestured at: a policy transferred to a body it was not validated on inherits none of that body's safety testing, so the very portability §14 celebrated is a certification liability §18 has to weigh.

The chapter also closes a loop the whole book has been drawing. The compounding-error problem named in §1.1, developed in §6.3, and deferred through Part 4 finally gets its deployment-time answer here: you cannot train it away, so you monitor for the drift and stop safely when it appears. The silent failures from §2.4 get their production

answer in §17.4's logging. The evaluation discipline from Chapter 15 becomes a safety instrument in §17.3. Chapter 17 is where the book's threads about what these models cannot do stop being cautions and become engineering.

What the chapter has not covered

Two gaps worth naming. The chapter treated safety mainly as physical safety, force, collision, workspace, and said little about the higher-level question of a policy that does the wrong task competently and safely, following a misinterpreted instruction to completion without ever tripping a force limit. That is an alignment problem more than a control problem, and the runtime-authorization line (§17.2) is only starting to formalize it. The chapter also stayed on single-robot deployment; it did not develop the safety of multi-robot systems like §14.6's leaderless coordination, where the hazard is emergent from the interaction rather than present in any one policy, and where the monitoring and rollback story is genuinely unsettled.

Chapter 17's contribution to the book's argument is to convert the recurring caution, that these models are more capable than they are analyzable, into a working deployment discipline and an honest boundary. You can build the layers, run the comparisons, watch the logs, and reduce the risk to something a serious operator can carry. You cannot prove the risk is gone, and the chapter's refusal to pretend otherwise is the most important thing it teaches. §17.x makes the layer concrete: you will wrap a force-and-workspace safety shield around a fine-tuned OpenVLA and induce three failures on purpose to watch it catch each one, followed by the chapter's full reading list.

17.x Hands-on exercise + chapter references

§17.1 asked you to accept that a safety layer has worst-case behavior you can state, where the policy it wraps does not. This exercise makes you build the layer and then attack it, so the claim stops being a sentence and becomes three logged vetoes you produced on purpose. The TOC states the target: add a force-and-workspace safety layer in front of a fine-tuned OpenVLA and demonstrate it catches three concrete failure modes you induce deliberately. The three are the ones §17.2 said a shield exists for: a force spike from unexpected contact, a workspace breach from an oversized action, and a velocity spike from a hallucinated jump. If §17.1 was right, the shield vetoes all three and logs why. If it was wrong, one slips through, and you have found a hole in your own safety layer, which is exactly the kind of thing you want to find on a laptop rather than on a floor.

The same honest fork as the last two chapters applies. Wrapping a live fine-tuned OpenVLA (§16.x) wants a GPU and the checkpoint. The shield itself is pure, cheap, non-learned code, which is the whole reason it can be certified, so it runs on a CPU in seconds and you can test it against scripted actions that stand in for whatever the policy might emit. Build and attack the shield on the CPU path; wrap the real policy if you have the hardware. The lesson is identical either way, because the shield does not care where the action came from.

What you need

Full path: a GPU, a fine-tuned OpenVLA checkpoint from §16, and a robot or simulator to execute on. Shield path: Python and NumPy. No policy required, because you will feed the shield the offending actions directly, which is both easier and more rigorous, since a scripted attack hits the exact predicate you mean to test.

Exercise 17.x.1 — Build the shield

Start from §17.2's shield and make it concrete. Four predicates, a defined safe fallback, and a violation log. Keep every check cheap, because a shield that needs a network cannot run on the fast clock.

```
import numpy as np

class Limits:
    max_force = 20.0            # newtons
    box = (np.array([0.2, -0.3, 0.0]), np.array([0.7, 0.3, 0.5]))  # xyz min, max
    max_step = 0.05             # meters per control step
```

```
def in_box(xyz, box):
    lo, hi = box
    return bool(np.all(xyz >= lo) and np.all(xyz <= hi))

def shield(state, action, L=Limits):
    next_xyz = state["ee_xyz"] + action["delta_xyz"]
    checks = {
        "force":     state["wrist_force"] < L.max_force,
        "workspace": in_box(next_xyz, L.box),
        "velocity":  float(np.linalg.norm(action["delta_xyz"])) < L.max_step,
    }
    if all(checks.values()):
        return action, None
    violated = [k for k, ok in checks.items() if not ok]
    return {"delta_xyz": np.zeros(3), "hold": True}, violated   # decelerate-and-hold
```

The fallback returns a hold, not a no-op, for the §17.2 reason: on a moving robot "hold position" is defined and "do nothing" is not. The second return value is the violation list, which is the log entry §17.4 alerts on. Confirm a benign action passes: a small in-box step with low force should return `(action, None)`.

Exercise 17.x.2 — Induce the force failure

The first attack is the contact spike. Simulate the robot meeting something it should not, a fixture, a wall, a hand, by handing the shield a state whose wrist force is over the limit, with an action that would keep pushing.

```
state = {"ee_xyz": np.array([0.4, 0.0, 0.2]), "wrist_force": 35.0}  # 35 N > 20 N
action = {"delta_xyz": np.array([0.02, 0.0, 0.0])}                  # keep pushing in
safe, violated = shield(state, action)
assert violated == ["force"] and safe["hold"]
print("force veto:", violated)
```

The point to sit with is that the shield never asked why the force was high. It did not need to know whether the policy hallucinated, the calibration drifted, or a person reached in. A rising force with a commanded push is refused on the reading alone, which is the property that makes the check certifiable: its behavior is stated over the sensor value, not over the network that produced the action.

Exercise 17.x.3 — Induce the workspace failure, the §2.4 way

The second attack reproduces a silent bug from §2.4 rather than an obvious one. Feed the shield an action with the wrong normalization scale, the kind a wrong `unnorm_key` produces, so the delta is an order of magnitude too large and would fling the end-effector out of the workspace.

```
state = {"ee_xyz": np.array([0.65, 0.0, 0.2]), "wrist_force": 2.0}
# a detokenizer with the wrong unnorm_key emits deltas at the wrong scale:
action = {"delta_xyz": np.array([0.5, 0.0, 0.0])}   # 0.5 m step, off the table
safe, violated = shield(state, action)
assert "workspace" in violated and "velocity" in violated
print("oversized-action veto:", violated)
```

This is the exercise's most important case, because it is the failure that has no exception and no telltale in the loss. The policy is confident, the numbers are the right shape, and only their magnitude is wrong. Two predicates catch it at once, the workspace bound and the velocity ceiling, which is defense in depth (§17.1) doing its job: the oversized action trips more than one independent check, so even if you had mis-set the box, the velocity limit still stops it. A success-rate benchmark would have scored this as a failed grasp and moved on; the shield turns it into a logged, attributable veto.

Exercise 17.x.4 — Induce the velocity spike, and read the log

The third attack is the hallucinated jump: an in-box, low-force action that is simply too large a single step, the discontinuity a policy sometimes emits when the scene confuses it.

```
state = {"ee_xyz": np.array([0.4, 0.0, 0.2]), "wrist_force": 1.0}
action = {"delta_xyz": np.array([0.0, 0.09, 0.0])}   # 9 cm in one step, inside the box
safe, violated = shield(state, action)
assert violated == ["velocity"]

# now run a stream and collect the log the way §17.4 wants it
log = []
for a in scripted_attack_stream():          # your three attacks, interleaved with benign steps
    _, v = shield(current_state(), a)
    if v: log.append({"t": now(), "violated": v})
print(f"{len(log)} vetoes; by predicate:",
      {k: sum(k in e['violated'] for e in log) for k in ['force','workspace','velocity']})
```

The per-predicate tally at the end is the seed of §17.4's alerting: a shield-intervention rate broken out by which check fired tells you not just that the policy is misbehaving but how, and a rising count in one column points at the cause. Force vetoes climbing means contact trouble; workspace and velocity vetoes climbing together means the scale bug from 17.x.3. The log is the difference between a shield that keeps you safe and a shield that also tells you why you needed it.

Going further

Two extensions worth the time if you have the hardware. Wrap the actual fine-tuned OpenVLA (arXiv:2406.09246) so the shield sees real policy outputs, and measure the shield-intervention rate on a normal run, which is your safety baseline for §17.4's alerting. And add a fourth, semantic check the predicates cannot express, "do not move toward the region tagged as a person", using a VLM-as-monitor on the slow clock in the Code-as-Monitor (Zhou et al., 2025) style, then confirm the cheap shield still runs underneath it and still wins on any disagreement. That is the §17.2 layering made real: expressive checks on top, certifiable checks at the bottom.

Chapter 17 reading list

Cited across §17.1–§17.6, grouped by the job each reference does. Full entries for everything in the book live in Appendix E.2; this is the chapter-local subset.

The safety layer and runtime monitoring

- Alshiekh, M., et al. (2018). "Safe Reinforcement Learning via Shielding." AAAI. §17.2's source for the shield: a correct-by-construction layer between policy and actuators that overrides unsafe actions. The idea predates VLAs and is what the predicate monitor implements.
- Zhou, Q., et al. (2025). "Code-as-Monitor: Constraint-Aware Visual Programming for Reactive and Proactive Robotic Failure Detection." CVPR 2025. §17.1 and §17.2's VLM-as-monitor: compiling free-form constraints into visual-program checks, and the reactive-versus-proactive distinction, with the caveat that a learned monitor reintroduces the opacity it watches.
- Runtime Action Authorization for VLA Models, A Survey (2026). arXiv:2606.00090. §17.2's framing of the shield as an authorization gate, and the design space of what authorizes an action against what specification.
- Hide-and-Seek in Trajectories: Discovering Failure Signals for VLA Runtime Monitoring (2026). arXiv:2605.30834. §17.2's learned failure detector and its honest problem: step-level failure annotation is noisy, so a monitor trained on it inherits the noise.

Evaluation on hardware

- Atreya, P., Pertsch, K., et al. (2025). "RoboArena: Distributed Real-World Evaluation of Generalist Robot Policies." §17.3's model for interleaved, pairwise, drift-robust comparison, scaled across labs.
- Liu, B., et al. (2023). "LIBERO: Benchmarking Knowledge Transfer for Lifelong Robot Learning." arXiv:2306.03310. §17.3's borrowed axis idea: break an A/B comparison out by the kinds of variation that matter, because B can win on easy tasks and lose near a person.

The residual risks and the attack surface

- Safety of Vision-Language-Action Models, A Survey (2026). arXiv:2604.23775. §17.5's map of the whole safety area, cited as a survey because the field moves fast enough that any single result is provisional.
- TRAP: Adversarial-Patch Hijacking of VLA Chain-of-Thought Reasoning (2026). arXiv:2603.23117. §17.5's canonical physical attack: a printed pattern that steers the action without any digital access.
- VLA-Hijack: Transferable Black-Box Patch Attacks Across VLA Architectures (2026). arXiv:2605.28083. §17.5's transferability result: one patch hijacks OpenVLA, UniVLA, and CronusVLA without the target's weights, so an attacker needs a model like yours, not yours.
- Partially-Observable Adversarial Patch Attacks on VLA Policies (2026). arXiv:2606.03556. §17.5's extension where the patch need not stay in full view to work.
- Lost in Fog: Sensor Perturbations Expose Reasoning Fragility in Driving VLAs (2026). arXiv:2605.21446. §17.5's non-adversarial fragility: ordinary fog degrades reasoning, the brittleness you cannot keep out of a workcell.
- ForesightSafety-VLA: A Diagnostic Safety Benchmark (2026). arXiv:2606.27079, and the CVPR 2026 AdvML adversarial-VLA challenge (arXiv:2607.11560). §17.5's early attempts to measure safety failures rather than task successes.

The backward thread

- Featherstone, R. (2008). *Rigid Body Dynamics Algorithms*. Springer. §17.1's impedance-control layer, the compliant contact behavior from §4.3 reused as a hardware safety property that holds even when the checks above it fail. The certification the chapter cannot give the policy is the certification §4.1's symbolic planner could give its plan, which is the trade §4.4 named.

Chapter summary

Chapter 17 converted the book's recurring caution, that these models are more capable than they are analyzable, into a deployment discipline and an honest boundary. You can now wrap a VLA in a runtime safety layer of cheap, certifiable predicates with a defined safe fallback, and after this exercise you have watched that layer catch a force spike, an oversized-action scale bug, and a hallucinated jump that you induced on purpose. You can run a hardware A/B comparison that interleaves, pairs, blinds, tests the difference rather than two intervals, and refuses to drop the safety column, so the new policy you ship is actually the better and safer one rather than the luckier one. You can build the production logging, leading-indicator alerting, and pre-defined rollback that catch the silent failures a crash-only monitor misses, and close the loop that turns your worst rollouts into your next training data. And you can state, without softening it, what none of this certifies: that a learned policy cannot be formally verified, that it carries an adversarial and natural-perturbation attack surface your success benchmarks never test, and that deploying one is managed risk watched closely, not a guarantee. Chapter 18 takes the open problems this honesty exposes, cross-embodiment transfer, long-horizon and dexterous tasks, video pretraining, and reasoning joined to action, and asks where the field goes from here.

Chapter 18 — Open problems and what comes next

18.1 Generalization across embodiments

Every chapter until now has treated the robot as fixed. You picked a base model in §16.1, collected teleop data on your arm in §16.2, fine-tuned, and evaluated on that same arm. The policy learned your gripper, your camera placement, your seven joints. Move it to a different robot and it breaks, because the action it emits at step one, a seven-number vector of joint deltas, means something entirely different to a machine with a parallel jaw and a different reach. This is the embodiment problem, and it is the first of the open questions this chapter takes seriously, because it is the one standing directly between the field and the thing the word "foundation" was supposed to promise: one model, many robots, the way one language model serves every writing task.

What makes it hard is that "embodiment" is not one variable. It is the number of joints and how they are arranged, the gripper or hand and how many fingers it has, where the cameras sit and what they see, the control rate the hardware runs at, and the space the actions live in. A policy trained on a Franka Panda has memorized a particular value of every one of these. Two robots that a person would call similar, two 7-DOF arms with parallel grippers, still differ in link lengths, joint limits, and camera extrinsics, and a policy that never saw the second one has no reason to control it correctly. The failure is not subtle. It is the arm driving to the wrong place on move one.

The bet, and where it stands

§12.4 introduced the wager that a lot of the field placed on this problem: Open X-Embodiment (arXiv:2310.08864). Pool demonstrations from many robots into one dataset, train a single policy on all of it, and hope the shared structure of manipulation, reach toward the thing, close on the thing, move it, transfers across the differences in hardware. The RT-X models trained this way did show positive transfer. A policy trained on the pooled data outperformed policies trained on any single robot's data alone, which is the result that made the whole approach credible. Octo (arXiv:2405.12213) built on the same dataset with a design meant to absorb heterogeneity directly: a transformer with swappable input and output heads, so a new robot with a different observation and action layout could be attached without retraining the trunk.

So the bet paid off, partially, and the partiality is the interesting part. Transfer works well inside a rough equivalence class. Table-top arms doing pick-and-place share enough that pooling their data helps, and a generalist trained across a dozen of them beats a specialist on the ones it has seen and reaches a usable starting point on ones it has not. Push outside that class and the story gets worse fast. A bimanual system, a mobile manipulator, a quadruped, a humanoid: these do not share an action space with a table-top arm in any way a naive concatenation of joint vectors can express, and simply adding their data to the pool can hurt the arm policies rather than help them. §15.2 made the same point from the data side, that Open X-Embodiment has no single branded successor and that growth now comes from large teleop fleets feeding individual foundation models rather than from one dataset serving everything. The dream of a universal pool ran into the fact that a humanoid and a UR5 are not doing the same math.

What action space are we even generalizing over?

The technical core of the problem is a representation question, and it is worth stating plainly because most of the current research bets are really bets on the answer. If you want one policy to drive many robots, the policy has to emit something that means the same thing to all of them. What is that something?

Make the mismatch concrete. Suppose your policy outputs a 7-vector: six joint velocities plus a gripper command. On the Franka it was trained on, element four drives the elbow, and a value of 0.3 there produces a specific, learned amount of elbow flexion. Feed that identical vector to a Kinova with a different joint ordering and different gear ratios, and element four now drives a wrist rotation at a different scale, so the arm does not do a slightly wrong version of the task, it does something unrelated to it. The numbers transfer; the meaning does not. That single example is the whole problem in miniature, and it is why the choice of what the policy emits matters more than almost any other design decision when many robots are the target.

The candidates each fail somewhere. End-effector pose in Cartesian space, "move the gripper to this position and orientation," is appealingly robot-agnostic; a pose is a pose regardless of the arm underneath it. But it pushes the whole problem down into the inverse kinematics and controller, which is exactly the §4.2–§4.3 machinery, and it says nothing at all for a robot whose "end effector" is five fingers or two arms. Joint-space targets are precise and directly executable, and they are the least transferable thing imaginable, since joint 3 on one robot is not joint 3 on another. A learned universal action space is the ambitious middle: UniAct (arXiv:2501.10105) proposes a shared vocabulary of "universal actions," a codebook of behaviors that is common across embodiments, with a small per-robot decoder translating a universal action into that robot's specific commands. This is the cleanest statement of the idea that there is a level of abstraction, above joints and below language, where all manipulators are doing the same thing. Whether that level really exists, or whether it is an artifact that works on the benchmarks and frays on hardware, is genuinely open. It is a good problem for a student to pick up in §18.5, because you can probe it with one robot and a released checkpoint.

The boundary that is dissolving: whole-body control

There is a shift happening right now that scrambles a distinction this book drew carefully back in Chapter 4, and it belongs here because it is the most concrete form the embodiment question is taking. In Chapter 4, locomotion and manipulation lived on opposite sides of a line. Getting a legged robot to walk was a model-based control problem, a matter of dynamics, contact schedules, and a hand-tuned controller you could reason about; manipulation was where the learned policies went. A humanoid, under that division of labor, ran a classical whole-body controller for its legs and torso and handed off to a VLA for its arms and hands. Two systems, two design philosophies, a clean interface between them.

That interface is being deleted. Figure's Helix-02, introduced in §14.2, ships what the company calls "System 0," a neural whole-body controller that replaces the hand-coded locomotion stack, folding balance and stepping into the same learned system that drives the hands. Google DeepMind's Gemini Robotics 2, announced at the end of July 2026 and covered in §14.6, went further and demonstrated the first VLA to drive an entire humanoid, legs, torso, arms, and multi-fingered hands, under one unified policy, on Apptronik's Apollo 2. Loco-manipulation, walking-while-carrying, reaching-while-balancing, stops being a handoff between a classical controller and a learned one and becomes a single network's output. WholebodyVLA (OpenDriveLab, ICLR 2026) is the open-research version of the same direction.

Read this against §4.4, which argued that classical methods remain load-bearing in modern robots precisely in locomotion and low-level control. That argument was correct when it was written and it is now the exact frontier being contested. The whole-body policies are betting that the model-based locomotion controller, one of the last places where hand-derived dynamics clearly beat learning, can be absorbed into the same foundation model that does everything else. If that bet holds, the embodiment problem and the locomotion problem collapse into each other: generalizing across embodiments stops meaning "across different arms" and starts meaning "across different bodies," legged and wheeled and bimanual alike, with the policy responsible for figuring out how each one moves.

One policy, twenty robots

The most striking recent data point on cross-embodiment scale is LingBot-VLA 2.0, released in July 2026: a 6-billion-parameter open-source VLA reported to run one policy, unmodified, across twenty distinct robot morphologies from seventeen manufacturers, from Unitree and AgiBot to Franka and Fourier. Take the claim with the usual caution owed to a fresh release, and it still marks how far the target has moved. Open X-Embodiment pooled data from twenty-two embodiments to train a policy you then specialized. The newer systems aim for one checkpoint that a person unpacks and points at whatever robot is in the room, which is a categorically harder ask and the one the field has decided to make.

None of this is solved. Transfer across near-identical arms is routine; transfer across genuinely different bodies is a research result, not a product guarantee, and the right action representation to make it reliable is unsettled. What has changed since the early Open X-Embodiment days is the ambition and the evidence that the ambition is not hopeless: whole-body neural control is real hardware now, universal action spaces have released checkpoints you can poke at, and the classical locomotion boundary from Chapter 4 is under active demolition. The next open problem, §18.2, holds even the best of these single-embodiment-generalizing policies to a harder standard: not whether one model can drive many robots, but whether any of them can yet do the long-horizon, dexterous tasks that a human hand makes look trivial.

18.2 Long-horizon and dexterous tasks

Watch a person make a sandwich and you watch two things a robot cannot yet do well. The task runs for two minutes across a dozen sub-goals, open the jar, spread the filling, close it again, and stack, and a single dropped step ruins the result. And each of those sub-goals asks the hand to do something contact-rich and precise: unscrew a lid with three fingers, slide a knife under a slice, feel when the grip is firm enough and no firmer. Long horizon and high dexterity, the two capabilities a human hand and a human plan make look free, are the second open problem, and unlike embodiment transfer they are not close to solved. §18.1 asked whether one policy can drive many robots. This section asks the harder question underneath it: whether any policy, on any single robot, can yet do the tasks that actually fill a kitchen or a workcell.

Two problems that get bundled and should not

Long-horizon and dexterous are different failures, and treating them as one word obscures where the field is stuck. Long-horizon difficulty is a problem of time and error accumulation. Dexterity is a problem of contact and control bandwidth. A robot can be good at one and hopeless at the other, and most current systems are exactly that.

The long-horizon problem is the compounding-error problem from §6.3, scaled up until it dominates. A behavior-cloned policy makes a small error, the small error moves the state slightly off the training distribution, the next action is drawn from a worse-conditioned part of the model, and the errors feed on themselves. Over a five-second grasp this is a nuisance you can often absorb. Over a two-minute, twelve-step task it is fatal, because the probability of executing all twelve steps correctly is the product of twelve per-step success rates, and even 95% per step compounds to roughly 54% over twelve. The arithmetic is unforgiving and it explains why demos are short. A one-minute autonomous task is a genuine research result; a ten-minute one is rare enough to make headlines.

The dexterity problem is different in kind. A parallel-jaw gripper has one degree of freedom and forgives a lot: close it near an object and the two plates find the grasp. A five-fingered hand has twenty or more degrees of freedom, and using them means controlling contact, when a fingertip touches, how hard, in what sequence, with feedback fast enough to catch a slip before the object falls. This is where the model-free imitation approach strains. The training signal, teleoperated demonstrations, is thin for hands because teleoperating twenty joints is hard for the human operator too, and the control bandwidth needed to react to a slip is higher than the 5-to-20 Hz many VLAs run at from §14.4. Contact dynamics are stiff and near-discontinuous, exactly the regime the smooth flow-matching heads from §10.4 were designed to handle but have not fully conquered.

Where the horizon problem is being attacked

The clearest line of work accepts that a single flat policy will not hold a long task together and reintroduces structure. Long-VLA (arXiv:2508.19958) is a representative attempt: rather than asking one policy to emit a two-minute trajectory in one undifferentiated stream, it gives the model a way to represent and sequence sub-tasks, so the long task is decomposed into segments the policy can actually execute and a higher level tracks which segment it is in. LiLo-VLA (arXiv:2602.21531) works the same territory. The intuition should feel familiar, because it is the §4.1 symbolic planner's intuition wearing new clothes: hard long tasks are easier when you decompose them into sub-goals and solve the sub-goals. The classical planners did this with hand-written operators; the new work tries to learn the decomposition and keep it differentiable, which is the part that is genuinely open.

A second attack comes from a direction Chapter 5 would recognize. Physical Intelligence's *π0.6, documented in their RECAP note (pistar06.pdf), takes a π-family policy and improves it with reinforcement learning from the robot's own experience rather than from more human demonstrations alone. This matters for long horizons specifically, because RL optimizes the whole return, the eventual completion of the task, rather than imitating one step at a time, and a policy trained against the full-task reward has pressure to recover from the off-distribution states that compounding error drives it into. That is exactly the failure §6.3 said imitation cannot fix on its own and §6.5 said RL was the tool for. The* π0.6 result is the field circling back to reward-based learning to buy the robustness that pure cloning cannot, now on top of a foundation model rather than from scratch.

Where the dexterity problem is being attacked

Dexterity has attracted dedicated foundation models, which is itself the news. For most of this book the story was one generalist policy doing many tasks; dexterity is hard enough that two well-funded efforts have decided it deserves a purpose-built model. Genesis AI's GENE and RLWRLD's RLDX-1, both surfacing through 2026, are foundation models aimed squarely at dexterous, contact-rich manipulation rather than at the pick-and-place tasks that fill Open X-Embodiment. The bet behind them is that dexterity is not a task you get for free by scaling a generalist, that the data, the action representation, and the control rate a five-fingered hand needs are different enough to warrant specializing the whole stack.

Whether that bet is right is open, and it is a good example of a live disagreement a reader can track. The generalist camp holds that dexterity emerges from scale, that a large enough policy trained on enough hand data will pick up contact control the way RT-2 picked up semantics (§12.5). The specialist camp, GENE and RLDX-1 among them, holds that contact-rich control has structure, tactile feedback loops, force modulation, finger gaiting, that you should build in rather than wait for, and that a generalist spread across a hundred embodiments will never allocate enough capacity to any one hand. The honest status: nobody has shown human-level dexterity from either approach, and the gap to a human hand is still large. A person re-grasps a slipping object without thinking; the best robot hands do it slowly, in the lab, on objects they have seen.

The measurement problem hiding underneath

There is a reason progress here is hard to read, and it connects back to Chapter 15. Long-horizon and dexterous tasks are exactly the ones the standard benchmarks measure worst. A benchmark of short table-top tasks (§15.4) says nothing about a two-minute task, and a success-rate number on a grasp says nothing about the quality of the manipulation, whether the object was handled gently, whether the grip would have survived a perturbation. The field is partly stuck because it is optimizing what it can measure, and the measurements reward short and forgiving over long and dexterous. §15.5's insistence on time-to-completion and on variance, not just binary success, is a partial fix, and it is no accident that the labs pushing hardest on dexterity are also the ones building their own evaluations rather than trusting LIBERO.

Consider one concrete illustration of how badly the two problems interact. A folding-laundry task, a favorite in demo reels, is both long and dexterous: it runs for minutes, and each fold asks the hand to manage a deformable object whose state the robot cannot fully observe. A rigid box has a pose you can estimate; a shirt has a configuration space that is effectively infinite, and the gripper has to feel its way through contact it cannot predict from vision alone. When a laundry demo works, it usually works on one shirt, folded one way, from one starting configuration, and the moment you hand it a different garment the success rate falls off a cliff. That cliff is the honest measure of

where the field is: not the polished clip, but how fast performance degrades when the object or the sequence changes even slightly. A reader who wants to feel the frontier directly can reproduce the cliff on a tabletop with two objects of different stiffness and a three-step task.

This is where a reader with one robot can actually contribute, which §18.5 develops. You do not need a fleet to study long-horizon failure; you need one arm, one genuinely multi-step task, and the discipline to log where in the sequence the policy breaks and why. The compounding-error curve is reproducible on a tabletop. So is the observation that a policy's dexterity collapses the moment you swap the training object for a slightly slipperier one, which is the generalization-to-contact question the specialist models are betting their existence on.

Long horizons and dexterity are the capabilities that separate a compelling demo from a robot that earns its keep, and they remain the clearest statement of how far there is to go. The horizon problem is being met with learned decomposition and with reinforcement learning bolted onto foundation policies; the dexterity problem is being met with purpose-built models and a still-unsettled argument about whether it needed them. Neither is close to a human hand executing a two-minute plan without a thought. The next section turns to a resource both problems might eventually draw on: the enormous quantity of video showing humans doing exactly these tasks, and the models beginning to learn from it.

18.3 Video-pretrained action models

The bottleneck this whole book has circled is data. §11.5 made the argument in full: robot data is expensive, a teleoperated episode costs a human operator's time on a physical machine, and the largest robot datasets are tiny next to the corpora that trained language and vision models. Open X-Embodiment pooled roughly a million episodes and that was a landmark; a video model trains on billions of clips. So the obvious question, asked for years and only recently answered with something concrete, is whether the ocean of video showing humans doing things, cooking, assembling, folding, could pretrain an action model the way web text pretrained GPT. The tantalizing part is that video contains exactly the long-horizon dexterous behavior §18.2 said robots cannot do. The frustrating part is that video has no action labels.

Why video is hard to learn actions from

A robot demonstration is a sequence of observations paired with the actions that produced them: image, then the joint command the teleoperator sent, then the next image. Behavior cloning (§6.1) needs that pairing, because it learns a map from observation to action, and the action is the training target. A YouTube video of someone slicing an onion has the observations, every frame, and none of the actions. You see the knife move; you do not see the torques the wrist applied, and you could not read them off even if you tried, because the human's joints are not the robot's joints. This is the label gap, and it is why decades of robot learning could not simply ingest video the way NLP ingested text.

There are two ways around the gap, and the field has tried both. The first is to use video only to learn representations, not actions: train a visual encoder on video so it produces features that make downstream action learning cheaper, then attach a small action head trained on the scarce robot data. R3M (Nair et al., 2022) was an early, clean version of this, and it worked, in the modest sense that a video-pretrained encoder beat one trained from scratch on the robot data alone. The limitation is that it only ever moves the visual front end; the action policy still needs robot demonstrations, so the data bottleneck is loosened, not broken. The second way is more ambitious and is where the recent excitement lives: learn a world model from video, a model of how the world evolves, and then use that model as the substrate an action policy plugs into, so that most of what the system knows about physics and object behavior came from video and only the last mile came from robot data.

The concrete result: V-JEPA 2 and V-JEPA 2-AC

For most of this book's writing, video pretraining for control was a promising direction with thin evidence. That changed with V-JEPA 2 (arXiv:2506.09985), and it is worth being precise about what it showed, because the result is the strongest existing argument that the video bet pays off. §9.4 introduced V-JEPA as a video-prediction world model that learns by predicting in a latent space rather than reconstructing pixels, which §9.2's latent-dynamics

discussion motivated: predicting whether the representation of the next state is right is a better learning signal than predicting every pixel, most of which are irrelevant. V-JEPA 2 scaled that recipe on internet video, over a million hours, learning a world model with no robot data and no action labels at all.

The action-conditioned variant, V-JEPA 2-AC, is where control enters. Meta post-trained the video world model into a policy using a small amount of robot interaction data, under 62 hours of unlabeled robot video, and the resulting system did zero-shot pick-and-place on a real robot: reaching and grasping objects it was not explicitly trained to manipulate, in a new lab, without task-specific demonstration. The number that made people look twice was the comparison: V-JEPA 2-AC reportedly reached around 80% success on a held-out pick-and-place task where Octo (arXiv:2405.12213), a strong conventional generalist trained the usual way, managed roughly 15%. Take the exact figures as provisional, the way §18.1 said to take LingBot's morphology count, and the shape of the claim is still the important thing. A world model built almost entirely from video, with a thin layer of robot data on top, beat a policy built the conventional way by a wide margin on a task neither was trained on. That is the video bet working, at least once, on real hardware.

What the result does and does not prove

Be careful about what to conclude. V-JEPA 2-AC did zero-shot pick-and-place, which is the short, forgiving end of the task spectrum, precisely the tasks §18.2 said the benchmarks over-reward. It did not do a two-minute dexterous task, and there is no result yet showing video pretraining closes the long-horizon or fine-manipulation gaps. What the result proves is narrower and still important: that a world model learned from action-free video transfers to control, that the physics and object-permanence and motion structure absorbed from watching are usable by a policy, and that the amount of robot data needed to convert the world model into a controller is small. That last point is the one that matters for the data bottleneck. If 62 hours of robot video can turn a video world model into a working policy, the economics of building a new robot's policy change, because 62 hours is a weekend of teleoperation, not the fleet-years Open X-Embodiment represented.

The open questions stack up from there. Does the advantage hold on contact-rich tasks, where the video shows a hand doing something the robot's gripper cannot copy? Human video has a worse embodiment gap than robot video, since a human hand is not a robot hand, and the label problem returns in a new form: even if you learn what happens from video, mapping it onto the robot's own actions is unsolved for anything beyond gross reaching. And there is the architecture argument from §9.5, still unresolved, about whether the world model should be a separate module a policy plans against or whether the policy should absorb it, which V-JEPA 2's separation-then-post-training approach answers one way and the end-to-end VLAs answer the other.

It helps to be concrete about why the pixel-versus-latent choice matters, because it is the design decision that made V-JEPA 2 work where earlier video models stalled. A pixel-reconstruction world model spends most of its capacity predicting things a controller does not care about: the exact texture of a wall, the flicker of a shadow, the leaves moving outside a window. None of that helps a robot decide how to grasp, and a model graded on reconstructing it wastes its learning on detail. Predicting in latent space, the §9.2 idea, changes the grading: the model only has to get the representation of the next state right, so it is free to throw away the pixels that do not matter and keep the structure that does, object identity, rough position, the fact that a pushed cup slides rather than teleports. That is why a latent video model transfers to control and a pixel one mostly does not, and it is the single most important thing to carry out of this section if you are choosing an architecture rather than just tracking the field.

Genie and the interactive-video direction

V-JEPA 2 is the prediction-and-planning branch. There is a second branch worth naming, because it attacks the label gap from the opposite side. Genie (Bruce et al., ICML 2024), introduced in §9.4, learns a generative interactive environment from video: it infers a latent action between consecutive frames, so it recovers a made-up action space from action-free video by asking what latent variable best explains the transition from this frame to the next. Where V-JEPA 2 learns a world model and adds real actions later, Genie invents an action space during pretraining and lets you act inside the learned world. For control, the appeal is that the inferred latent actions might align with real ones well enough to bootstrap a policy, turning action-free video into pseudo-labeled data. The line from Genie to a deployed robot policy is longer than V-JEPA 2's and less demonstrated, but it is the cleaner statement of the dream: recover actions from video that never recorded them.

Video pretraining moved from a slide labeled "future work" to a real result on real hardware in the span of this book's writing, which is the best reason to take it seriously and the best reason to be careful about it. The V-JEPA 2-AC number says the ocean of video is usable; it does not yet say it solves the hard tasks, and the embodiment gap between a human hand and a robot gripper is a real limit the pick-and-place result was allowed to sidestep. What the section leaves you with is a shift in where the advantage sits: for years the answer to "we need more robot data" was "collect more," and video pretraining is the first serious argument that the answer might instead be "watch." The next section turns to a different resource the field is chasing, not more data but more deliberation: letting a model reason in language before it acts.

18.4 Reasoning + action: where LLM-style chains of thought meet control

Ask a language model a hard question and, if it is any good, it does not answer immediately. It writes out intermediate steps, a chain of thought, and the steps make the final answer better. The obvious transplant is to give a robot the same habit: before emitting an action, let the model reason in language about what it sees, what the goal decomposes into, and what to do next. When this book's earlier drafts were written, that was a speculative pitch. It is not speculative now. Reasoning-plus-action is an active subfield with real systems, and it carries a bonus §17.5 cared about: a model that shows its reasoning is easier to inspect than one that emits actions from a single silent pass. This section is about what that transplant buys, where it costs, and how the field is trying to keep the benefit without paying the latency.

What "reasoning before acting" actually means here

The idea has a concrete form, and it is worth pinning down before the tradeoffs make sense. An embodied chain-of-thought policy, of which ECoT (Zawalski et al., 2024) is the canonical early version, does not go straight from image-and-instruction to action tokens. It first generates intermediate text: a description of the scene, the sub-task it should tackle, sometimes the coordinates of the relevant object or a plan in words, and only then the action. The reasoning is generated by the same model, in the same autoregressive stream, so the action is conditioned on the model's own written-out plan rather than on the raw inputs alone. §8.2's Decision Transformer showed that control can be sequence modeling; embodied CoT extends the sequence backward, so that language reasoning tokens precede the action tokens the robot executes.

The reported payoff matches the language-model intuition. Reasoning helps most on the tasks that need it: long-horizon tasks where the plan matters (§18.2), tasks with linguistic ambiguity where the model has to interpret "the cup next to the blue one," and out-of-distribution situations where a policy benefits from thinking rather than reflexively pattern-matching. ECoT and its successors show generalization gains on exactly these, which makes sense, because the reasoning is where a novel instruction gets decomposed into something the low-level action head has seen before.

It also matches the intuition in the opposite direction, and this is the part people forget: reasoning does not help on tasks that do not need it, and it can hurt. A short, unambiguous grasp the policy has done ten thousand times gains nothing from a paragraph of deliberation, and the deliberation adds latency and a fresh chance to talk itself into the wrong plan. This is the same lesson §14.1 taught about dual systems, that not every action needs the slow path, and it means embodied reasoning is not a universal upgrade you bolt onto every policy. It is a tool for the hard fraction of tasks, and knowing which fraction is itself a decision the system has to make, either by a fixed schedule or, harder, by the model recognizing when it is confused enough to be worth thinking. That triaging problem, when to reason at all, is less studied than the reasoning itself and is a clean place for new work.

The cost, and why it is not a footnote

There is a hard problem baked into this, and it is not incidental: reasoning is slow. Generating a paragraph of chain-of-thought before every action means running the language model to produce dozens or hundreds of tokens, and §14.4 spent a whole section on why a VLA that cannot close its control loop fast enough is not deployable. A policy that stops to write an essay before each motion cannot run at 20 Hz. This is the tension the whole subfield is organized around, and it is the same tension §14 resolved with dual systems: a slow reasoner and a fast controller running at different rates, the reasoner setting direction a few times a second and the controller producing smooth

motion in between. Reasoning-plus-action, in its deployable form, is often a dual-system architecture wearing the reasoning label, with the chain of thought living on the slow clock.

Gemini Robotics-ER (arXiv:2510.03342), introduced in §14.3, is the industrial version of this bet. DeepMind's "Embodied Thinking" is embodied chain-of-thought at foundation-model scale, a model that reasons about the physical scene, spatial relationships, object affordances, the steps a task needs, and then drives action. It is the third or fourth entry in the dual-system family the book has been tracking, and its distinguishing feature is that the high-level system's deliberation is explicitly reasoning rather than a black-box latent. That choice is the §17.5 analyzability argument made into a product decision: a system whose intermediate reasoning is legible text is one whose mistakes you can sometimes read, which matters for a technology nobody can formally verify.

The clever fix: reason in training, not at inference

The most interesting recent move sidesteps the latency problem outright, and it is worth understanding because it is the kind of idea a student could have and extend. ERVLA (arXiv:2606.03784) asks a sharp question: if chain-of-thought helps the model learn, does the model still need to generate it at test time? Their answer is no. ERVLA trains with reasoning traces, so the model learns from the decomposed, spelled-out plans, but uses a CoT-dropout strategy so that at inference it predicts actions directly without generating the reasoning tokens. The reasoning shapes what the model learns and then gets out of the way when the clock is running. The reported numbers are strong, roughly 86.9% on LIBERO-Plus and 53.2% on VLABench, and the mechanism is the point: you can get the generalization benefit of reasoning as a training signal while paying none of the inference latency, because the essay was a scaffold, not a runtime requirement.

This resolves, at least partially, the tension the rest of the section built up. If reasoning's value is that it organizes learning, then it belongs in the loss, not in the forward pass, the same way a good curriculum shapes a student who then answers fluently without narrating every step. The open question ERVLA opens rather than closes is how much reasoning has to survive to inference. Some tasks may genuinely need runtime deliberation, a truly novel situation the training traces never covered, and dropping the chain of thought entirely would give up the ability to think on the fly that motivated the whole idea. Where the line sits, between reasoning-as-training-signal and reasoning-as-runtime-capability, is unsettled and is one of the more tractable open problems in the chapter.

There is a subtler worry the subfield has started to confront, and it is worth stating because it cuts against the whole premise. When a language model writes a chain of thought, we assume the reasoning caused the answer. That assumption is shakier than it looks: a model can produce plausible reasoning that has little to do with how it actually arrived at the action, a rationalization rather than a computation. For a robot this matters more than for a chatbot, because §17.5's analyzability argument rests on the reasoning being faithful, on the written plan actually reflecting what the policy will do. If the chain of thought is a story the model tells that does not bind its motor output, then legible reasoning buys the appearance of transparency without the substance, and a monitor reading the reasoning would be reassured by a plan the robot was never going to follow. Whether embodied chain-of-thought is faithful in this sense, whether the reasoning constrains the action or merely decorates it, is open and under-measured, and it is a place where a careful experiment, perturb the reasoning and see if the action changes, would say something the field needs to hear.

A related thread: pointing, not just planning

Reasoning about a scene is not only about task decomposition; it is also about grounding language in specific pixels, and a parallel line of work treats that as the reasoning to prioritize. Embodied-R1 (arXiv:2508.13998) and related systems emphasize spatial reasoning, getting the model to localize the object an instruction refers to, output a point or a region, and act on that grounding, which is the failure mode where a policy understands "pick up the wrench" as language but reaches for the wrong tool. This is reasoning in a narrower, more checkable sense than a full chain of thought, and it has the property that its output is verifiable: a predicted grasp point either lands on the wrench or does not, where a paragraph of reasoning is harder to score. For a reader deciding where to contribute, grounded spatial reasoning has the advantage that you can measure whether it worked.

Reasoning joined to action stopped being a thought experiment during this book's writing and became a crowded, fast-moving subfield with a clear internal logic. The promise is real, deliberation helps on the long, ambiguous, novel

tasks that reflexes fail. The cost is real too, reasoning is slow and control is fast, and the field's best answers are to split the clocks the way §14 did or, more cleverly, to use reasoning as a training signal that never has to run at inference, which is ERVLA's bet. What none of it has yet delivered is a robot that reasons its way through a genuinely new physical problem in real time, the way the language-model analogy promises; that remains ahead. With the four research bets of this chapter now on the table, cross-embodiment transfer, long-horizon dexterity, video pretraining, and reasoning, the next section steps back to ask what a reader should read next, and where a person with one robot and a year can actually push.

18.5 What to read next, and how to contribute

A book like this has a failure mode: the reader closes it impressed and does nothing, because the field looks like it belongs to a dozen labs with thousand-GPU clusters and robot fleets. That impression is half true and half a trap. The frontier results do come from big labs. But the open problems in this chapter have corners that a person with one robot, or no robot and a good simulator, can push on in a year, and several of the field's useful contributions came from exactly that. This section is the practical exit from the book: what to read to go deeper, and how to turn reading into a project that produces something real.

What to read, and in what order

Reading everything is not a plan. Reading the right five things and then building is. Here is the short path through the primary sources, chosen so each one earns its time.

Start with the papers you have already been using. OpenVLA (arXiv:2406.09246) is the model you ran in Chapter 2 and fine-tuned in Chapter 16, and rereading it now, with the whole book behind you, is different from reading it cold; the design choices that were opaque in Chapter 2 are legible after Chapters 10 through 13. Then π0 (arXiv:2410.24164), because it is the cleanest statement of the flow-matching action head that Chapter 13 built up, and because the π-family is where the continuous-action lineage keeps moving. These two are your anchors; you can run one and you understand the other.

For the open problems specifically, read one representative paper per bet rather than the survey. Open X-Embodiment (arXiv:2310.08864) for cross-embodiment, because the dataset and the RT-X transfer result are the thing every later cross-embodiment paper is arguing with. V-JEPA 2 (arXiv:2506.09985) for video pretraining, because §18.3 argued it is the first result that made the video bet concrete and the paper is where the 62-hours-to-a-policy claim is laid out. For reasoning-plus-action, read ERVLA (arXiv:2606.03784) not because it is the most famous but because its CoT-dropout idea is the most instructive: it shows you a place where a simple, sharp question, does the reasoning need to run at inference, produced a real result. Save the surveys for when you need breadth. The VLA survey at arXiv:2505.04769 and the efficient-VLA survey at arXiv:2510.24795 are good maps once you already know the territory, and useless as a first read, because a survey teaches you the names of things and not how any of them works.

Two habits make the reading compound. Keep the arXiv IDs, not the titles, because the field renames and re-releases and the ID is the stable handle, which is why this book cites by ID wherever it can (Appendix E.2 has the full list). And read the appendices and the failure sections of these papers, not just the results, because the appendix is where a paper admits what it could not get to work, and that admission is where the open problems actually live.

Where a one-robot researcher can push

The four bets are not equally accessible. Some need a fleet; some need a weekend. Matching the problem to what you have is the difference between a project you finish and one you abandon.

Cross-embodiment transfer (§18.1) sounds like it needs many robots, and the headline results do, but the tractable corner is the opposite: take a released cross-embodiment policy and study where it breaks on your one robot. The universal-action idea from UniAct (arXiv:2501.10105) makes a testable claim, that there is a shared action space above joints, and you can test it by fine-tuning a generalist onto a robot it never saw and measuring how much of its cross-robot knowledge survives. That is a one-robot experiment with a real result at the end.

Long-horizon and dexterous tasks (§18.2) are the most accessible of the four, which is the section's happiest surprise. The compounding-error curve reproduces on a tabletop: build one honestly multi-step task, run your fine-tuned policy on it fifty times, and log where in the sequence it fails and why. You will produce the per-step success-rate breakdown that most papers gesture at and few publish, and the failure taxonomy you build is exactly the kind of unglamorous, useful artifact the field is short on. Dexterity is harder because it needs a good hand, but if you have one, the generalization-to-contact question, does the grasp survive a slipperier object, is wide open and cheap to probe.

Video pretraining (§18.3) is the bet that most rewards having no robot at all, because most of the work is in simulation and on video datasets, and the label-gap problem is conceptual before it is physical. If your access is a GPU and not a robot, this is your corner: take a video-pretrained representation and measure how much robot data it actually saves on a fixed downstream task, which is the number V-JEPA 2 reported once and almost nobody has replicated.

Reasoning-plus-action (§18.4) sits in between. The ERVLA question, how much reasoning has to survive to inference, is answerable in simulation on LIBERO with a single GPU, and the grounded-spatial-reasoning line from Embodied-R1 (arXiv:2508.13998) has verifiable outputs a laptop can score. If you like the language-model side of the field more than the control side, this is where your existing intuitions transfer.

A year-1 research plan

Here is a concrete arc, the shape of a first serious project, written for someone with one arm or a simulator and a year of part-time attention.

The first two months are not research; they are reproduction, and skipping this step is the most common way a project dies. Reproduce a result you did not produce: fine-tune OpenVLA on a public dataset and match a published success rate within a few points, using the Chapter 16 recipe. If you cannot reproduce a known number, you cannot trust a new one, and the debugging you do here (§3.5) is the skill the rest of the project runs on.

Months three and four are for building the evaluation before the method, which §15.6 argued and which everyone ignores until it burns them. Pick one open problem, define the task, and build the eval that would detect progress on it: the multi-step task and its per-step logging for long horizons, the held-out embodiment for transfer, the data-efficiency curve for video pretraining. An honest evaluation you built yourself is worth more than a method, because a method without an evaluation is a demo, and the field has enough demos.

The back half of the year is the actual attempt: one change to one model, measured against the evaluation you already built, with the negative results kept. Most changes will not help, and the discipline is to report that they did not rather than to keep tuning until a number moves, which is how the reproducibility problem the field has got made. Write it up at 500 words first, the position note the §18.x exercise asks for, before you write it up at eight pages. If the result is real, the short version will already be clear; if it is not, the short version will show you why.

One more piece of advice, because it saves months. Pick a problem where you can get a signal in days, not one where the first measurement takes a quarter to set up. A student who chooses "improve dexterity on a five-fingered hand" needs the hand, the teleop rig, the data pipeline, and a working baseline before a single experiment runs, and most of that time produces no result. A student who chooses "measure how ERVLA's accuracy changes as you drop more of the reasoning trace" can get a curve on LIBERO in a week with one GPU, and the curve is a finding whether it goes up, down, or flat. Fast feedback is not a luxury for a first project; it is the thing that keeps the project alive long enough to teach you the field. The ambitious problem will still be there in year two, and you will be far better equipped to attack it having shipped something small first.

There is also a quieter kind of contribution worth naming, because it suits some people better than novel methods. Reproduction and negative results are undersupplied and genuinely valued: a clean writeup showing that a published number does not replicate on a second robot, or that a celebrated trick does not help outside its original benchmark, is a real service to a field with a reproducibility problem, and it asks for rigor rather than invention. If you are better at careful measurement than at inventing architectures, that is not a lesser path; it is the path the field is shortest on.

The through-line of this book has been that these models are more capable than they are understood, and that gap is not a reason to stand back. It is the opening. The frontier moves fast enough that a careful person with a modest setup and a real evaluation can find something the big labs skipped, because the big labs are optimizing headline

numbers and the useful corners are full of unmeasured failures. The last content section made this personal on purpose. The summary that follows collects what the chapter established, and the closing exercise turns this plan into the first 500 words of it.

18.6 Summary

Chapter 17 ended on an honest boundary: nobody can certify a foundation-model robot safe, and deployment is managed risk watched closely. Chapter 18 took the problems that honesty exposes and turned them into a map of where the field is trying to go, and into a plan for how a reader might help it get there. This is the last chapter, so the summary does double duty. It collects the four open bets the chapter laid out, and it closes the arc the book opened in §1.1, where "action" was named the hard part of robotics and the reason a robot that can describe a task still cannot reliably do it.

The ideas worth carrying forward

Generalization across embodiments is partly solved for similar robots and open for different ones, and the classical locomotion boundary is dissolving. §18.1 was blunt about where the "foundation" promise stands. Pooling data across robots, the Open X-Embodiment bet, produces real positive transfer within a rough class of similar machines, and falls apart across genuinely different bodies, because a 7-vector of joint deltas means something different to every robot and simple concatenation of action spaces does not survive the differences. The technical core is a representation question, what should the policy emit so that it means the same thing to many robots, and the candidates all fail somewhere: Cartesian pose is agnostic but empty for a five-fingered hand, joint targets are precise but untransferable, and a learned universal action space (UniAct, arXiv:2501.10105) is the ambitious middle whose reality is unproven. The live development is whole-body neural control, Helix-02's System 0 and Gemini Robotics 2's unified loco-manipulation policy, which folds the model-based locomotion that §4.4 called load-bearing into the same learned system that drives the hands, and so scrambles the clean classical/learned boundary Chapter 4 drew.

Long-horizon and dexterous tasks are two different failures, and neither is close to a human. §18.2 separated what gets bundled. Long horizons fail by the compounding error of §6.3 scaled up, where twelve steps at 95% each multiply to a coin flip, and the attacks are learned task decomposition (Long-VLA) and reinforcement learning from the robot's own experience bolted onto a foundation policy ($\pi^*0.6$), which is the field circling back to §6.5's tool for the robustness cloning cannot buy. Dexterity fails on contact and control bandwidth, where a twenty-DOF hand needs tactile feedback and force modulation at a rate most VLAs do not run, and the response is purpose-built dexterous foundation models (GENE, RLDX-1) whose very existence is a bet that dexterity will not emerge for free from a generalist. The gap to a human hand executing a two-minute plan without thinking is still large, and the benchmarks measure these tasks worst, which is part of why progress is hard to read.

Video pretraining stopped being a promise and produced a concrete result. §18.3 tracked the data bottleneck to its most hopeful answer. Robot data is scarce and video is an ocean, but video has no action labels, which for years kept the ocean out of reach. V-JEPA 2 (arXiv:2506.09985) learned a world model from over a million hours of internet video with no actions at all, and its action-conditioned variant, post-trained on under 62 hours of robot data, did zero-shot pick-and-place at roughly 80% where a conventional generalist managed roughly 15%. Take the numbers as provisional and the shape holds: a world model built almost entirely from watching, with a thin layer of robot data on top, beat the conventional recipe on a task neither was trained on. What it proves is narrow, short forgiving tasks, not long dexterous ones, and the human-to-robot embodiment gap remains a real limit. What it changes is the economics, because 62 hours is a weekend, not a fleet-year.

Reasoning joined to action is a real subfield now, and its central tension is latency. §18.4 followed the language-model habit of thinking before answering into control. Embodied chain-of-thought (ECoT, and Gemini Robotics-ER's "Embodied Thinking" at scale) generates intermediate reasoning before the action, and it helps most on the long, ambiguous, novel tasks where reflexes fail. It also collides head-on with §14.4's latency budget, because a model that writes an essay before each motion cannot close a 20 Hz loop, and the deployable forms put the reasoning on the slow clock of a dual system. The sharpest recent move, ERVLA (arXiv:2606.03784), sidesteps the cost entirely by training with reasoning traces and dropping them at inference, using reasoning as a learning signal rather than a runtime requirement, which is the kind of clean idea that tells you the subfield is still full of reachable questions.

The frontier is accessible in places, and a modest setup plus a real evaluation is a contribution engine. §18.5 refused the impression that this field belongs only to fleet-owning labs. Each open problem has a corner a one-robot or no-robot researcher can push: the where-does-it-break study for cross-embodiment transfer, the per-step failure taxonomy for long horizons, the data-efficiency replication for video pretraining, the inference-reasoning question for reasoning-plus-action. The year-1 plan puts reproduction first, evaluation before method, and one measured change with negative results kept, because a method without an evaluation is a demo and the field has enough of those.

One connective observation ties the four bets together, and it is worth stating because it is the picture the TOC asked you to be able to draw. All four are attacks on the same shortage: robots that generalize. Cross-embodiment work chases generalization across bodies, long-horizon and dexterity work chases generalization across task length and contact, video pretraining chases generalization from cheap data to expensive skills, and reasoning chases generalization to situations the training set never showed. They differ in where they attack, not in what they want, and a paper that seems to belong to one bet often borrows from another: whole-body control is a cross-embodiment result that only works because of the dual-system reasoning split, and video pretraining is a data story whose payoff is measured in the long-horizon tasks it might eventually reach. Seeing the bets as one problem viewed from four sides is what lets you place a new result rather than filing it as noise, and it is the difference between following the field and understanding it.

What you should be able to do now

Four things, matching the chapter's objectives.

You should be able to *summarize the open scientific questions in action models as of this writing.* Not as a list of buzzwords but as four specific bets with specific failure modes: what "generalize across embodiments" precisely means and why it is a representation problem, why long horizons and dexterity are different problems, what video pretraining can and cannot yet do, and why reasoning helps and what it costs.

You should be able to *map the current research bets onto a single picture.* World models, dexterous foundation models, humanoid and whole-body VLAs, video-pretrained action models, and reasoning-plus-action are not five unrelated efforts; they are attacks on the same shortage of capability from different sides, and you can place a new paper you encounter into that picture rather than treating it as noise.

You should be able to *identify which bet you are best positioned to contribute to.* Given your actual resources, one robot, a simulator, a GPU and no robot, you can name the corner of the frontier that matches what you have, because §18.5 matched each problem to the setup it needs and the accessible ones are not the ones with the biggest headlines.

You should be able to *leave the book with a reading list and a year-1 plan.* You have a short ordered path through the primary sources, a reason for each one, and a concrete first-project arc that starts with reproduction and ends with a measured result written up honestly.

Where the book leaves you

This chapter closes a book that started with a claim in §1.1: that action is the hard part, that the gap between a system that understands a task and one that reliably performs it is where robotics actually lives. Every part since was a way of narrowing that gap. Part 2 traced the lineage, planning and control and reinforcement learning and imitation, that defined the problem before foundation models arrived. Part 3 built the modern components, sequence models and world models and diffusion, that the VLAs are assembled from. Part 4 went through the foundation action models themselves, from RT-1 to $\pi 0$ to the dual-system humanoids, in enough depth to read the papers and not just the press. Part 5 was the engineering, fine-tuning and safety and deployment, that turns a checkpoint into something that runs near a person.

What the book cannot give you is a finished field, because there is not one. The honest state, the one Chapter 17 insisted on and Chapter 18 mapped, is that these models are the most capable robot controllers ever built and the least understood, that they generalize impressively and fail unpredictably, and that the questions between here and a robot you would trust in your kitchen are open in the literal sense: unanswered, and available. That is the note to end on. The field is young enough that the reader who finishes this book, reproduces a result, builds an honest

evaluation, and pushes on one corner is not a spectator to it. The exercise that follows is the first 500 words of that push.

18.x Hands-on exercise + chapter references

Every other chapter's exercise had you run code. This one, the last in the book, has you do the thing that actually starts a research career: read the current literature on one open problem and take a position on it in writing. The TOC states the target plainly: pick one open problem from the chapter, find three recent papers on it, and write a 500-word position note. That sounds soft next to wrapping a safety shield around OpenVLA, and it is the harder skill, because a position note forces you to say what you think is true and what you would do about it, which running someone else's code never does. §18.5 argued that a method without an evaluation is a demo. A field without a position is a reading list, and this exercise is how you stop being a reading list.

No hardware is required, which is the point. The gap between a person who has read this book and a person who contributes to the field is not a GPU; it is the habit of turning reading into a claim. This exercise builds that habit once, in 500 words, so that the year-1 plan from §18.5 has a first page.

Exercise 18.x.1 — Pick the problem that matches what you have

Do not pick the problem that sounds most exciting. Pick the one you could actually run an experiment on, because a position note you cannot follow with a project is an essay, and §18.5 matched each open problem to the setup it needs. If you have one robot arm, the long-horizon failure taxonomy (§18.2) or the cross-embodiment where-does-it-break study (§18.1) is yours. If you have a GPU and no robot, video pretraining's data-efficiency question (§18.3) or the ERVLA inference-reasoning question (§18.4) is yours. Write one sentence naming the problem and one sentence naming the experiment you could run on it. If the second sentence is hard to write, pick a different problem; the difficulty is telling you the truth.

Exercise 18.x.2 — Find three recent papers, the right way

Three papers, and the selection matters more than the number. Do not pick three that agree. Pick the one that best states the current approach, one that reports a result complicating it, and one from the last few months that you found yourself, not from this book's reference list, because the skill being tested is finding the frontier after the book stops. Use the arXiv listing for cs.RO and cs.LG, follow the citation graph forward from a paper you know using Semantic Scholar or Connected Papers, and read each paper's failure section before its results, per §18.5. For each, write two lines: what it claims, and what it admits it could not do. That second line is where your position will come from.

Exercise 18.x.3 — Write the 500 words, with a claim in the first sentence

Now the note. Five hundred words, and the first sentence is a claim, not a background statement. "Cross-embodiment transfer will not be solved by a universal action space, because the three papers below all report the same failure at the same place" is a first sentence. "Cross-embodiment transfer is an important open problem" is not, because it commits to nothing. Structure the rest in four short movements rather than a five-paragraph essay: the claim, the evidence from your three papers, the strongest objection to your claim stated in one honest sentence, and the experiment you would run to settle it. Keep the experiment small enough to actually do, because §18.5's year-1 plan says the note is the first page of the project, and a project you cannot start is a project you will not finish.

Two failure modes to avoid, both common. The hedging note, which surveys the three papers evenhandedly and concludes "more research is needed," is not a position and gets no credit; the whole exercise is refusing that sentence. The overconfident note, which claims the problem is trivially solved by an idea you have not tested, is worse, because it skips the objection movement that keeps you honest. A good note is a claim you would defend and an experiment that could prove you wrong, which is the shape of every result worth publishing.

Going further

Two extensions turn the exercise into the project. Run the small experiment the note ends with, even a rough version on a subset, and add a paragraph reporting what happened, because a position note backed by one data point outranks a dozen that are pure argument. And send the note to one person working on the problem, a paper author, a lab's public channel, a course instructor, and ask the single question the note raised for you; the field is small and young enough (§18.5) that this works more often than you expect, and a real answer from someone in the middle of the problem is worth more than another paper.

Chapter 18 reading list

Cited across §18.1–§18.6, grouped by the bet each reference serves. Full entries for the whole book live in Appendix E.2; this is the chapter-local subset, and it doubles as the ordered reading path §18.5 recommended.

Cross-embodiment generalization

- Open X-Embodiment Collaboration (2023). "Open X-Embodiment: Robotic Learning Datasets and RT-X Models." arXiv:2310.08864. §18.1's anchor: the pooled multi-robot dataset and the positive-transfer result every later cross-embodiment paper argues with.
- Octo Model Team (2024). "Octo: An Open-Source Generalist Robot Policy." arXiv:2405.12213. §18.1's heterogeneity-absorbing design, with swappable input and output heads; also §18.3's conventional-generalist baseline that V-JEPA 2-AC beat.
- Zheng, J., et al. (2025). "Universal Actions for Enhanced Embodied Foundation Models (UniAct)." arXiv:2501.10105. §18.1's learned universal action space, the cleanest statement of the claim that there is a shared behavior level above joints and below language.
- Google DeepMind (2026). "Gemini Robotics 2." Announcement, 2026-07-30. §18.1's whole-body loco-manipulation result, the first VLA to drive legs, torso, arms, and hands under one policy, dissolving the §4 locomotion boundary.

Long-horizon and dexterous tasks

- Long-VLA (2025). "Long-VLA: Unleashing Long-Horizon Capability of Vision-Language-Action Models." arXiv:2508.19958. §18.2's learned task-decomposition attack on the compounding-error problem.
- Fang, H., et al. (2025). "LiLo-VLA." arXiv:2602.21531. §18.2's companion long-horizon result in the same decomposition vein.
- Physical Intelligence (2025). "π*0.6 / RECAP: Learning from Experience." pi.website/download/pistar06.pdf. §18.2's reinforcement-learning-from-experience result on a foundation policy, the return of §6.5's tool for the robustness cloning cannot buy.

Video-pretrained action models

- Assran, M., et al. (2025). "V-JEPA 2: Self-Supervised Video Models Enable Understanding, Prediction and Planning." arXiv:2506.09985. §18.3's central result: a world model from over a million hours of video, post-trained into a zero-shot pick-and-place policy on under 62 hours of robot data.
- Bruce, J., et al. (2024). "Genie: Generative Interactive Environments." ICML 2024. §18.3's latent-action branch, inferring a made-up action space from action-free video, the cleaner statement of recovering actions video never recorded.
- Nair, S., et al. (2022). "R3M: A Universal Visual Representation for Robot Manipulation." CoRL 2022. §18.3's representation-only precursor: video pretraining that improves the visual front end but still needs robot demonstrations for the policy.

Reasoning joined to action

- Zawalski, M., et al. (2024). "Robotic Control via Embodied Chain-of-Thought Reasoning (ECoT)." §18.4's canonical embodied chain-of-thought, reasoning tokens generated before the action in one autoregressive stream.

- ERVLA (2026). "Revisiting Embodied Chain-of-Thought for Generalizable Robot Manipulation." arXiv:2606.03784. §18.4's sharpest move: train with reasoning traces, drop them at inference via CoT-dropout, paying none of the latency for the learning benefit.
- Gemini Robotics-ER (2026). "Embodied Thinking." arXiv:2510.03342. §18.4's foundation-scale embodied reasoning, and §17.5's analyzability argument turned into a product choice: legible reasoning over an opaque pass.
- Embodied-R1 (2025). arXiv:2508.13998. §18.4's grounded-spatial-reasoning line, where the reasoning output, a grasp point, is verifiable in a way a paragraph of chain-of-thought is not.

Surveys, for breadth after the primary sources

- VLA Models: Concepts, Progress, Applications and Challenges (2025). arXiv:2505.04769. §18.5's recommended map of the whole area, useful once you know the territory and useless as a first read.
- Efficient VLA Survey (2025). arXiv:2510.24795. §18.5's companion survey on the latency and compute constraints that §14.4 and §18.4 both run into.

Chapter summary

Chapter 18 took the open problems that Chapter 17's honesty exposed and turned them into a map and a plan. You can now summarize the four live research bets as specific questions with specific failure modes rather than as buzzwords: cross-embodiment transfer as a representation problem that works within a robot class and breaks across bodies, with whole-body neural control now dissolving the classical locomotion boundary from Chapter 4; long-horizon and dexterity as two distinct failures, one of accumulated error and one of contact bandwidth, neither near a human; video pretraining as the data bottleneck's most concrete answer yet, proven on short tasks by V-JEPA 2 and unproven on hard ones; and reasoning-plus-action as a real subfield whose central tension is the latency of thinking, with ERVLA's train-then-drop trick the sharpest current resolution. You can map these onto one picture as coordinated attacks on the same capability shortage, identify which one your actual resources fit, and leave with an ordered reading path and a year-1 arc that puts reproduction and evaluation before method. And after this exercise you have written the first 500 words of that arc: a position note that makes a claim, backs it with three papers you chose, states its own strongest objection, and ends in an experiment small enough to run. That note is where this book stops being something you read and starts being something you do. The field is young, the useful corners are full of unmeasured failures, and the person who finishes here and pushes on one of them is a participant in it, not an observer.

Appendix A — Linear algebra refresher

Appendix A. Linear algebra refresher

The body of this book treats linear algebra as a working tool, not a subject. Vectors, matrices, eigenvalues, and singular value decompositions all appear in chapter examples without ceremony, on the assumption that the reader has seen them before and needs only the fingertip-level fluency to keep up. This appendix is the safety net. Every object discussed here is illustrated with a robotics example from the body of the book — a manipulator Jacobian, a covariance matrix, an action-token codebook — so that the math arrives attached to a thing it does, rather than as a list of definitions. The treatment is intentionally short. If you find yourself wanting more, Strang (2016) is the canonical undergraduate text and Trefethen and Bau (1997) is the numerical-linear-algebra reference the rest of the field reaches for.

A.1 Vectors and inner products

A vector is an ordered tuple of real numbers. The notation $x \in \mathbb{R}^n$ says x has n components, indexed $x_1, x_2, \dots, x_n$. In robotics, the most frequent vectors you will write down are joint configurations $q \in \mathbb{R}^7$ for a Franka-class arm, end-effector poses $p \in \mathbb{R}^6$ when orientation is expressed as an axis-angle vector, and the seven-vector of end-effector deltas plus gripper command that OpenVLA (arXiv:2406.09246) emits at 5 Hz.

Two vectors $x, y \in \mathbb{R}^n$ have an *inner product* $\langle x, y \rangle = x^\top y = \sum_i x_i y_i$. The inner product is a scalar; geometrically it equals $\|x\| \, \|y\| \cos\theta$, where θ is the angle between x and y and $\|x\| = \sqrt{x^\top x}$ is the Euclidean norm. Three properties of inner products do load-bearing work in the body of the book. Bilinearity — linear in each argument when the other is fixed — is what makes a transformer's attention pattern $\mathrm{softmax}(QK^\top/\sqrt{d})$ computable in matrix form. Symmetry, $\langle x, y \rangle = \langle y, x \rangle$, is why a Gram matrix $X^\top X$ is symmetric. Positive-definiteness, $\langle x, x \rangle \geq 0$ with equality only at $x = 0$, is what lets the loss function in §3.3 be bounded below by zero.

Three derived quantities you will see often. The *Euclidean distance* $\|x - y\|$ measures how far apart two points are in configuration space. The *cosine similarity* $\langle x, y \rangle / (\|x\| \, \|y\|)$ is the inner product after length normalization; CLIP-style contrastive losses (Radford et al., 2021, ICML 2021) compute cosine similarity between image and text embeddings and that is the entire learning signal. The *projection* of x onto a unit vector u is $(u^\top x)\, u$; projection onto the column space of a matrix is the operation a least-squares solver performs under the hood.

A.2 Matrices as linear maps

A matrix $A \in \mathbb{R}^{m \times n}$ is two objects at once. As data, it is an m-by-n array of numbers, indexed A_{ij} for $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, n\}$. As a function, it is a *linear map* from $\mathbb{R}^n$ to $\mathbb{R}^m$: every vector $x \in \mathbb{R}^n$ has an image $Ax \in \mathbb{R}^m$, and the map satisfies $A(\alpha x + \beta y) = \alpha\, Ax + \beta\, Ay$ for any scalars. The second view is the one that matters; once you read A as a function, every operation the rest of the book performs on it becomes intelligible.

Three matrix examples that recur. The *manipulator Jacobian* $J(q) \in \mathbb{R}^{6 \times 7}$ of §3.1 maps joint velocities $\dot{q} \in \mathbb{R}^7$ to end-effector velocities $\dot{p} = J(q)\, \dot{q} \in \mathbb{R}^6$. The *covariance matrix* $\Sigma \in \mathbb{R}^{n \times n}$ of a Gaussian random vector encodes the pairwise covariances of its components; it is symmetric and positive semi-definite, and the diagonal entries are the per-component variances. The *weight matrix* $W \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ of a fully connected layer maps a d_{in}-dimensional input

embedding to a d_{out}-dimensional output embedding; the entire weight budget of OpenVLA — about seven billion numbers — is a stack of such matrices.

Matrix-matrix multiplication $C = AB$, where $A \in \mathbb{R}^{m\times k}$ and $B \in \mathbb{R}^{k\times n}$, produces $C \in \mathbb{R}^{m\times n}$ with entries $C_{ij} = \sum_\ell A_{i\ell} B_{\ell j}$. As linear maps, AB is the *composition* of B followed by A: $(AB)x = A(Bx)$. Composition is associative but not commutative, which is why the order of matrix factors in a transformer attention computation matters and why most GPU training time on a VLA is spent inside one of a small number of large matrix multiplies.

The *transpose* $A^\top \in \mathbb{R}^{n\times m}$ is the matrix with swapped indices, $(A^\top)_{ij} = A_{ji}$. The transpose appears ubiquitously because $\langle Ax, y\rangle = \langle x, A^\top y\rangle$ — moving a linear map between the two arguments of an inner product just transposes it — and that identity is the backbone of backpropagation, where the gradient of a loss with respect to an input of a layer is computed by multiplying the upstream gradient by the layer's transposed weight.

A.3 Rank, null space, and the four fundamental subspaces

The *rank* of a matrix $A \in \mathbb{R}^{m\times n}$ is the dimension of its column space — the subspace of $\mathbb{R}^m$ spanned by its columns. Rank is at most $\min(m, n)$. When rank equals $\min(m, n)$ the matrix is *full rank*; otherwise it is *rank-deficient* and behaves degenerately under inversion.

The *null space* of A, written $\ker(A)$ or $\text{null}(A)$, is $\{x \in \mathbb{R}^n : Ax = 0\}$. Its dimension is $n - \text{rank}(A)$. For the manipulator Jacobian of a redundant arm (7-DOF arm, 6-DOF task space, so $J \in \mathbb{R}^{6\times 7}$), the null space is a one-dimensional subspace of joint-velocity vectors that produce zero end-effector velocity — that is, the directions the joints can move without the gripper moving. Null-space control exploits this freedom to keep the elbow up while the gripper does work, and is one of the reasons 7-DOF arms are useful when 6-DOF ones would also reach the target.

The four fundamental subspaces — column space, null space, row space, left null space — are connected by the *fundamental theorem of linear algebra*: column space and left null space are orthogonal complements in $\mathbb{R}^m$; row space and null space are orthogonal complements in $\mathbb{R}^n$. The practical implication for the book is that a least-squares solution always exists and is unique up to a shift in the null space, and that the Moore-Penrose pseudoinverse — used by every numerical inverse-kinematics solver in §4.2 — picks the shift that minimizes the norm of x.

A.4 Eigenvalues and eigenvectors

An *eigenvector* of a square matrix $A \in \mathbb{R}^{n\times n}$ is a nonzero vector v such that $Av = \lambda v$ for some scalar λ, the corresponding *eigenvalue*. Geometrically, v is a direction along which A acts as pure scaling. A symmetric matrix has n orthogonal real eigenvectors and n real eigenvalues; the *spectral theorem* writes $A = V\Lambda V^\top$ where V stacks the eigenvectors in columns and Λ is the diagonal matrix of eigenvalues.

Eigenvalues do interpretable work in three places in the book. In control, the eigenvalues of the closed-loop dynamics matrix tell you whether a controller is stable (all eigenvalues in the open left half of the complex plane, for continuous-time systems). In probability, the eigenvalues of a covariance matrix are the variances along the principal axes of the Gaussian; this is the math under principal component analysis. In optimization, the eigenvalues of the Hessian of a loss function are the local curvatures; a learning-rate schedule that works well on a flat eigenvalue spectrum will diverge on a steep one, which is one motivation for adaptive optimizers like Adam.

A diagonalizable matrix $A = V\Lambda V^{-1}$ is essentially trivial to power: $A^k = V\Lambda^k V^{-1}$, where Λ^k is the diagonal matrix with λ_i^k on its diagonal. This is the trick value iteration in §5.2 implicitly uses: the Bellman operator is contractive with rate γ, which is its spectral radius, and the iteration converges geometrically at exactly that rate.

A.5 Singular value decomposition

The singular value decomposition (SVD) is the workhorse of numerical linear algebra. Every matrix $A \in \mathbb{R}^{m\times n}$ — not just symmetric, not just square — admits a factorization $A = U\Sigma V^\top$ where $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$ are orthogonal (columns orthonormal) and $\Sigma \in \mathbb{R}^{m\times n}$ is rectangular-diagonal with non-negative entries $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0$. The σ_i are the *singular values* of A.

Three uses of SVD recur. First, the *pseudoinverse*: $A^+ = V\Sigma^+ U^\top$, where Σ^+ inverts the nonzero singular values and zeros the rest. The damped least squares IK solver of §4.2 is structurally a pseudoinverse with the small singular

values truncated, which is what prevents joint velocities from blowing up near a Jacobian singularity. Second, *low-rank approximation*: the best rank-k approximation of A in the Frobenius norm is obtained by zeroing all but the top k singular values; this is the math under LoRA (Hu et al., 2022, ICLR 2022) and every adapter-based fine-tuning technique in Chapter 16. Third, *conditioning*: the ratio $\sigma_1/\sigma_{\min(m,n)}$ is the condition number of A, and a large condition number means the matrix is close to singular and inverting it amplifies noise.

A.6 Quadratic forms and positive-definiteness

A *quadratic form* is a function $f(x) = x^\top A x$ for some symmetric matrix A. The form is *positive definite* if $x^\top A x > 0$ for all nonzero x, *positive semi-definite* if $x^\top A x \geq 0$, and *negative definite* if the inequality reverses. The Hessian of the loss function in §3.5 is symmetric and locally positive semi-definite near a minimum; the inertia matrix $M(q)$ in §4.3's manipulator equation is positive definite for every physically realizable configuration, which is exactly why the computed-torque controller from that section is well-posed.

Positive-definite matrices have all-positive eigenvalues, are invertible, and admit a *Cholesky factorization* $A = LL^\top$ with L lower triangular. Cholesky is what every Gaussian sampler in the book uses internally: to sample $x \sim \mathcal{N}(0, A)$, draw $z \sim \mathcal{N}(0, I)$ and return Lz.

A.7 Rotation matrices, $\mathrm{SO}(3)$, and the rest of the rigid-body story

Robotics adds a piece of structured linear algebra that pure ML does not: rotations. A *rotation matrix* in three dimensions is a matrix $R \in \mathbb{R}^{3\times 3}$ with $R^\top R = I$ and $\det R = +1$. The set of all such matrices is the *special orthogonal group* $\mathrm{SO}(3)$; it is a three-dimensional manifold inside the nine-dimensional space of 3×3 matrices.

The body of the book treats $\mathrm{SO}(3)$ mostly through proxy representations — axis-angle vectors, quaternions, Euler angles — because that is what neural-network policies emit. Three things to remember about all of them. They are not unique: every rotation has two quaternion representations $\pm q$, and twelve Euler-angle conventions exist. Linear interpolation between rotation representations is not rotation interpolation: averaging two quaternions componentwise and renormalizing is not the same as the geodesic between them on $\mathrm{SO}(3)$, and the difference shows up as wobble in §10.2's Diffusion Policy outputs if the model parameterizes orientation wrong. The Jacobian of a rotation representation with respect to its parameters is non-trivial; the right object for differentiating through rotations is the *exponential map* from the Lie algebra $\mathfrak{so}(3)$ (skew-symmetric matrices) to $\mathrm{SO}(3)$. Murray, Li, and Sastry (1994) is the canonical reference for the Lie-group story; the body of the book uses the exponential map without proof.

A.8 Numerical linear algebra in two paragraphs

Almost every matrix operation in the body of the book is computed numerically by a library — NumPy, PyTorch, JAX — that uses BLAS underneath. The relevant practical knowledge is small. Floating-point matrix multiplication is associative in mathematics but not in code: $(AB)C$ and $A(BC)$ can differ at the last few bits, and a long chain of matrix multiplications can accumulate small errors that compound. This is why training in mixed precision (fp16 or bf16) sometimes diverges where fp32 does not, and why VLA training recipes are picky about which layers are kept in fp32.

Matrix inversion is almost always the wrong operation to call. Instead of computing $A^{-1}b$, solve the linear system $Ax = b$ directly with an LU or QR factorization; the result is the same and the conditioning is better. Pseudoinverses for non-square matrices are computed via SVD, not by $A^\top(AA^\top)^{-1}$ or analogues; the latter formula is correct on paper and numerically catastrophic when A is poorly conditioned, which is the most common case in inverse-kinematics work near a singularity.

A.9 A quick reference of identities

A handful of identities are used so often in the body of the book that they are worth memorizing.

The chain rule for matrix-valued functions: if $f : \mathbb{R}^n \to \mathbb{R}^m$ and $g : \mathbb{R}^m \to \mathbb{R}^p$, then the Jacobian of the composition $g \circ f$ at x is $J_{g\circ f}(x) = J_g(f(x))\, J_f(x)$. This is the matrix form of the chain rule and the mathematical backbone of backpropagation.

The matrix-multiplication identity for transposes: $(AB)^\top = B^\top A^\top$. Used implicitly every time gradients flow backward through a linear layer.

The Woodbury identity for low-rank updates of an inverse: $(A + UV^\top)^{-1} = A^{-1} - A^{-1}U(I + V^\top A^{-1}U)^{-1}V^\top A^{-1}$. This is the identity under LoRA's training-time inference path, and the identity Kalman filters use to update a covariance matrix from a low-dimensional measurement.

The trace cyclicity property: $\text{tr}(ABC) = \text{tr}(BCA) = \text{tr}(CAB)$. This is used in derivations of every loss whose gradient involves a trace, and it is the identity behind the "reparameterization trick" in §3.4's discussion of VAEs.

The Frobenius inner product: $\langle A, B \rangle_F = \text{tr}(A^\top B) = \sum_{ij} A_{ij} B_{ij}$. This is the inner product that makes the space of matrices into a vector space; the Frobenius norm $\|A\|_F = \sqrt{\langle A, A \rangle_F}$ is what "distance between weight matrices" almost always means.

Five identities, four sections of background, one quick-reference list — the rest is in the body of the book. If you forget what $A^\top A$ is, look here; if you forget why the singular values matter to a controller, look at §4.2 and then come back.

Appendix B — Probability and information theory

Appendix B. Probability and information theory

Action models are statistical objects. The policy $\pi(a \mid s)$ is a conditional distribution; the loss in §3.4 is an expected negative log-likelihood; the KL divergence appears in PPO's clipping objective, in the ELBO of a VAE-style world model, and in the contrastive loss that trained CLIP. The body of the book invokes these objects without re-deriving them. This appendix is the bench manual: every concept is stated precisely once, illustrated with one robotics example, and linked back to the section where it does its real work.

B.1 Probability spaces, random variables, and distributions

A *probability space* is a triple $(\Omega, \mathcal{F}, P)$: a sample space Ω, a σ-algebra $\mathcal{F}$ of subsets of Ω called events, and a probability measure P that assigns each event a number in $[0, 1]$ with $P(\Omega) = 1$. For the body of this book the measure-theoretic scaffolding can stay implicit; what matters is that there is a procedure (sometimes physical, sometimes simulated) that produces outcomes, and a function that assigns probabilities to sets of those outcomes.

A *random variable* $X : \Omega \to \mathcal{X}$ is a function from the sample space to a value space. When $\mathcal{X}$ is a finite or countable set, X is *discrete*; when $\mathcal{X} = \mathbb{R}^n$, it is *continuous*. A discrete random variable is fully described by its probability mass function $p(x) = P(X = x)$; a continuous one by its probability density function $p(x)$, where $P(X \in B) = \int_B p(x)\, dx$ for measurable sets B. The notation $X \sim p$ means "X is distributed according to p" and is the line you will see at the top of every algorithm box from §3.3 onward.

Three distributions appear in this book often enough to be worth naming. The *Bernoulli* $\mathrm{Bern}(\theta)$ describes a coin flip with bias θ and is the distribution behind every binary classification head. The *categorical* $\mathrm{Cat}(\theta_1, \dots, \theta_K)$ describes a single draw from K options with probabilities θ_i; the action-token distribution of RT-1 (arXiv:2212.06817) and OpenVLA (arXiv:2406.09246) is a categorical over 256 bins per action dimension. The *multivariate Gaussian* $\mathcal{N}(\mu, \Sigma)$ with mean $\mu \in \mathbb{R}^n$ and positive-definite covariance $\Sigma \in \mathbb{R}^{n \times n}$ has density

$$p(x) = \frac{1}{(2\pi)^{n/2} |\Sigma|^{1/2}} \exp\!\Big(-\tfrac{1}{2}(x-\mu)^\top \Sigma^{-1} (x-\mu)\Big),$$

and is the workhorse model for sensor noise, dynamics noise, and the action distribution of every Gaussian policy in §7.

B.2 Expectation, variance, and the law of large numbers

The *expectation* of a random variable is its probability-weighted average:

$$\mathbb{E}[X] = \begin{cases} \sum_x x\, p(x) & \text{discrete,} \\ \int x\, p(x)\, dx & \text{continuous.} \end{cases}$$

Expectation is linear: $\mathbb{E}[\alpha X + \beta Y] = \alpha\,\mathbb{E}[X] + \beta\,\mathbb{E}[Y]$, with no independence required. This single property is why so much of the math in reinforcement learning works: the expected return decomposes into a sum of expected per-step rewards regardless of how the random states and actions are correlated through the policy. The *variance* $\mathrm{Var}[X] = \mathbb{E}[(X - \mathbb{E}[X])^2]$ measures spread. For a vector-valued random variable, the analogue is the covariance matrix $\Sigma = \mathbb{E}[(X-\mu)(X-\mu)^\top]$, symmetric and positive semi-definite as Appendix A.6 promised.

The *law of large numbers* says that the empirical mean of N independent samples from p converges to $\mathbb{E}[X]$ as $N \to \infty$. This is the mathematical license for Monte Carlo estimation: $\mathbb{E}_p[f(X)] \approx \frac{1}{N}\sum_{i=1}^{N} f(X_i)$ with X_i drawn from p. The *central limit theorem* sharpens the law into a rate: the standard error of the empirical mean scales as $1/\sqrt{N}$, which is why doubling your training data only reduces the policy gradient estimator's noise by about 30%. Every "we trained for 5 seeds and report mean ± std" line in a robot-learning paper is a finite-sample version of these two theorems.

B.3 Conditional probability, independence, and Bayes' rule

Two events A and B are *independent* if $P(A \cap B) = P(A)P(B)$. The conditional probability of A given B is

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)},$$

defined whenever $P(B) > 0$. *Bayes' rule* is the same identity rearranged: $P(A \mid B) = P(B \mid A)P(A)/P(B)$. In the body of the book, Bayes' rule shows up in the IRL discussion of §6.4 (recover the reward R given demonstrations τ by computing $P(R \mid \tau)$), in the diffusion-model derivation of §10.1 (the reverse process is the conditional density of the cleaner sample given the noisier one), and implicitly in every Kalman-filter-style state estimator a real robot runs under its perception stack.

The closely related concept is *conditional independence*: X and Y are conditionally independent given Z if $P(X, Y \mid Z) = P(X \mid Z)P(Y \mid Z)$. The *Markov property* in §5.1 is precisely the claim that the future is conditionally independent of the past given the present state, and it is the single assumption that makes the MDP formalism tractable.

B.4 KL divergence and the Kullback-Leibler family

The Kullback-Leibler divergence between two distributions p and q on the same space is

$$\mathrm{KL}(p \,\|\, q) = \mathbb{E}_p\!\left[\log \frac{p(X)}{q(X)}\right] = \int p(x) \log \frac{p(x)}{q(x)}\, dx.$$

KL is non-negative and zero only when $p = q$ almost everywhere. It is not symmetric — $\mathrm{KL}(p\|q) \neq \mathrm{KL}(q\|p)$ in general — and it is not a metric, but it is the *correct* divergence for almost every information-theoretic argument in machine learning. Three concrete uses to keep in mind. PPO (Schulman et al., 2017, arXiv:1707.06347) penalizes $\mathrm{KL}(\pi_{\text{new}}\|\pi_{\text{old}})$ to prevent destructive policy updates and the size of that penalty is the single most-tuned hyperparameter of the algorithm. The VAE objective of §3.4 includes a $\mathrm{KL}(q\|p)$ term that pushes the encoder distribution toward a Gaussian prior; Kingma and Welling (2014, arXiv:1312.6114) is the canonical reference. Diffusion training loss is, after rearrangement, a sum of per-timestep KL divergences between the true and learned reverse transitions; the noise-prediction objective most VLAs use is the "reparameterization" of that KL sum.

A practical note: KL is finite only when q assigns positive probability everywhere p does. When $q(x) = 0$ but $p(x) > 0$, the KL is infinite. Numerical implementations therefore add a small ϵ inside the log or smooth the distributions; if a training run's loss suddenly spikes to NaN, "a distribution went to zero in the support of the other" is the second culprit to check, after "the learning rate is too high."

B.5 Entropy and cross-entropy

The *entropy* of a discrete distribution p is

$$H(p) = -\sum_x p(x) \log p(x) = \mathbb{E}_p[-\log p(X)].$$

Entropy is non-negative and is maximized by the uniform distribution on a given support. It measures the average number of nats (or bits, if base 2) needed to describe a sample from p. In control, high policy entropy means the policy is exploring; low policy entropy means it has committed. SAC (Haarnoja et al., 2018, ICML 2018) makes entropy maximization an explicit term in its objective, which is the mechanism that keeps it exploring while learning.

The *cross-entropy* between p and q is

$$H(p, q) = -\sum_x p(x) \log q(x) = H(p) + \mathrm{KL}(p \| q).$$

If p is fixed (as it is when p is the empirical distribution of training data), minimizing the cross-entropy in q is equivalent to minimizing $\mathrm{KL}(p\|q)$. This is the equivalence behind every "cross-entropy loss = log-likelihood loss" claim in §3.4: the empirical-data distribution plays the role of p, the model distribution plays the role of q, and Bayes' optimal predictor is the q that minimizes $H(p, q)$. Behavior cloning (§6.2) is exactly cross-entropy minimization between the demonstrator's action distribution and the policy's action distribution.

B.6 Likelihood, log-likelihood, and maximum likelihood

A *likelihood* is a probability density read as a function of the parameters θ that produced it, with the data held fixed: $\mathcal{L}(\theta; D) = p_\theta(D)$. The *log-likelihood* is its logarithm, $\ell(\theta; D) = \log p_\theta(D)$, and is the quantity actually optimized in practice — products turn into sums, underflow disappears, and the derivative is easier. The *maximum likelihood estimator* (MLE) is $\widehat{\theta} = \arg\max_\theta \ell(\theta; D)$.

Three uses, all already in the book by Chapter 12. The supervised loss in §3.4 is a negative log-likelihood. The reward of an MDP is sometimes inferred via maximum likelihood from demonstrations (maximum-entropy IRL is the classic example, treated lightly in §6.4). The training objective of every autoregressive language model that gets repurposed as a VLA backbone — Llama-2 for OpenVLA, PaLI-X for RT-2 — is the next-token log-likelihood of the training corpus.

Two known failure modes of MLE worth knowing about. First, MLE is *consistent* (converges to the truth as $N \to \infty$) but not necessarily efficient at finite N, and on small datasets a Bayesian or regularized estimator may dominate it. Second, MLE is *model-misspecification-sensitive*: if the chosen family p_θ does not include the data-generating distribution, MLE finds the θ that minimizes $\mathrm{KL}(p_{\text{data}}\|p_\theta)$, not the truth, and the asymmetry of KL means the result is the *support-covering* mode rather than the most likely one. This asymmetry is why mode collapse in GAN-style generators is a different failure than mode covering in MLE-trained density models.

B.7 The reparameterization trick and gradients of expectations

A recurring problem in ML is to compute gradients of an expectation $\mathbb{E}_{p_\theta}[f(X)]$ with respect to θ, the parameters of the distribution. Two strategies dominate. The *score- function estimator* (used in REINFORCE and policy gradient, §7.2) rewrites the gradient as

$$\nabla_\theta \mathbb{E}_{p_\theta}[f(X)] = \mathbb{E}_{p_\theta}[f(X)\, \nabla_\theta \log p_\theta(X)]\,.$$

This is correct in expectation, but has high variance and is the reason policy gradient methods need millions of environment steps.

The *reparameterization trick* applies when X can be written as a deterministic function of θ and an auxiliary noise variable $\epsilon \sim p(\epsilon)$: $X = g_\theta(\epsilon)$. Then

$$\nabla_\theta \mathbb{E}_{p_\theta}[f(X)] = \mathbb{E}_{p(\epsilon)}[\nabla_\theta f(g_\theta(\epsilon))]\,.$$

The gradient slides inside the expectation because ϵ does not depend on θ. For a Gaussian with parameters μ, σ, the reparameterization is $X = \mu + \sigma\,\epsilon$ with $\epsilon \sim \mathcal{N}(0, 1)$. This trick is what makes the VAE trainable in §3.4 and what makes the noise-injection in Diffusion Policy (§10.2) differentiable end-to-end. When you can reparameterize, you should; the variance reduction over the score-function estimator is dramatic.

B.8 The ELBO and variational inference

For a latent-variable model $p_\theta(x, z) = p_\theta(x \mid z)p(z)$, the log-likelihood of an observation x is

$$\log p_\theta(x) = \log \int p_\theta(x \mid z)p(z)\,dz,$$

an integral that is intractable in essentially every interesting case. The variational trick introduces an *approximate posterior* $q_\phi(z \mid x)$ and decomposes the log-likelihood as

$$\log p_\theta(x) = \underbrace{\mathbb{E}_{q_\phi}[\log p_\theta(x \mid z)] - \mathrm{KL}(q_\phi(z \mid x) \| p(z))}_{\mathrm{ELBO}(\theta, \phi; x)} + \mathrm{KL}(q_\phi(z \mid x) \| p_\theta(z \mid x)).$$

The second KL on the right is non-negative, so the ELBO (evidence lower bound) is a *lower bound* on the log-likelihood. Maximizing the ELBO simultaneously fits the model (θ) and tightens the variational approximation (ϕ). This is the entire machinery of the VAE (Kingma and Welling, 2014, arXiv:1312.6114) and the latent dynamics model in DreamerV3 (Hafner et al., 2023, arXiv:2301.04104) that Chapter 9 treats in detail.

B.9 Information-theoretic asides used in the book

A handful of additional information-theoretic notions appear in the body of the book and are worth a one-line gloss.

Mutual information $I(X; Y) = \mathrm{KL}(p(x, y) \| p(x)p(y))$ measures how much knowing one variable tells you about the other. Contrastive losses (CLIP, InfoNCE) are lower-bound estimators of mutual information.

Differential entropy of a continuous random variable extends the discrete entropy formula with an integral and a sign convention. A Gaussian with covariance Σ has differential entropy $\frac{1}{2}\log((2\pi e)^n |\Sigma|)$, which is why "maximum-entropy Gaussian prior" is the default well-behaved prior.

Jensen's inequality says that for any convex function ϕ and random variable X, $\phi(\mathbb{E}[X]) \le \mathbb{E}[\phi(X)]$ (reversed for concave ϕ). This is the inequality behind the ELBO derivation, behind every "lower-bound argument" in variational inference, and behind the proof that KL is non-negative.

B.10 A practitioner's checklist

Three reminders for the kind of bug that probability theory creates in practice. First, never let a probability go to zero in code; clamp densities away from zero by a tiny ϵ, or work in log-space throughout. Second, never use a `softmax(logits)` followed by `log`; use the numerically stable `log_softmax` your framework provides. Third, when you sample from a Gaussian for any reason — VAE, diffusion, policy noise — verify the covariance you are using is the one you mean: a 7-vector "with variance 0.1" might mean covariance $0.1 \cdot I$ or covariance $0.1^2 \cdot I = 0.01 \cdot I$, and the field uses both conventions inconsistently. Read the data loader; do not trust the paper.

That, plus the linear-algebra background of Appendix A and the PyTorch glue of Appendix C, is the math the rest of the book leans on.

Appendix C — PyTorch and JAX primer

Appendix C. PyTorch and JAX primer

The body of this book is implementation-shaped. The 50-line training loop of §3.3 is PyTorch, the OpenVLA checkpoint of Chapter 2 is PyTorch, the LoRA fine-tune of Chapter 16 is PyTorch. Two important parts of the modern stack are not: π0 (arXiv:2410.24164) and Octo (arXiv:2405.12213) ship as JAX/Flax models, and any reader who wants to do more than fine-tune a checkpoint will eventually have to read JAX code. This appendix is the cheat sheet for both frameworks. It assumes Python familiarity and skips installation; if `import torch` and `import jax` already work, you are in the right place.

C.1 Tensors and devices

Both frameworks revolve around a tensor type: a multidimensional array that can live on CPU, GPU, or TPU, and that participates in automatic differentiation. Three differences are worth knowing immediately.

PyTorch tensors are *mutable* and live on the device specified at creation:

```
import torch
x = torch.randn(3, 4, device="cuda")
x[0, 0] = 1.0   # in-place mutation; fine
```

JAX arrays are *immutable* and live on whichever device JAX chose by default (which you can override with `jax.device_put`):

```
import jax.numpy as jnp
x = jnp.array([[1.0, 2.0], [3.0, 4.0]])
# x[0, 0] = 1.0  -> TypeError: 'ArrayImpl' object does not support item assignment
x = x.at[0, 0].set(1.0)   # functional update; returns a new array
```

The `x.at[i].set(v)` pattern is JAX's answer to in-place writes, and it shows up everywhere — you will see it in optimizer step functions, in indexed updates to environment state, and in the body of every scan loop. It feels awkward for one afternoon and then disappears into the background.

The second difference: PyTorch dispatches operations *eagerly* by default, executing each one when its line of code runs. JAX traces operations *lazily*, building a computational graph that is compiled the first time you call the wrapping function. The third: PyTorch's default precision is fp32; JAX's default is fp32 too, but TPU work often defaults to bf16 because of hardware. Mixed-precision training in PyTorch goes through `torch.cuda.amp.autocast`; in JAX it goes through dtype-promotion rules and an explicit policy from `flax.linen`.

C.2 Autograd in two paragraphs

The single feature that makes either framework worth using is reverse- mode automatic differentiation. In PyTorch, every tensor remembers how it was produced if it has `requires_grad=True`, and calling `.backward()` on a scalar loss populates `.grad` attributes throughout the graph:

```
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = (x ** 2).sum()
y.backward()
print(x.grad)        # tensor([2., 4., 6.])
```

In JAX, gradients are produced by *function transformations*. The function `jax.grad(f)` returns a new function that computes the gradient of `f` with respect to its first argument; the original `f` is unchanged:

```
import jax
def loss(x):
    return (x ** 2).sum()
grad_fn = jax.grad(loss)
print(grad_fn(jnp.array([1.0, 2.0, 3.0])))   # [2. 4. 6.]
```

The functional style is what makes JAX composable: `jax.grad`, `jax.jit`, `jax.vmap`, and `jax.pmap` can be stacked in any order. The imperative style is what makes PyTorch feel like writing NumPy. In practice, large-scale VLA training favors PyTorch for its ecosystem and JAX for its TPU-native parallelism; the field has not converged.

C.3 A minimal PyTorch training loop

The training loop of §3.3 is the canonical template. The skeleton:

```
import torch
import torch.nn as nn
from torch.utils.data import DataLoader

model = MyPolicy().cuda()
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4)
loader = DataLoader(my_dataset, batch_size=64, shuffle=True, num_workers=4)

for epoch in range(num_epochs):
    for batch in loader:
        obs = batch["obs"].cuda(non_blocking=True)
        act = batch["act"].cuda(non_blocking=True)
        pred = model(obs)
        loss = nn.functional.mse_loss(pred, act)
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()
```

Six lines of substance, three lines of glue. The substance is: move batch to GPU, run forward, compute loss, zero gradients, backward, optimizer step. Every one of those steps has a JAX analogue, but the JAX version returns a new optimizer state and new parameters rather than mutating in place; see §C.5.

Four practical refinements you will see almost everywhere in the body of the book. *Gradient clipping*: `torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)` before `optimizer.step()`, to prevent occasional exploding gradients from destroying training. *Learning rate scheduling*: typically a linear warmup for the first 1000 steps followed by cosine decay, implemented via `torch.optim.lr_scheduler`. *Mixed precision*: wrap the forward in `with torch.cuda.amp.autocast(dtype=torch.bfloat16):` and use a `GradScaler` for fp16 (bf16 does not need a scaler). *Checkpoint saving*: `torch.save({"model": model.state_dict(), "opt": optimizer.state_dict(), "step": step}, path)` every N steps; checkpointing optimizer state is what lets you resume from preemption without re-warmup.

C.4 Modules, parameters, and state dicts

A PyTorch `nn.Module` is a class that bundles parameters, buffers, and a `forward` method. The canonical pattern:

```
class SmallPolicy(nn.Module):
    def __init__(self, obs_dim=20, act_dim=7, hidden=256):
        super().__init__()
        self.fc1 = nn.Linear(obs_dim, hidden)
        self.fc2 = nn.Linear(hidden, hidden)
        self.fc3 = nn.Linear(hidden, act_dim)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        return self.fc3(x)
```

`model.parameters()` returns an iterator over every leaf parameter the module owns; `model.state_dict()` returns an `OrderedDict` mapping parameter names to tensors. The two are how a checkpoint gets saved and loaded, and how a fine-tune restricts itself to a subset of parameters: filter `parameters()` by name and hand only the survivors to the optimizer. OpenVLA's LoRA fine-tune does exactly this — the 7B base parameters are frozen with `requires_grad = False`, the LoRA adapters are added with `requires_grad = True`, and only the adapters appear in the optimizer's parameter list.

A common confusion: `state_dict` includes both parameters (learned) and buffers (not learned but part of the model state — running mean and variance of batch-norm layers, for example). Saving only `parameters()` and not the buffers will silently produce a checkpoint that "loads cleanly" but evaluates wrong on subsequent runs.

C.5 The same loop in JAX/Flax

JAX has no notion of a stateful model. A model in Flax is a declaration of a function-with-parameters, and the parameters are ordinary PyTree leaves that you pass explicitly. The same small policy:

```
import flax.linen as nn
import jax
import jax.numpy as jnp
import optax

class SmallPolicy(nn.Module):
    hidden: int = 256
    act_dim: int = 7

    @nn.compact
    def __call__(self, x):
        x = nn.relu(nn.Dense(self.hidden)(x))
        x = nn.relu(nn.Dense(self.hidden)(x))
        return nn.Dense(self.act_dim)(x)

model = SmallPolicy()
params = model.init(jax.random.PRNGKey(0), jnp.zeros((1, 20)))
optimizer = optax.adamw(3e-4)
opt_state = optimizer.init(params)

@jax.jit
def update(params, opt_state, batch):
    def loss_fn(p):
        pred = model.apply(p, batch["obs"])
        return jnp.mean((pred - batch["act"]) ** 2)
    loss, grads = jax.value_and_grad(loss_fn)(params)
    updates, opt_state = optimizer.update(grads, opt_state, params)
```

```
    params = optax.apply_updates(params, updates)
    return params, opt_state, loss
```

Four things are different. The model's parameters live outside the model object, in a separate `params` PyTree. Forward passes use `model.apply(params, x)` rather than `model(x)`. The optimizer is a separate object whose state is also passed explicitly. The whole training step is wrapped in `jax.jit`, which compiles it to a single XLA program the first time it is called — slow on call number one, fast forever after.

The functional style buys two things. The same `update` function works on a single device and across multiple devices (with `jax.pmap` or `jax.shard_map`), without changing the implementation. And the behavior of `update` is completely determined by its arguments — there is no hidden state mutated in the background — which makes it straightforward to reason about, to checkpoint, and to debug.

C.6 Vectorization, parallelization, and the function-transform family

The transforms JAX exposes are worth listing as a group, because they are the single feature most often cited as the reason to choose JAX over PyTorch.

`jax.jit(fn)` compiles `fn` with XLA. Inputs of fixed shape and dtype are required at trace time, and Python-level control flow that depends on tensor values must be lifted to `jax.lax.cond` and `jax.lax.scan`.

`jax.grad(fn)` returns the gradient of `fn` with respect to its first argument. `jax.value_and_grad(fn)` returns both the value and the gradient in one pass.

`jax.vmap(fn, in_axes=...)` maps `fn` across a leading batch axis as if it were a Python `for` loop, but produces vectorized code. This is what eliminates the explicit batch dimension in many JAX models: write the function for a single example, then `vmap` it.

`jax.pmap(fn)` parallelizes `fn` across multiple devices, replicating the parameters and sharding the batch. Multi-host setups use `jax.lax.with_sharding_constraint` for finer-grained control.

PyTorch's analogues exist but are less unified. `torch.compile` is the JIT story; `torch.vmap` exists but is less integrated than its JAX counterpart; multi-device training is done through `DistributedDataParallel` or `FSDP`. The body of the book uses PyTorch's defaults for everything except the JAX-native models in Chapters 12 and 13.

C.7 Multi-GPU in one paragraph

Most VLA fine-tunes in this book fit on a single GPU. When they do not, the standard PyTorch idiom is `torch.nn.parallel.DistributedDataParallel` (DDP), which replicates the model on each GPU, shards the batch, and averages gradients via NCCL all-reduce. For models that exceed a single GPU's memory, *FSDP* (fully sharded data parallel) shards the parameters themselves across GPUs and reassembles them just in time for each forward and backward. The launch incantation is

```
torchrun --nproc-per-node=8 train.py
```

and inside the script, `init_process_group("nccl")` wires everything up. JAX's equivalent is `jax.distributed.initialize()` plus the function transforms above. Single-host multi-GPU is two lines of boilerplate in either framework; multi-host setups are more involved and the framework-specific docs are the right place to look.

C.8 Five pitfalls that have eaten a week of someone's time

A short, unranked list of failure modes that show up often enough to be worth printing.

The `.cuda()` call that does not happen. A model on GPU and data on CPU produces an error message that names the device mismatch explicitly; a model on GPU and a single tensor that secretly stayed on CPU silently runs much slower. Run `next(model.parameters()).device` to confirm.

The forgotten `optimizer.zero_grad()`. PyTorch accumulates gradients across `backward()` calls; if you forget to zero, the second batch's gradients are added to the first. Always call `zero_grad(set_to_none=True)` before each backward, or use the simpler `optimizer.zero_grad()` if you do not need the small memory savings.

The shape that broadcasts when you meant it to error. PyTorch's broadcasting rules are lenient: a `[B, D]` tensor and a `[D]` tensor combine without complaint. If you swap an axis upstream and the resulting shape still broadcasts, the model trains on garbage. Adding `assert x.shape == (B, D, T)` at module boundaries during development catches more bugs than it has any right to.

The data loader that pickles too much. Each `DataLoader` worker forks the parent process; if the parent has loaded a 7B-parameter model before the loader starts, each worker copies it. Construct the data loader before the model, or use `num_workers=0` during debug.

The JAX recompilation that fires on every step. `jax.jit` keys the compilation cache on the *shapes and dtypes* of the arguments, not their values. If any argument's shape changes per step — variable- length sequences, a batch that has one fewer example at the end of the epoch — JAX recompiles, and the second iteration of training suddenly takes thirty seconds. Pad to a fixed shape, or use `jax.lax.dynamic_slice` to handle variable lengths inside a fixed- shape buffer.

C.9 Reading π0's JAX code

The book's chapters use PyTorch by default, but Chapter 13's treatment of π0 (Black et al., 2024, arXiv:2410.24164) leans on the official JAX implementation. Three reading hints if you have not used JAX before.

The model class will be a `flax.linen.Module` (or, increasingly, `flax.nnx` for the newer codebases). Its parameters are not inside the class; they are produced once by `model.init(...)` and threaded through every subsequent call. When you see `model.apply(params, x)` in π0's training script, that is the JAX-native forward pass.

Anywhere you would have a Python `for` loop in PyTorch — over diffusion timesteps, over decoding tokens, over a recurrent unroll — JAX uses `jax.lax.scan` instead. `scan` looks like a fold: it carries a state forward through a fixed number of iterations and accumulates outputs along the way. Reading `scan`-based code feels foreign for a few minutes; the payoff is that the entire loop compiles to one XLA program.

Anywhere you would have a random number in PyTorch (via the global RNG), JAX takes an *explicit PRNG key* as an argument. The key is split via `jax.random.split(key)` into one key for each random operation, and the resulting code is deterministic in a way PyTorch's randomness is not. The π0 codebase passes PRNG keys through nearly every function call; the apparent boilerplate is what gives the model its bit-exact reproducibility.

C.10 A choice grid

When should you use which framework? Three rough heuristics.

If you are fine-tuning a published VLA, use whatever framework the authors released the checkpoint in. OpenVLA, RT-1, RT-2, RDT-1B, TinyVLA, SmolVLA — PyTorch. π0, Octo — JAX. Converting checkpoints between frameworks is possible but lossy in the kind of details (parameter naming, optimizer state) that matter for the second epoch.

If you are training a new model from scratch on a single 8-GPU node, PyTorch is the path of least resistance. The ecosystem (HuggingFace, PEFT, accelerate, deepspeed) is mature and the debugging story is better.

If you are training on TPUs or multi-host clusters of ≥ 64 GPUs, JAX is the path of least resistance. The compiled-by-default model and the unified parallelism story (`pmap`, `shard_map`) are worth the learning curve.

This is the framework chapter. The next one (Appendix D) is the simulator chapter — how to set up the MuJoCo or Isaac Lab instance on which the policies you just learned to train will actually be trained.

Appendix D — Setting up a robotics simulator

Appendix D. Setting up a robotics simulator

The chapters that ask you to run code — Chapter 2's LIBERO demo, Chapter 5's gridworld, Chapter 7's PPO on HalfCheetah, Chapter 16's fine-tuning loop — all assume a working simulator on your machine. This appendix is the step-by-step setup for the two simulators the body of the book uses: MuJoCo (for the small benchmarks and the Decision Transformer work) and Isaac Lab (for the large-scale, photo-realistic manipulation and locomotion work). It also covers attaching a USD asset so that you can extend an existing scene with your own robot, and the basic teleop loop so that you can record a small dataset for fine-tuning. The simulator landscape changes more often than the algorithms do, so where versions are specified, treat them as known- good at the time of writing and update against the official docs if those have moved.

D.1 Why two simulators, and not one

MuJoCo is small, fast, and CPU-friendly. A single MuJoCo environment runs at hundreds of frames per second on a laptop without a GPU, and the engine has been the backbone of academic RL benchmarks since Todorov, Erez, and Tassa (2012, IROS). LIBERO (Liu et al., 2023, NeurIPS) — the manipulation benchmark Chapter 2 loads — is a MuJoCo derivative, and so are D4RL, MetaWorld, and most of the published benchmarks that Chapters 5 through 10 reach for. If you only have a laptop, MuJoCo is the simulator you will use for the first three-quarters of the book.

Isaac Lab is large, photo-realistic, and GPU-native. Built on NVIDIA's Omniverse and Isaac Sim, it runs thousands of parallel environments on a single GPU at 60 Hz, with PhysX-backed rigid-body and contact dynamics and renderable RGB-D output. The trade-off is the install footprint (Omniverse alone is ~30 GB), a hard NVIDIA-GPU requirement, and a learning curve that is steeper than MuJoCo's by the better part of an afternoon. The chapters that fine-tune a VLA with sim-to-real (Chapter 16) and the chapters that run dexterous manipulation experiments (Chapters 13 and 15) need Isaac Lab; the earlier chapters do not.

If you can install only one, install MuJoCo first. The book's first runnable code lives inside it.

D.2 Installing MuJoCo

MuJoCo has been open source since 2022 and is pip-installable. On Ubuntu 22.04 and macOS 14+:

```
python3.10 -m venv ~/venvs/action-models
source ~/venvs/action-models/bin/activate
pip install --upgrade pip
pip install mujoco==3.2.5 dm-control gymnasium[mujoco] numpy
```

Confirm the install with the canonical one-liner:

```
import mujoco
import mujoco.viewer
model = mujoco.MjModel.from_xml_string("""
```

```
<mujoco>
  <worldbody>
    <body><geom type="box" size="0.1 0.1 0.1"/></body>
  </worldbody>
</mujoco>""")
data = mujoco.MjData(model)
mujoco.viewer.launch(model, data)
```

If a window opens with a small white cube sitting at the origin, MuJoCo is working. On macOS the viewer needs the `mjpython` launcher in place of `python` for the GUI to render correctly: `mjpython -m mujoco.viewer --mjcf=path/to/scene.xml`. On Ubuntu under WSL2, the viewer requires WSLg (Windows 11) or an X server forwarded to a Linux X11 client.

LIBERO sits on top of `robosuite`, which sits on top of MuJoCo. The install adds two more lines:

```
pip install robosuite
pip install git+https://github.com/Lifelong-Robot-Learning/LIBERO.git
```

A LIBERO benchmark task can then be loaded as:

```
from libero.libero import benchmark
bm = benchmark.get_benchmark_dict()["libero_object"]()
task = bm.get_task(0)        # the first of the ten "Object" tasks
env = task.env(render=False)
obs = env.reset()
```

A working LIBERO install is the prerequisite for the OpenVLA loop in Chapter 2; the chapter's hands-on directory has a `verify_libero.py` that exercises the loop end to end and prints the observation shape.

D.3 Installing Isaac Lab

Isaac Lab requires an NVIDIA GPU with at least 8 GB of memory (16 GB is recommended), driver version 535+, and Ubuntu 22.04 or Windows 11 with WSL2. macOS is not supported. The 2026 install path is through Isaac Sim 4.5 or later; the older "standalone Omniverse launcher" path is deprecated.

```
# 1. Install the NVIDIA driver and CUDA 12.x via Ubuntu's package manager.
sudo apt install nvidia-driver-535 nvidia-cuda-toolkit
# 2. Install Isaac Sim. The recommended method in 2026 is the pip
#    wheel; the older Omniverse Launcher GUI install is also supported.
pip install isaacsim==4.5.0 --extra-index-url https://pypi.nvidia.com
# 3. Install Isaac Lab from its GitHub repo.
git clone https://github.com/isaac-sim/IsaacLab.git ~/IsaacLab
cd ~/IsaacLab && ./isaaclab.sh --install
```

Confirm the install by running a canonical example:

```
cd ~/IsaacLab
./isaaclab.sh -p source/standalone/demos/quadrupeds.py
```

If a window opens with a herd of small quadruped robots running on a plane, Isaac Lab is working. First-launch compilation takes a few minutes; subsequent launches are fast.

Two common install failures. *Driver mismatch*: `nvidia-smi` works but Isaac Sim crashes on launch with a kernel error. The fix is almost always to update the driver to 535+ and reboot. *X-server mismatch*: the GUI launches but renders a black window. This is a WSLg or VirtualGL issue on remote machines; the workaround is to run with `--headless` and use the recording-and-playback tools described in §D.5.

D.4 Loading a USD asset

Both Isaac Sim and Isaac Lab use the Universal Scene Description (USD) format pioneered by Pixar. A USD file (`.usd`, `.usda`, or `.usdc`) describes a scene as a hierarchical tree of "prims" — geometric primitives, references, materials, lights — and is the format every NVIDIA robotics asset ships in. Most academic robotics assets ship in URDF (XML-flavored) or MJCF (also XML-flavored, the MuJoCo dialect); both need to be converted to USD before Isaac Lab will accept them.

The conversion utility:

```
./isaaclab.sh -p source/standalone/tools/convert_urdf.py \
    --input  path/to/my_robot.urdf \
    --output path/to/my_robot.usd \
    --merge-joints
```

After conversion, the asset can be loaded into a scene as:

```
import isaaclab.sim as sim_utils
from isaaclab.assets import ArticulationCfg

my_robot_cfg = ArticulationCfg(
    prim_path="/World/MyRobot",
    spawn=sim_utils.UsdFileCfg(usd_path="path/to/my_robot.usd"),
    init_state=ArticulationCfg.InitialStateCfg(
        pos=(0.0, 0.0, 0.5),
        joint_pos={"joint_0": 0.0, "joint_1": -0.5},
    ),
)
```

Three things commonly go wrong on the first conversion. URDF inertia tags are sometimes wrong by a factor of 10 (a 1-kg link claiming 10 kg·m² of inertia); the simulator will solve but the dynamics will feel wrong, and PhysX may flag the asset as unstable. Mesh collisions default to the visual mesh, which is usually too detailed to simulate efficiently; pre-compute a convex decomposition (the `--collision-approximation convexDecomposition` flag) and Isaac Lab will use that for contact. Joint limits in radians vs. degrees: PhysX expects radians, URDF allows either, and a 90 in radians is a joint that flies out the floor on the first step.

D.5 Recording a teleop dataset

The fine-tuning recipe of Chapter 16 needs demonstrations. The canonical way to produce them is teleoperation: a human operator drives the robot through the task, the simulator records (observation, action, reward, done) tuples, and the result is a dataset of episodes ready for behavior cloning. Both MuJoCo and Isaac Lab support keyboard and 3D-mouse teleop out of the box; SpaceMouse hardware (the 3Dconnexion family) is the standard.

In MuJoCo/robosuite, a basic teleop loop:

```
import robosuite as suite
from robosuite.devices import SpaceMouse

env = suite.make("Lift", robots="Panda", has_renderer=True,
                 has_offscreen_renderer=False, control_freq=20)
device = SpaceMouse()
device.start_control()

episodes = []
for ep in range(10):
    obs = env.reset()
    transitions = []
    done = False
```

```
    while not done:
        action = device.get_controller_state()["dpos_drot"]
        next_obs, reward, done, info = env.step(action)
        transitions.append((obs, action, reward, next_obs, done))
        env.render()
        obs = next_obs
    episodes.append(transitions)
```

Saving the result to an HDF5 file in the format the rest of the book expects (one group per episode, datasets `observations`, `actions`, `rewards`, `dones`) is twenty more lines of glue; the chapter's hands- on directory has the working version.

In Isaac Lab, teleop goes through the `Teleop` workflow in `isaaclab.envs`; the API is busier but the principle is the same. The canonical example to crib from is `source/standalone/workflows/teleoperation/teleop_se3_agent.py`.

Three pragmatic notes about teleop datasets. *Frequency*: 20 Hz is the standard for end-effector-pose teleop; 50 Hz for joint-velocity teleop. Sampling faster than the human can react produces redundant frames; sampling slower produces jerky trajectories. *Reset discipline*: the operator should re-randomize the scene between episodes (block position, lighting, target pose), otherwise the dataset over-represents one initial condition. *First-episode junk*: the first one or two episodes of any teleop session contain the operator learning the controls; throw them out before training.

D.6 Headless rendering and recording

Most training runs do not need a viewer. To run a MuJoCo environment without rendering at all, just construct it without the viewer call and step forward; the engine runs orders of magnitude faster than real time on a CPU. To run Isaac Lab without rendering, append `--headless` to the launch command:

```
./isaaclab.sh -p source/standalone/tutorials/00_sim/spawn_prims.py --headless
```

Headless runs still produce camera observations via the `isaaclab.sensors.Camera` API; the cameras render to GPU buffers that the policy reads directly, no window required. Recording a video from a headless run uses `imageio`:

```
import imageio
frames = []
for step in range(200):
    obs = env.step(action)
    frames.append(obs["camera"]["rgb"])
imageio.mimsave("rollout.mp4", frames, fps=20)
```

Videos are how you debug a policy that "works in success rate" but "feels weird"; the eyeball is the highest-bandwidth diagnostic instrument in robot learning.

D.7 A platform-specific cheat sheet

A condensed reference for the install and launch commands by operating system.

Ubuntu 22.04 with an NVIDIA GPU: MuJoCo via pip, Isaac Lab via the two-command install above, both run natively. The most common failure is a stale CUDA driver; `sudo apt upgrade nvidia-driver-535 && reboot` fixes most things.

Ubuntu 22.04 without an NVIDIA GPU: MuJoCo via pip; Isaac Lab is not supported. Use Gymnasium environments for the earlier chapters and reach for a cloud GPU only when Chapter 16 demands it.

macOS (Apple Silicon, M-series): MuJoCo via pip works out of the box, with the `mjpython` launcher for the GUI; LIBERO works with the same caveat. Isaac Lab is unsupported; use a Linux machine or a cloud GPU instance.

Windows 11 with WSL2: MuJoCo works under WSL2 with WSLg. Isaac Lab requires both WSLg and a recent NVIDIA driver; the install is otherwise identical to native Ubuntu.

Cloud (a single A100 or L40S on RunPod, Lambda, or vast.ai): MuJoCo and Isaac Lab both run unmodified; the install is identical to a native Ubuntu install. The hassle is filesystem persistence — preemptible instances erase state on every shutdown — so put checkpoints on attached object storage and not on the instance's local disk.

D.8 What to do when the simulator misbehaves

Three failure modes that every reader will encounter at least once.

The physics blew up. The robot's joints fly out the floor at simulation step 0. The cause is almost always an asset issue — bad inertia tags, a self-colliding mesh, joint limits in the wrong units. Run the asset's `analyze` step first (`mujoco._analyze` or Isaac Lab's USD validator) and fix what it complains about.

The policy outputs values the simulator clips. The model emits an action of magnitude 50 but the simulator's joint torque limit is 10. The clipping is happening silently and the model never gets a useful gradient. Add an `assert action.abs().max() < limit` somewhere in the data loader and confirm the bound is what you think it is.

The episode is too long or too short. The default episode length of a benchmark may be 200 steps; your task takes 500. The model learns a policy that "finishes" at step 199 by giving up. Reading the environment config and confirming `max_episode_steps` against your task's actual duration is a two-minute check that saves a half-day of confusion.

A working simulator is the unsexy prerequisite for everything else in this book. With MuJoCo running for the small benchmarks and Isaac Lab running for the large ones, every code listing in Chapters 2 through 17 should produce the result the chapter describes. The appendix that comes next (Appendix E) is the consolidated reading list; the appendix after that (F) is the model zoo table.

Appendix E — Canonical references (full bibliography, by chapter)

Appendix E. Canonical references and the offline source mirror

Each chapter cites its primary references in the chapter body. This appendix consolidates them into a single reading list organized by chapter and documents the offline source mirror that ships with the project's working directory so citations can be verified without an internet connection.

E.1 How to use the offline source mirror

All canonical arXiv PDFs cited in the book are mirrored under `sources/pdfs/` in the project's working directory, with filenames of the form `arxiv_<id>_<short-title>.pdf` — for example, `arxiv_2406.09246_OpenVLA.pdf`. Companion blog posts, vendor technical reports, and product pages that lack arXiv equivalents are saved as HTML snapshots under `sources/html/` with descriptive filenames matching their domain — for example, `huggingface_blog_pi0_and_pi0_fast.html` or `figure_ai_news_helix.html`. The machine-readable index `sources/SOURCES_INDEX.txt` lists every URL ⟶ local-file mapping, plus the curl-based `download_sources.sh` script used to (re)generate the mirror.

Four citation-hygiene rules apply throughout the book.

Prefer the arXiv ID (for example, arXiv:2406.09246) over a hosted URL. arXiv IDs are stable; URLs are not.

When a model is best described by a vendor blog or technical report rather than a peer-reviewed paper — Helix, Helix 02, RoboBrain2.0, ρα (Rho-alpha) — the canonical URL is given and the snapshot date is recorded in `SOURCES_INDEX.txt`.

Survey citations are kept up to date in each chapter's reference list and in §E.2 of this appendix. A reader who wants a single point of entry to the literature starts with the surveys listed under Chapter 1.

Numerical claims — parameter counts, training-hours figures, success rates — are always cited to the model's primary reference. The Model Zoo in Appendix F reuses those numbers verbatim and points back to the primary-reference column rather than re-citing the originals.

E.2 Reading list, by chapter

Chapter 1. The robot learning problem

- Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement Learning in Robotics: A Survey. *International Journal of Robotics Research*, 32(11), 1238–1274.
- Sapkota, R. et al. (2025). Vision-Language-Action Models: Concepts, Progress, Applications and Challenges. arXiv:2505.04769.
- Zhang et al. (2025). Pure Vision Language Action (VLA) Models — A Comprehensive Survey. arXiv:2509.19012.
- Ni et al. (2025). Embodied Arena — A Comprehensive, Unified, and Evolving Evaluation Platform. arXiv:2509.15273.

- Xu et al. (2025). An Anatomy of Vision-Language-Action Models — From Modules to Milestones and Challenges. arXiv:2512.11362.

Chapter 2. Your first VLA, end-to-end

- Kim, M. J. et al. (2024). OpenVLA — An Open-Source Vision-Language- Action Model. arXiv:2406.09246.
- Liu, B. et al. (2023). LIBERO — Benchmarking Knowledge Transfer for Lifelong Robot Learning. *NeurIPS 2023 Datasets and Benchmarks track*.
- Garcia et al. (2025). LIBERO-Para — A Diagnostic Benchmark and Metrics for Paraphrase Robustness in VLA Models. arXiv:2603.28301.

Chapter 3. Math and ML prerequisites in 30 minutes

- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
- Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.
- Paszke, A. et al. (2019). PyTorch — An Imperative Style, High- Performance Deep Learning Library. *NeurIPS 2019*.

Chapter 4. Classical action models: planning and inverse dynamics

- Fikes, R. E., & Nilsson, N. J. (1971). STRIPS — A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence*, 2(3–4), 189–208.
- McDermott, D. et al. (1998). PDDL — The Planning Domain Definition Language. *Yale Center for Computational Vision and Control TR-98-003*.
- Helmert, M. (2006). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26, 191–246.
- Garrett, C. R., Lozano-Pérez, T., & Kaelbling, L. P. (2020). PDDLStream — Integrating Symbolic Planners and Blackbox Samplers via Optimistic Adaptive Planning. *ICAPS 2020*.
- Ahn, M. et al. (2022). Do As I Can, Not As I Say — Grounding Language in Robotic Affordances (SayCan). arXiv:2204.01691.
- LaValle, S. M. (2006). *Planning Algorithms*. Cambridge University Press.
- Featherstone, R. (2008). *Rigid Body Dynamics Algorithms*. Springer.
- Khatib, O. (1987). A Unified Approach for Motion and Force Control of Robot Manipulators — The Operational Space Formulation. *IEEE Journal of Robotics and Automation*, 3(1).
- Hogan, N. (1985). Impedance Control — An Approach to Manipulation. *ASME Journal of Dynamic Systems, Measurement, and Control*.

Chapter 5. Learning from rewards: MDPs and reinforcement learning

- Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.
- Howard, R. A. (1960). *Dynamic Programming and Markov Processes*. MIT Press.
- Puterman, M. L. (1994). *Markov Decision Processes — Discrete Stochastic Dynamic Programming*. Wiley.
- Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning — An Introduction* (2nd ed.). MIT Press.
- Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement Learning in Robotics — A Survey. *International Journal of Robotics Research*, 32(11).
- Kaelbling, L. P., Littman, M. L., & Cassandra, A. R. (1998). Planning and Acting in Partially Observable Stochastic Domains. *Artificial Intelligence*, 101(1–2).

Chapter 6. Learning from demonstrations: behavior cloning and imitation learning

- Pomerleau, D. A. (1988). ALVINN — An Autonomous Land Vehicle in a Neural Network. *NeurIPS 1988*.
- Ross, S., Gordon, G. J., & Bagnell, J. A. (2011). A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning (DAgger). *AISTATS 2011*.
- Ng, A. Y., & Russell, S. J. (2000). Algorithms for Inverse Reinforcement Learning. *ICML 2000*.
- Argall, B. D., Chernova, S., Veloso, M., & Browning, B. (2009). A Survey of Robot Learning from Demonstration. *Robotics and Autonomous Systems*, 57(5).

Chapter 7. Deep RL for control: DQN to SAC and PPO

- Mnih, V. et al. (2015). Human-level Control through Deep Reinforcement Learning (DQN). *Nature*, 518(7540).
- Lillicrap, T. P. et al. (2016). Continuous Control with Deep Reinforcement Learning (DDPG). *ICLR 2016*.
- Schulman, J. et al. (2017). Proximal Policy Optimization Algorithms (PPO). arXiv:1707.06347.
- Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods (TD3). *ICML 2018*.
- Haarnoja, T. et al. (2018). Soft Actor-Critic. *ICML 2018*.
- Tobin, J. et al. (2017). Domain Randomization for Transferring Deep Neural Networks from Simulation to the Real World. *IROS 2017*.

Chapter 8. Sequence models meet control

- Vaswani, A. et al. (2017). Attention Is All You Need. *NeurIPS 2017*.
- Chen, L. et al. (2021). Decision Transformer — Reinforcement Learning via Sequence Modeling. *NeurIPS 2021*.
- Janner, M., Li, Q., & Levine, S. (2021). Offline Reinforcement Learning as One Big Sequence Modeling Problem (Trajectory Transformer). *NeurIPS 2021*.
- Fu, J. et al. (2020). D4RL — Datasets for Deep Data-Driven Reinforcement Learning. arXiv:2004.07219.
- Liu et al. (2024). RoboMamba — Multimodal State Space Model for Efficient Robot Reasoning and Manipulation. arXiv:2406.04339.
- Wu et al. (2025). A Survey on Efficient Vision-Language-Action Models. arXiv:2510.24795.

Chapter 9. World models and model-based learning

- Ha, D., & Schmidhuber, J. (2018). World Models. *NeurIPS 2018*.
- Hafner, D. et al. (2019). Learning Latent Dynamics for Planning from Pixels (PlaNet). *ICML 2019*.
- Hafner, D., Lillicrap, T., Ba, J., & Norouzi, M. (2020). Dream to Control — Learning Behaviors by Latent Imagination (Dreamer). *ICLR 2020*.
- Hafner, D. et al. (2023). Mastering Diverse Domains through World Models (DreamerV3). arXiv:2301.04104.
- Bruce, J. et al. (2024). Genie — Generative Interactive Environments. *ICML 2024*.
- LeCun, Y. (2022). A Path Towards Autonomous Machine Intelligence. OpenReview position paper.

Chapter 10. Diffusion and flow models for action generation

- Ho, J., Jain, A., & Abbeel, P. (2020). Denoising Diffusion Probabilistic Models. *NeurIPS 2020*.
- Song, Y. et al. (2021). Score-Based Generative Modeling through Stochastic Differential Equations. *ICLR 2021*.
- Chi, C. et al. (2023). Diffusion Policy — Visuomotor Policy Learning via Action Diffusion. *RSS 2023*.
- Zhao, T. Z., Kumar, V., Levine, S., & Finn, C. (2023). Learning Fine-Grained Bimanual Manipulation with Low-Cost Hardware (ACT). *RSS 2023*.
- Lipman, Y. et al. (2023). Flow Matching for Generative Modeling. *ICLR 2023*.
- Liu, X., Gong, C., & Liu, Q. (2023). Flow Straight and Fast — Learning to Generate and Transfer Data with Rectified Flow. *ICLR 2023*.

Chapter 11. The VLA recipe: from CLIP to RT-1

- Radford, A. et al. (2021). Learning Transferable Visual Models from Natural Language Supervision (CLIP). *ICML 2021*.
- Jang, E. et al. (2022). BC-Z — Zero-Shot Task Generalization with Robotic Imitation Learning. *CoRL 2021*.
- Brohan, A. et al. (2022). RT-1 — Robotics Transformer for Real-World Control at Scale. arXiv:2212.06817.
- Reed, S. et al. (2022). A Generalist Agent (Gato). *TMLR*.

Chapter 12. Scaling up: PaLM-E, RT-2, OpenVLA, Octo (and friends)

- Driess, D. et al. (2023). PaLM-E — An Embodied Multimodal Language Model. arXiv:2303.03378.
- Brohan, A. et al. (2023). RT-2 — Vision-Language-Action Models Transfer Web Knowledge to Robotic Control. arXiv:2307.15818.

- Kim, M. J. et al. (2024). OpenVLA. arXiv:2406.09246.
- Octo Model Team / Ghosh, D. et al. (2024). Octo — An Open-Source Generalist Robot Policy. arXiv:2405.12213.
- O'Neill, A. et al. (2024). Open X-Embodiment — Robotic Learning Datasets and RT-X Models. arXiv:2310.08864.
- Liu et al. (2024). RDT-1B — A Diffusion Foundation Model for Bimanual Manipulation. arXiv:2410.07864.

Chapter 13. Smooth control: π0 and flow-matching action heads

- Black, K. et al. (2024). π0 — A Vision-Language-Action Flow Model for General Robot Control. arXiv:2410.24164 (companion technical report at pi.website/download/pi0.pdf).
- Pertsch, K. et al. (2025). FAST — Efficient Action Tokenization for Vision-Language-Action Models. arXiv:2501.09747.
- Frontier Robotics (2025). SimVLA — A Simple VLA Baseline for Robotic Manipulation. arXiv:2602.18224.
- Xiaomi Robotics (2025). Xiaomi-Robotics-0. arXiv:2602.12684.
- Hugging Face (2025). SmolVLA. arXiv:2506.01844.

Chapter 14. Dual-system architectures: Helix and GR00T N1

- Figure AI (2024). Helix — A Vision-Language-Action Model for Generalist Humanoid Control. figure.ai/news/helix.
- Figure AI (2026). Helix 02 — A General-Purpose Humanoid System. figure.ai/news/helix-02.
- NVIDIA (2025). GR00T N1 — An Open Foundation Model for Generalist Humanoid Robots. arXiv:2503.14734.
- Driess, D. et al. (2023). PaLM-E (antecedent of VLM + controller decoupling). arXiv:2303.03378.
- Ahn, M. et al. (2022). Do As I Can, Not As I Say (SayCan). *CoRL 2022*.

Chapter 15. Datasets, benchmarks, and evaluation; specialized and adjacent VLAs

- Wei et al. (2025). LiLo-VLA — Compositional Long-Horizon Manipulation via Linked Object-Centric Policies. arXiv:2602.21531.
- Long-VLA Team (2025). Unleashing Long-Horizon Capability of Vision- Language Models for Manipulation (Long-VLA). arXiv:2508.19958.
- Tianjin University / Huawei (2025). Embodied-R1 — Reinforced Embodied Reasoning for Manipulation. arXiv:2508.13998.
- Huang et al. (2024). LEO — An Embodied Generalist Agent in 3D World. arXiv:2311.12871.
- OpenDriveVLA Team (2025). OpenDriveVLA — Towards End-to-End Autonomous Driving with VLAs. arXiv:2503.23463.
- Zheng et al. (2025). UniAct — A Universal Action Model for Cross- Embodiment Robot Manipulation. arXiv:2501.10105.
- TinyVLA Team (2024). TinyVLA — Toward Fast, Data-Efficient Vision- Language-Action Models for Robotic Manipulation. arXiv:2409.12514.

Chapter 16. Fine-tuning a VLA for your robot

- Hu, E. J. et al. (2022). LoRA — Low-Rank Adaptation of Large Language Models. *ICLR 2022*.
- Kim, M. J. et al. (2024). OpenVLA. arXiv:2406.09246.
- Octo Model Team / Ghosh, D. et al. (2024). Octo. arXiv:2405.12213.
- Wu et al. (2025). A Survey on Efficient Vision-Language-Action Models. arXiv:2510.24795.
- TinyVLA Team (2024). TinyVLA. arXiv:2409.12514.
- Hugging Face (2025). SmolVLA. arXiv:2506.01844.
- Liu et al. (2024). RoboMamba. arXiv:2406.04339.

Chapter 17. Evaluation, safety, and deployment

- García, J., & Fernández, F. (2015). A Comprehensive Survey on Safe Reinforcement Learning. *Journal of Machine Learning Research*, 16.
- Alshiekh, M. et al. (2018). Safe Reinforcement Learning via Shielding. *AAAI 2018*.

- Brunke, L. et al. (2022). Safe Learning in Robotics — From Learning- Based Control to Safe Reinforcement Learning. *Annual Review of Control, Robotics, and Autonomous Systems*, 5.
- Amodei, D. et al. (2016). Concrete Problems in AI Safety. arXiv:1606.06565.
- Garcia et al. (2025). LIBERO-Para. arXiv:2603.28301.
- Ni et al. (2025). Embodied Arena. arXiv:2509.15273.

Chapter 18. Open problems and what comes next

- Zhang et al. (2025). Pure Vision Language Action (VLA) Models — A Comprehensive Survey. arXiv:2509.19012.
- Sapkota, R. et al. (2025). Vision-Language-Action Models — Concepts, Progress, Applications and Challenges. arXiv:2505.04769.
- Xu et al. (2025). An Anatomy of Vision-Language-Action Models — From Modules to Milestones and Challenges. arXiv:2512.11362.
- Wu et al. (2025). A Survey on Efficient Vision-Language-Action Models. arXiv:2510.24795.
- Stanford HAI (2025). *AI Index Report 2025* — Robotics chapter.
- Firoozi, R. et al. (2023). Foundation Models in Robotics — Applications, Challenges, and the Future. arXiv:2312.07843.

E.3 Where the mirror is incomplete

Three categories of citation in this book are not held in the offline mirror. *Textbooks* — Sutton and Barto, Bishop, Strang, LaValle, Featherstone, Murray-Li-Sastry — are commercial works and not mirrored; the reader is expected to either own a copy or to access them through institutional channels. *Older pre-arXiv papers* — Fikes and Nilsson (1971), Pomerleau (1988), Khatib (1987), Hogan (1985), McDermott et al. (1998) — predate arXiv and are cited by venue; canonical scans exist on author homepages and university repositories and are linked to in `SOURCES_INDEX.txt` where stable URLs were available at snapshot time. *Vendor technical reports* — the Figure AI Helix and Helix 02 posts, the BAAI RoboBrain2.0 report, the Microsoft ρα report — are HTML pages that have been snapshotted to `sources/html/` with the date of capture recorded; if the live version has since changed, the snapshot is the canonical reference for what is claimed in the book.

A reader who finds a missing reference is asked to open an issue on the book's companion repository so the next edition's mirror script can pick it up.

Appendix F — VLA model zoo

Appendix F. Model zoo

This appendix is a reference table of the 24 vision-language-action and related foundation action models discussed in the book. Each row summarizes the developer, the backbone architecture, the parameter count, a one-line distinguishing feature, and the primary reference. The "Chapter" column points to the chapter where each model is treated in depth; many models appear briefly in adjacent chapters as well.

Numerical entries are taken from each model's primary reference, as cited in Appendix E.2 and in the relevant chapter bodies. A dash (—) indicates the primary reference does not report that figure, or reports a range that does not fit in a single cell; consult the chapter for the qualified version.

F.1 The table

Model	Developer	Backbone	Params	Distinguishing feature	Primary ref	Chapter
PaLM-E	Google DeepMind	PaLM + ViT-22B	up to 562B	Injects continuous sensors into LLM embedding space; embodied multimodal LLM.	arXiv:2303.03378	Ch. 12
RT-1	Google DeepMind	Robotics Transformer (decoder-only)	—	FiLM multimodal fusion; image and action tokenization; 130k demos across 700+ tasks.	arXiv:2212.06817	Ch. 11
RT-2	Google DeepMind	PaLI-X / PaLM-E	5B–55B	Co-fine-tuned on web + robotics data; actions as discrete text tokens; chain-of-thought reasoning.	arXiv:2307.15818	Ch. 12

Model	Developer	Backbone	Params	Distinguishing feature	Primary ref	Chapter
RT-2-X	22-institution collaboration	—	—	Multi-embodiment generalization across 20+ robot hardware types.	arXiv:2310.08864	Ch. 12, 15
OpenVLA	Stanford / Berkeley / TRI / DeepMind	Llama-2 7B + SigLIP + DinoV2	7B	Open-source generalist VLA; LoRA touches only 1.4% of parameters; supports quantization.	arXiv:2406.09246	Ch. 2, 12, 16
Octo	Berkeley / Stanford / CMU / DeepMind	ViT-S/B transformer	27M–93M	Block-wise attention to add or remove sensory inputs; open source; reward-free imitation.	arXiv:2405.12213	Ch. 12, 16
RDT-1B	Tsinghua	Diffusion Transformer	1B	Large-scale diffusion model for bimanual manipulation; zero-shot generalization.	arXiv:2410.07864	Ch. 13
$\pi 0$	Physical Intelligence	PaliGemma + flow-matching action expert	3.3B	Continuous actions at 50 Hz; ~10,000 hours of robotic trajectories; dexterous multi-stage tasks.	arXiv:2410.24164	Ch. 13
$\pi 0$-FAST	Physical Intelligence	Autoregressive $\pi 0$ + FAST tokenizer	—	Frequency-space (DCT) action-sequence tokenization; ~5× faster training.	arXiv:2501.09747	Ch. 13

Model	Developer	Backbone	Params	Distinguishing feature	Primary ref	Chapter
SimVLA	Frontier Robotics	Standard VLM + lightweight action transformer	0.5B–0.8B	Decoupled perception and control; flow-matching denoising; on par with $\pi 0.5$ on LIBERO.	arXiv:2602.18224	Ch. 13
Xiaomi-Robotics-0	Xiaomi Robotics	Qwen3-VL-4B + flow-matching DiT	4.7B	Real-time execution; flow-matching action expert.	arXiv:2602.12684	Ch. 13
SmolVLA	Hugging Face	SmolVLM2 encoder-decoder + flow-matching transformer	450M	Consumer-GPU and MacBook deployable; asyn-chronous inference decouples VLM from execution.	arXiv:2506.01844	Ch. 13, 16
TinyVLA	Midea / academic collab.	Lightweight pretrained VLM (<1.4B) + diffusion decoder	<1.4B	Diffusion-based policy decoder; distilled from larger VLAs; fast, data-efficient inference.	arXiv:2409.12514	Ch. 16
RoboMamba	Tongji University	Mamba SSM + frozen CLIP	—	Linear-scaling SSM backbone replaces transformer; 3.7M-parameter MLP policy head.	arXiv:2406.04339	Ch. 8, 16
Helix / Helix 02	Figure AI	Dual-system (7B VLM + 80M visuomotor)	7.08B	Full upper-body 35-DoF continuous control; on-board embedded GPUs; “Sport Mode”.	figure.ai/news/helix and helix-02	Ch. 14

Model	Developer	Backbone	Params	Distinguishing feature	Primary ref	Chapter
GR00T N1	NVIDIA	Dual-system (System 2 VLM + System 1 reactive)	2.2B	Humanoid-centric; 93.3% language-following; 3000h+ of human video + robot + synthetic data.	arXiv:2503.14734	Ch. 14
ρα (Rho-alpha)	Microsoft Research	Phi family + action expert	—	Tactile sensing + online learning from human corrections; BusyBox benchmark.	MSR technical report	Ch. 14, 17
Embodied-R1	Tianjin University / Huawei Noah's Ark	Qwen-2.5-VL-3B-Instruct	3B	Reinforced fine-tuning (RFT); multi-task reward curriculum; affordance prediction.	arXiv:2508.13998	Ch. 15
RoboBrain2.0	BAAI	Qwen2.5-VL-72B-Instruct (fine-tuned)	7B–32B	Reinforced embodied reasoning; spatial-referring data; pointing fine-tuning.	BAAI technical report	Ch. 12
LiLo-VLA	UNC / Georgia Tech / CMU	OpenVLA-OFT or π0.5 backbone	—	Modular: decouples transport (reaching) from interaction; object-centric visual masking.	arXiv:2602.21531	Ch. 15
Long-VLA	Westlake / Zhejiang / Xi'an Jiaotong	MDT + GPT-2-style transformer	—	Phase-aware input masking; long-horizon skill chaining; L-CALVIN benchmark.	arXiv:2508.19958	Ch. 15

Model	Developer	Backbone	Params	Distinguishing feature	Primary ref	Chapter
LEO	BAAI	Transformer-based 3D model	—	Two-stage 3D vision–language alignment; embodied 3D scene understanding.	arXiv:2311.12871	Ch. 15
UniAct	Tsinghua / Shanghai AI Lab	Autoregressive transformer	—	Universal atomic actions for cross-embodiment heterogeneity.	arXiv:2501.10105	Ch. 15, 18
OpenDriveVLA	—	Autoregressive trajectory generator	—	Unified 2D / 3D perception → driving trajectories; closed-loop end-to-end control.	arXiv:2503.23463	Ch. 15

F.2 How to read the zoo

The table is laid out so that consecutive rows roughly correspond to the order in which the models are introduced in the book. Rows 1–4 (PaLM-E through RT-2-X) are the Google DeepMind lineage covered in Chapters 11–12. Rows 5–6 (OpenVLA, Octo) are the open-source generalist policies treated in Chapter 12 and revisited in Chapter 16. Rows 7–13 (RDT-1B through RoboMamba) are the continuous- action and efficient-VLA family of Chapter 13 and §16.x. Rows 14–17 (Helix through ρα) are the dual-system models of Chapter 14. Rows 18–24 (Embodied-R1 through OpenDriveVLA) are the specialized and adjacent models — long-horizon, 3D, cross-embodiment, driving — covered in Chapter 15.

Three caveats on the parameter-count column. The number reported is the total parameter count of the deployed model unless otherwise noted; for dual-system models the column reports the sum of the slow (VLM) and fast (sensorimotor) components and the breakdown is given in the corresponding chapter. For models that ship in multiple sizes (RT-2 at 5B and 55B, Octo at 27M and 93M, RoboBrain2.0 at 7B and 32B), the column reports the range; the chapter treatment uses the size the authors emphasize. For models that report only an active parameter count or a sparse-expert total, the chapter body explains the convention used; the zoo cell is the total.

F.3 What the zoo deliberately omits

A few well-known systems are absent from the table by design. Standalone vision-language models (Llama-3 family, Qwen-VL, PaLI-X) are excluded — they are backbones for VLAs in this table, not VLAs themselves. Pure robotics policies without language conditioning (Diffusion Policy, ACT) are excluded — they appear in Chapter 10 as *architectures* rather than as *foundation models*, and the distinction matters: a Diffusion Policy trained on one task is not a foundation model, but it is part of the lineage that produced the action heads of π0 and RDT-1B. Pure planners with LLM "brains" (SayCan, Code as Policies, Inner Monologue) are excluded — they emit symbolic plans rather than low-level actions, and so technically belong to the lineage of Chapter 4 rather than the zoo of Part 4.

The boundary between "foundation action model" and "large policy" is not crisp; the inclusion criterion used here is that the model (i) ingests both vision and language and emits actions, (ii) has been pretrained or co-trained on a

corpus that includes general web-scale content rather than only robotics data, and (iii) is presented by its authors as a step toward generality rather than as a single-task demonstration. This is the same criterion the surveys cited in Chapter 1 use, and disagreement at the boundary (Octo vs. RT-2; LEO vs. PaLM-E) is the kind of disagreement that the introductory chapter of each survey spends a section on.

F.4 Cross-references

A small number of cross-cutting facts may be useful when comparing rows.

The single largest parameter count in the table belongs to PaLM-E (up to 562B); the single smallest is SmolVLA (450M). The ratio is about 1,250×, which is the dynamic range the field currently operates over.

The four models that use a flow-matching action head — π0, SimVLA, Xiaomi-Robotics-0, SmolVLA — share the architectural lineage laid out in Chapter 10 and revisited in Chapter 13. The two that use a diffusion-transformer head — RDT-1B and TinyVLA — share a different lineage; the contrast is the substance of §10.5.

The two dual-system models — Helix and GR00T N1 — share the System- 1 / System-2 architectural pattern but disagree on the latency budget assigned to each system; Chapter 14's §14.4 has the breakdown.

The open-source weights, in order of permissiveness, are: OpenVLA (Apache 2.0), Octo (MIT), RDT-1B (Apache 2.0), SmolVLA (Apache 2.0), TinyVLA (research license), GR00T N1 (NVIDIA research license), π0 (Physical Intelligence research license). RT-1, RT-2, PaLM-E, and the Helix family are not openly weighted. This matters for Chapter 16; the recipes there assume you have a checkpoint, and the open-weight column of the table is the candidate list.

For the full bibliographic entries of every primary reference in this table, see Appendix E.2. For the lineage that produced this zoo — CLIP → RT-1 → RT-2 → OpenVLA → π0 → Helix / GR00T N1 — see Chapters 11 through 14. For the open scientific problems that remain across the zoo, see Chapter 18.