1 Proof of Life: A Reproducible Robotics RL Loop

Vlad Prytula

2026-02-28

Contents

1 Proof of Life: A Reproducible Robotics RL Loop

This chapter covers:

- Setting up a containerized MuJoCo + Gymnasium-Robotics environment that works on Linux (NVIDIA GPU) and Mac (Apple Silicon)
- Verifying GPU access, physics simulation, and headless rendering through a structured test sequence
- Running a complete training loop and inspecting the artifacts it produces
- Establishing the experiment contract -- checkpoints, metadata, evaluation reports -- that you will reuse in every later chapter
- Introducing three diagnostic questions that prevent wasted compute

Reinforcement learning for robotics has an uncomfortable failure mode: everything runs, nothing learns. There are no compiler errors, no stack traces, no red text. The training loop finishes, the checkpoint saves, and when you evaluate the policy, the robot sits still -- or flails -- or does something that looks vaguely intelligent but succeeds 3% of the time.

This chapter helps you catch that failure mode early. By the end, you will have a working environment you can trust -- not because it "seems fine," but because you have concrete evidence: a rendered frame proving the physics engine works, a saved checkpoint proving the training loop completes, and a set of diagnostic habits that will serve you for the rest of the book.

We call this a "proof of life" -- borrowed from hostage negotiations, where it means evidence that someone is still alive. Here, it means evidence that the computational environment is alive: capable of producing valid, reproducible results.

The chapter is structured in two parts. First, we inspect the environment by hand -- looking at observations, manually computing rewards, and checking success conditions (section 1.5, "Build It"). Then we run the automated verification pipeline that checks the full stack from GPU to training loop (section 1.6, "Run It"). This two-part pattern -- understand the pieces, then run the pipeline -- recurs in every chapter of the book.

> **Tip:** If you want to verify your setup immediately, jump to section 1.6 and run `bash docker/dev.sh python scripts/ch00_proof_of_life.py all`. If it passes, your environment works. Then come back here to understand what it checked and why each test matters.

1.1 Why robotics RL fails silently

Let's start with the problem that motivates everything in this chapter.

You train a policy for eight hours. The training loop runs without errors. You evaluate the policy and find a success rate of 0%. Zero. You check the reward curve -- it is flat.

You check the logs -- nothing unusual. What went wrong?

The answer could be any of a dozen things: a rendering backend that silently fails, a CUDA driver mismatch that forces CPU training without telling you, a package version incompatibility that changes the observation format, a reward function that returns the wrong sign. The code *ran*. It just did not *work*.

This is harder to debug than a typical software bug. A web server that crashes is easier to fix than a web server that serves wrong answers. A compiler error is easier to fix than silent data corruption. In RL, the equivalent of silent data corruption is a flat reward curve -- and it can waste days of compute before you notice.

The problem gets worse with robotics. Physics simulators like MuJoCo have system-level dependencies (shared libraries, GPU drivers, rendering backends) that can fail in platform-specific ways. A setup that works on your laptop may break on a remote server. A setup that works today may break after a system update. And because RL training is stochastic by nature -- different random seeds produce different trajectories, different gradient updates, different final policies -- it can be genuinely hard to tell whether a bad result is caused by a software bug, a configuration error, or just bad luck.

The "runs but doesn't learn" problem

To make this concrete: in our experience developing the code for this book, we encountered a bug where the rendering backend (EGL) failed to initialize but the error was caught and silently swallowed. Training proceeded using a fallback code path that changed the observation normalization. The reward curve looked plausible -- it moved, it did not diverge -- but the success rate stayed at 0% for 500,000 steps. The fix was a two-line change to an environment variable. But finding the bug took an entire day, because nothing *crashed* and the training output looked superficially normal.

Henderson et al. (2018) documented a reproducibility crisis in deep RL that goes beyond individual bugs: many published results could not be replicated, even by the original authors, because the experimental conditions were underspecified. Their paper showed that seemingly minor details -- random seed, network initialization, even the number of parallel environments -- could change the outcome from "learns successfully" to "fails completely."

In our experience, the single most useful thing you can do before training is verify that your environment, rendering stack, training loop, and evaluation protocol all work correctly. Once you have that confidence, a training failure tells you something interesting -- the problem lies in the algorithm, the hyperparameters, or the task itself, not in a broken setup.

This chapter gives you that confidence systematically. By the end, you will have verified the full stack, so you can focus on the interesting problems.

Three questions before you train

Before running any experiment in this book, we find it useful to ask three questions. These come from the mathematician Hadamard, who studied when problems have reli-

able solutions. The math is not important here -- they translate directly to engineering:

1. **Can this be solved?** Is there a policy that could achieve the goal, given the observation and action spaces? For some tasks, the answer is genuinely uncertain. A policy with no access to the object's position cannot learn to push it. A 2-layer MLP may not have enough capacity for a high-dimensional visual task.
2. **Is the solution reliable?** Will different random seeds give similar results, or is success a fluke? If you train five times and succeed once, you probably got lucky -- the algorithm is not reliably solving the task.
3. **Is the solution stable?** Will small changes in hyperparameters or environment configuration break it? A solution that only works with a learning rate of exactly 3e-4 and falls apart at 5e-4 is fragile and hard to trust.

If the answer to any of these is "no" or "we don't know," we have work to do before training.

For this chapter, the answers are reassuring. The task (FetchReachDense, where the robot moves its end-effector toward a target) is well within the capability of standard RL algorithms -- even random exploration occasionally reaches the goal. Results are consistent across seeds; the success rate does not vary wildly. And the environment is not sensitive to small configuration changes. But asking these questions explicitly -- even when the answers are easy -- builds a habit that prevents expensive mistakes later.

We will use these three questions as a lightweight checklist at the start of every chapter. In later chapters, where we tackle sparse rewards (Chapter 5) and contact-rich manipulation (Chapter 6), the answers become less obvious -- and the questions become more valuable.

1.2 The task family: goal-conditioned Fetch manipulation

The experiments in this book use the Fetch family of environments from Gymnasium-Robotics. These are simulated robotic manipulation tasks built on the MuJoCo physics engine -- a high-fidelity rigid-body dynamics simulator originally developed at the University of Washington and now maintained by Google DeepMind. A 7-degree-of-freedom (7-DOF) Fetch robotic arm sits on a table and must achieve goals: reaching a target position, pushing an object to a location, or picking up an object and placing it elsewhere.

Let's get a feel for what we are working with. The Fetch arm is a silver industrial manipulator about a meter tall, with seven visible joints connecting chunky links that taper down to a parallel-jaw gripper with rubber finger pads. It sits behind a low table, and its reachable workspace -- the volume where it can operate -- spans roughly 60 x 70 x 20 cm on and above the table surface. When the arm moves, it traces smooth, deliberate arcs; you issue Cartesian velocity commands ("move right, move up"), and an internal inverse-kinematics controller translates those into joint motions. In the simplest task, FetchReach, a small red sphere floats in the air near the table, and success means the fingertips arrive within 5 cm of that sphere -- roughly the width of two fingers side by side. It sounds easy, but getting a learning algorithm to do it reliably from scratch is where the interesting challenges begin.

What makes these tasks interesting for learning is that they are goal-conditioned: the robot receives a desired goal (where to move, where to push the object) that changes every episode. The policy must generalize across goals, not just memorize a single target. This is a step toward real-world utility -- a robot that can only reach one fixed position is not very useful.

Why this environment family

A reasonable question at this point: why build an entire book around one robot in one simulator? Most RL textbooks rotate between CartPole, Atari, and MuJoCo locomotion -- a different environment each chapter. We take the opposite approach, and we want to be explicit about why.

We chose Fetch for three reasons:

1. **Goal conditioning enables the algorithms we care about.** The central challenge of this book is learning manipulation from sparse binary rewards -- "did you succeed or not?" -- rather than hand-designed distance signals. Solving this requires Hindsight Experience Replay (HER, Chapter 5), which needs an environment that separates `achieved_goal` from `desired_goal` so that failed trajectories can be relabeled as successes for different goals. Fetch environments provide this interface natively. Many popular RL benchmarks (CartPole, Ant, HalfCheetah) do not, which would force us to retrofit goal conditioning rather than teach it.

2. **The difficulty ladder lives inside one family.** Fetch provides five distinct challenges without changing the API:

   - Reaching a target (arm control only, continuous feedback)
   - Reaching with sparse reward (same task, binary signal -- the HER motivation)
   - Pushing an object (contact physics, multi-phase control)
   - Picking and placing (grasp coordination, goals in the air)
   - Pushing from camera images (same task, but raw pixels instead of state vectors)

   Each step adds exactly one new difficulty. When HER improves sparse Push from 0% to 99% success, you know the improvement comes from the algorithm, not from switching to an easier environment. This controlled progression is hard to achieve when hopping between unrelated benchmarks.

3. **Research comparability.** Fetch environments are widely used in the goal-conditioned RL literature (Plappert et al., 2018; Andrychowicz et al., 2017). Our results are directly comparable to published work. When we report 95% success on FetchPush-v4 with SAC+HER, a reader can check that number against the original HER paper and know we are in the right ballpark.

Depth over breadth

This is a deliberate pedagogical choice: we believe you learn more from understanding one task deeply than from running many tasks superficially. The cost is real -- you will not see locomotion, multi-agent coordination, or real hardware in the main text. We

address portability in the appendices (Appendix C ports the methodology to PyBullet's PandaGym; Appendix E demonstrates it on NVIDIA Isaac). But the core curriculum stays with Fetch because depth enables something breadth cannot: controlled variable elimination.

When a training run fails in Chapter 5 (sparse rewards), is the problem the algorithm, the reward, the exploration, or the environment? Because you already solved the same environment with dense rewards in Chapters 3-4, you can rule out the environment. When pixel-based training struggles in Chapter 9, you already solved the same task from state vectors -- so you know the task is solvable, and the problem must be in the visual pipeline. With a new environment each chapter, every failure has five possible causes; with one environment family, you isolate variables the way a lab scientist would.

The Fetch environments form a natural difficulty ladder that we will climb throughout the book. Notice that the rightmost column spans all ten chapters -- same robot, same simulator, increasing challenge:

Environment	Task	What changes
FetchReachDense-v4	Move end-effector to target	Baseline: dense reward, state vectors
FetchReach-v4	Same task, sparse reward	Reward signal: continuous -> binary
FetchPush-v4	Push block to target position	Task complexity: contact, multi-phase
FetchPickAndPlace-v4	Pick up block, place at target	Coordination: grasp + lift + place
FetchPush-v4 (pixels)	Push block, from camera images	Observation: 25D vectors -> 84x84 in

All five rows share the same robot, the same action space, and (except for pixels) the same observation structure. The "What changes" column is the key: each row adds exactly one new challenge. By keeping everything else constant, we isolate what matters -- when sparse-reward training fails, the culprit is the reward signal, not a new environment's quirks. When pixel training is slower, the culprit is the observation modality, not different dynamics.

Observations. Every Fetch environment returns a dictionary with three keys:

- observation -- the robot's proprioceptive state (joint positions, velocities, end-effector position). For FetchReach, this is a 10-dimensional vector. For environments with objects, it is larger (25D for FetchPush and FetchPickAndPlace) because it includes the object's position, rotation, and velocity.
- desired_goal -- where the robot should move its end-effector (or the object, for push/pick-and-place). A 3D position in Cartesian space (x, y, z coordinates in meters).
- achieved_goal -- where the end-effector (or object) currently is. Also a 3D position.

This three-part structure is a convention from the Gymnasium GoalEnv interface. It will show up in every chapter, so it is worth getting familiar with now. The separation of achieved_goal and desired_goal from the main observation is what makes goal conditioning explicit -- the environment literally tells the policy "here is where you are, here is where you should be."

Actions. Actions are 4-dimensional vectors: three components for the desired Cartesian velocity of the end-effector (dx, dy, dz) and one for gripper control (open/close), with each component bounded to [-1, 1]. For reaching tasks, the gripper dimension does not matter -- the robot is just moving its arm. For push and pick-and-place, the gripper becomes essential: the robot must learn to close the gripper around an object and open it at the target location.

The Cartesian action space is an important design choice, because it means the learning algorithm does not need to figure out how joint torques map to end-effector motion -- an inverse kinematics controller handles that internally. The policy therefore operates at a higher level of abstraction: "move the hand right and close the gripper." This makes the learning problem tractable for the algorithms we use.

Rewards. Fetch environments come in two flavors:

- *Dense reward* (e.g., `FetchReachDense-v4`): the reward is the negative Euclidean distance between `achieved_goal` and `desired_goal`. Closer is better. The reward is always negative or zero, with 0 meaning the goal is perfectly achieved.
- *Sparse reward* (e.g., `FetchReach-v4`): the reward is -1 if the goal is not achieved and 0 if it is. "Achieved" means the distance is below a threshold (5 cm for Reach).

Dense rewards give the learning algorithm continuous feedback -- "you're getting warmer." Sparse rewards give a binary signal -- "success or failure." Sparse rewards are harder to learn from but more natural: in real robotics, you often know only whether the task succeeded, not by how much you missed. Learning from sparse rewards is one of the central challenges of this book, and we will tackle it directly in Chapter 5 using Hindsight Experience Replay (HER -- an algorithm that turns failed attempts into useful training data by asking "what goal would this attempt have achieved?").

For this chapter, we use `FetchReachDense-v4` -- the easiest variant. We are not trying to learn anything interesting yet. We are verifying that the machinery works.

> **NOTE:** We formalize dense and sparse rewards mathematically in Chapter 2. For now, the intuition is sufficient: dense = continuous distance feedback, sparse = binary success/failure signal.

1.3 The experiment contract

Every chapter in this book produces concrete artifacts. Not screenshots, not "it looked like it was working," not a training curve in TensorBoard (a web-based visualization dashboard for monitoring training runs) that you squint at and decide looks "good enough" -- files on disk that you can inspect, share, and reproduce.

Here is the contract:

Artifact	Format	Example
Checkpoints	Stable Baselines 3 (SB3) `.zip` + `.meta.json`	checkpoints/ppo_FetchReach
Evaluation reports	JSON	results/ch02_ppo_fetchreac
TensorBoard logs	Event files	runs/ppo/FetchReachDense-v
Videos	MP4	videos/ppo_FetchReachDense

The ppo in the example filenames refers to PPO (Proximal Policy Optimization), the algorithm used for training -- we introduce PPO in Chapter 3. The two most important columns are "Format" and "Example," because they make the contract concrete: checkpoints are files with known paths and machine-readable metadata, and evaluation reports are JSON -- not prose, not impressions, but structured data with success rates, episode returns, goal distances, and seed counts. When this book says "94% success rate," there is a JSON file that contains that number, and you can verify it yourself.

Why provenance matters. Henderson et al. (2018) documented a reproducibility crisis in deep RL: many published results could not be replicated, even by the original authors, because the experimental conditions were underspecified. Random seeds, hyperparameters, library versions, and hardware all matter. A single random seed can be the difference between 95% success and 40% success on the same algorithm with the same hyperparameters. A result with provenance -- one you can trace back to a specific command, seed, and environment -- is worth far more than one without it.

Our defense is simple: every experiment in this book is defined by a single command, produces versioned artifacts, and records the conditions under which it ran. The `.meta.json` file alongside each checkpoint captures the algorithm, environment, seed, step count, and library versions. The evaluation JSON records exactly which checkpoint was evaluated, on which environment, with which seeds, and whether the policy was deterministic. If you want to know how we got a number, you can find the exact command in the chapter, run it yourself, and compare.

We find it helpful to hold ourselves to a simple standard: every number we report should trace back to a file on disk. If we can point to the JSON, the result is real. If we cannot, we treat it as unverified.

The training and evaluation CLIs. Every chapter uses the same two entry points:

- `train.py` -- trains a policy. Key flags: `--algo` (PPO, SAC, or TD3 -- algorithm names we introduce in Chapters 3-4), `--env` (environment ID), `--seed`, `--total-steps`, and `--her` (enables Hindsight Experience Replay for off-policy algorithms). Produces a checkpoint `.zip` and a `.meta.json`.
- `eval.py` -- evaluates a saved checkpoint. Key flags: `--ckpt` (path to the checkpoint), `--env`, `--n-episodes`, `--seeds`, `--deterministic`, and `--json-out`. Produces a JSON evaluation report.

You do not need to remember these flags now -- each chapter provides the exact commands. The point is that the interface is consistent: same CLI, same artifact format, same evaluation protocol. When you learn to read evaluation JSON in Chapter 3, that skill applies to every chapter that follows.

This chapter's contract is lighter than later chapters (we are smoke-testing, not training a real policy), but the pattern starts here. You will see it in every chapter that follows.

1.4 Setting up the environment

Prerequisites

Before you start, you need two things on your host machine:

1. **Docker.** Verify with `docker --version`. Any recent version (20+) works.
2. **NVIDIA Container Toolkit** (Linux with GPU only). Verify with `docker run --rm --gpus all nvidia/cuda:12.0-base nvidia-smi`. You should see your GPU listed.

On Mac, Docker Desktop is all you need -- there is no GPU passthrough, and the workflow uses CPU inside a Linux container.

If either check fails, consult your system administrator or Docker's installation documentation. This book does not cover Docker installation, but the above two commands are a fast way to confirm you are ready.

Docker as environment specification

We use Docker containers for all experiments, primarily for reproducibility.

Here is the scenario we are trying to prevent: you train a policy, achieve 94% success, and send the code to a colleague. They install the dependencies, run the same command, and get 12% success -- or a crash. The difference? Their MuJoCo version is slightly different. Or their CUDA toolkit does not match. Or a transitive dependency upgraded since you pinned yours. These problems are invisible and maddening.

A Docker container packages your code, its dependencies, the operating system libraries, and the configuration into a single artifact. The container is identified by a content-addressable hash, so "I ran image abc123" is unambiguous. If the container runs on your machine, it runs on your colleague's machine, on a cloud server, or on a DGX workstation -- with the same behavior.

The two-layer architecture

Our container has two layers, and the separation is deliberate:

- **Base layer.** The NVIDIA PyTorch image (`nvcr.io/nvidia/pytorch:25.12-py3`) provides CUDA, cuDNN, and PyTorch pre-configured and tested by NVIDIA. This is the heavyweight layer -- several gigabytes, carefully version-matched. It rarely changes between experiments.
- **Project layer.** On top of the base, our `docker/Dockerfile` adds the project-specific dependencies: MuJoCo rendering libraries (`libegl1`, `libosmesa6` for headless rendering), `gymnasium`, `gymnasium-robotics`, `stable-baselines3`, `tensorboard`, and `imageio`. These are lighter, and they change when we update our experiment code.

The benefit of this separation is practical: if you change a Python dependency, only the lightweight project layer rebuilds. You do not need to re-download the multi-gigabyte NVIDIA base.

9

NOTE: For strict reproducibility, the Docker image digest (the content hash) is the true specification -- it freezes the entire dependency tree. If you need to capture exact versions for a paper or report, run `pip freeze` inside the container and save the output. The `requirements.txt` file specifies minimum version constraints (e.g., `stable-baselines3>=2.4.0`), which is sufficient for the experiments in this book.

The dev.sh entry point

The `docker/dev.sh` script is the single entry point for everything in this book. It handles building the image, launching the container, mounting your code, setting up a Python virtual environment, and dropping you into a shell. You do not need to learn Docker commands -- `docker/dev.sh` wraps them all.

```
# Enter an interactive development shell
bash docker/dev.sh

# Or run a single command
bash docker/dev.sh python scripts/ch00_proof_of_life.py all
```

When you run docker/dev.sh, it:

1. Checks for the `robotics-rl:latest` image and builds it if missing (this takes a few minutes on first run)
2. Launches a container with GPU access (`--gpus all` on NVIDIA systems)
3. Mounts your repository at `/workspace` inside the container, so changes you make to code on the host are instantly visible inside the container and vice versa
4. Runs as your host user ID (so files created inside have your ownership, not root's)
5. Creates a `.venv` virtual environment and installs dependencies from `requirements.txt`
6. Activates the environment and runs your command (or starts a shell)

Every command in this book that starts with `bash docker/dev.sh ...` follows this pattern. You will type it many times. If you prefer, you can enter an interactive shell with `bash docker/dev.sh` and then run commands directly inside the container.

> **TIP:** Three environment variables control headless rendering: `MUJOCO_GL=egl` selects the hardware-accelerated EGL backend, `PYOPENGL_PLATFORM=egl` configures PyOpenGL to match, and `NVIDIA_DRIVER_CAPABILITIES=al` enables full GPU access in the container. The `docker/dev.sh` script sets all of these automatically. You should not need to set them yourself unless you are debugging rendering issues (see section 1.7).

The virtual environment inside the container

You might wonder why we need a virtual environment inside an already-isolated container. There are practical reasons: it lets us do editable installs (`pip install -e .`) for rapid development, lets project dependencies shadow container defaults when needed, and keeps the dependency specification explicit in `requirements.txt` rather than scattered across the Dockerfile.

On NVIDIA systems, the venv is created with `--system-site-packages`, which lets it inherit PyTorch from the base container. This avoids reinstalling the large, CUDA-specific PyTorch package. On Mac, where there is no CUDA, the venv installs CPU PyTorch from scratch.

The first time you run `docker/dev.sh`, it installs all dependencies -- this adds a minute or two. Subsequent runs skip installation (it hashes `requirements.txt` and only reinstalls when the file changes).

SIDEBAR: Running on Mac (Apple Silicon)

The same `docker/dev.sh` command works on Mac. The script auto-detects your platform (uname `-s` = Darwin for Mac) and uses an appropriate Docker image (`robotics-rl:mac` built from `docker/Dockerfile.mac`, which uses `python:3.11-slim` as its base instead of the NVIDIA image).

Key differences from Linux/NVIDIA:

	Linux/NVIDIA	Mac
Compute	CUDA GPU	CPU
Rendering	EGL (hardware-accelerated)	OSMesa (software)
Throughput	~600 fps	~60-100 fps
Docker image	`robotics-rl:latest`	`robotics-rl:mac`
Base image	`nvcr.io/nvidia/pytorch`	`python:3.11-slim`

The 6-10x throughput gap is mostly about faster CPUs and more cores on Linux machines, not GPU acceleration. For state-based RL (Chapters 1-8), both platforms are CPU-bound: MuJoCo physics dominates, and a 256x256 MLP processing a 25D observation vector completes in microseconds regardless of device. Mac is fully viable through Chapter 8 -- training runs that take seconds on DGX take tens of minutes on Mac, which is fine for learning and iteration. The GPU becomes important at Chapter 9 (pixel observations, where a CNN processes 84x84 images every step) and essential at Appendix E (Isaac Lab, where physics itself runs on the GPU). For Chapter 9, RAM is usually the binding constraint rather than GPU speed -- see that chapter for buffer-size guidance.

Apple's MPS (Metal Performance Shaders) backend exists for PyTorch but cannot work inside Docker, since Docker on Mac runs a Linux VM that has no access to the Metal framework. CPU is the reliable default.

If you encounter out-of-memory errors on Mac, increase Docker Desktop's memory allocation to at least 8 GB (Settings -> Resources -> Memory -> Apply & restart).

1.5 Build It: Inspecting what the robot sees

Before running the full pipeline, let's look under the hood. This section teaches you to inspect the environment directly -- the observation structure, the reward computation,

and the success signal. These are the building blocks that every later chapter depends on.

In later chapters, Build It sections will have you implementing algorithms from scratch -- writing loss functions, replay buffers, and update loops in raw PyTorch. This chapter's Build It is lighter because there is no algorithm to implement yet. Instead, you will learn to talk to the environment: query its observations, manually compute rewards, and check success conditions. These are diagnostic skills that come in handy whenever something goes wrong in a later chapter.

1.5.1 The observation dictionary

The first thing to understand about any RL environment is what the agent sees. In Fetch environments, the agent sees a dictionary, not a flat vector. This is unusual compared to classic RL environments like CartPole, where the observation is a simple array.

Create a Fetch environment and inspect what it gives you:

```python
import gymnasium as gym
import gymnasium_robotics  # registers Fetch envs

env = gym.make("FetchReachDense-v4")
obs, info = env.reset(seed=42)

print(type(obs))            # <class 'dict'>
print(sorted(obs.keys()))  # ['achieved_goal', 'desired_goal', 'observation']

print(obs["observation"].shape)    # (10,)
print(obs["achieved_goal"].shape)  # (3,)
print(obs["desired_goal"].shape)   # (3,)
```

The observation dictionary has three arrays:

- obs["observation"] (shape (10,)) -- the robot's proprioceptive state: gripper position (3D), gripper linear velocity (3D), finger positions (2D), and finger velocities (2D).
- obs["achieved_goal"] (shape (3,)) -- where the end-effector currently is in 3D Cartesian space.
- obs["desired_goal"] (shape (3,)) -- where we want the end-effector to be.

Notice that achieved_goal is redundant with the first three elements of observation (both report the end-effector position). This redundancy is by design -- it lets the goal-conditioned interface work uniformly across environments. For pushing and pick-and-place tasks, achieved_goal will be the *object* position, not the end-effector position, while the robot's own position stays in observation.

What the values mean physically. The numbers in these arrays are not abstract -- they correspond to real physical quantities in the simulated world. The goal positions are in meters relative to the MuJoCo world frame. As Figure 1.1 shows, the Fetch arm sits on a table with a workspace spanning roughly x in [1.0, 1.6], y in [0.4, 1.1], and

z in [0.4, 0.6] -- a roughly 60 cm x 70 cm x 20 cm box on and above the table. The velocities in `observation` are in meters per second. Understanding these scales will help you debug later: if you ever see goal positions at (0, 0, 0) or velocities in the thousands, something is wrong with the environment state.

> **Checkpoint.** Run the code above inside the container (`bash docker/dev.sh python -c "..."`). Verify that you get three keys, that shapes match `(10,)`, `(3,)`, and `(3,)`, and that values are finite floating-point numbers (not NaN, not inf). Check that the goal positions are within the workspace bounds described above. If shapes differ, check your gymnasium-robotics version.
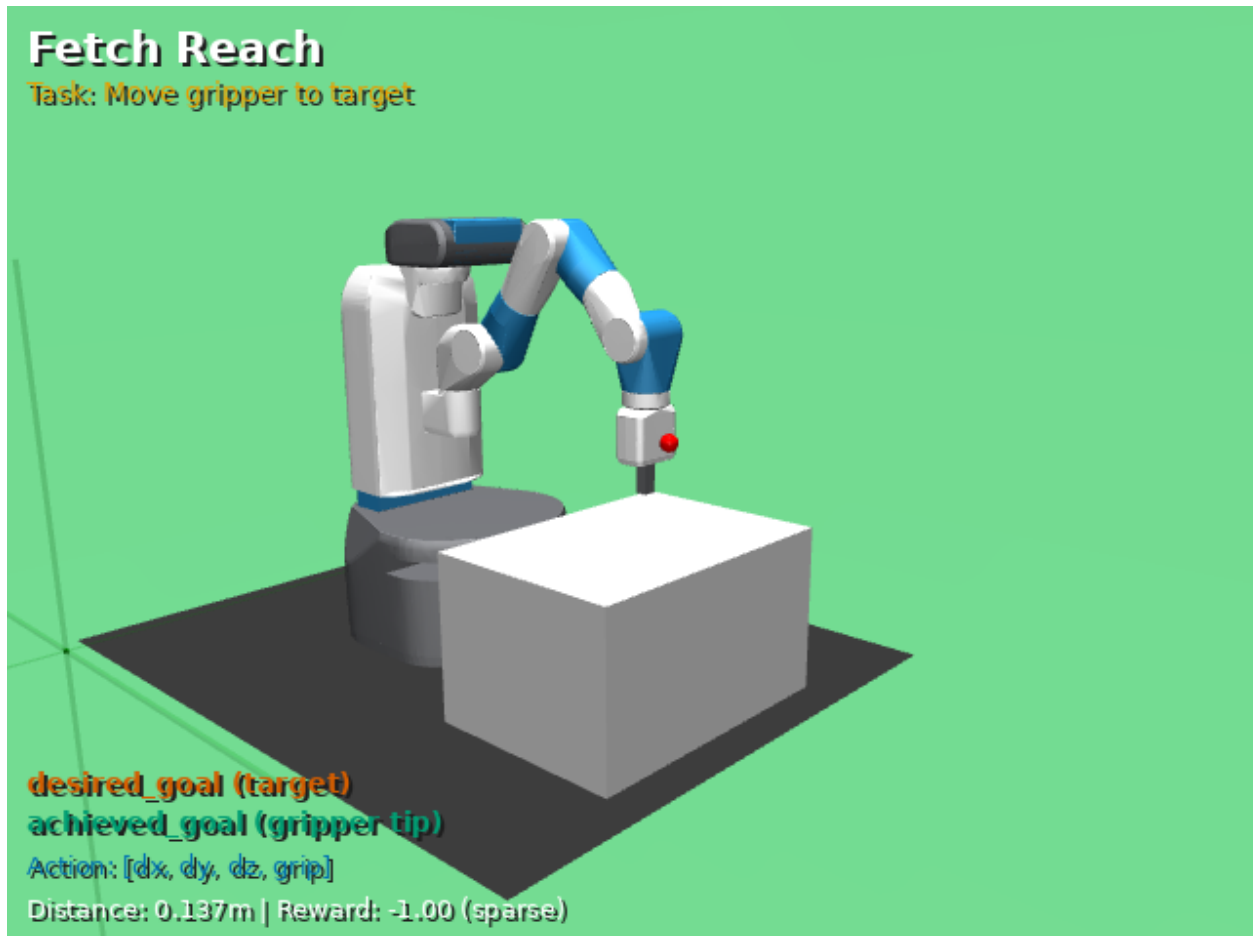
Figure 1.1: The FetchReach environment after `env.reset()`. The red sphere marks the desired goal -- the 3D position the end-effector must reach. The gripper's current position is the achieved goal. The observation dictionary separates these into `desired_goal` and `achieved_goal`, making goal conditioning explicit. (Generated by `bash docker/dev.sh python scripts/capture_proposal_figures.py env-setup --envs FetchReach-v4`. FetchReachDense-v4 is visually identical; only the reward differs.)

1.5.2 Manual compute_reward: the critical invariant

Every Fetch environment exposes a `compute_reward` method that takes an achieved goal, a desired goal, and an info dictionary, and returns a reward. We access it via `env.unwrapped.compute_reward(...)` because `gym.make()` wraps the environment in several layers (time limits, order enforcement) that do not expose `compute_reward` directly -- `.unwrapped` reaches through to the base Fetch environment. This method is the foundation of Hindsight Experience Replay (HER), which we introduce in Chapter 5. HER works by asking: "what goal would this trajectory have achieved?" and recomputing rewards accordingly. If `compute_reward` does not match the reward that `env.step()` returns, HER learns from incorrect data.

Let's verify this invariant directly:

```python
import numpy as np

env = gym.make("FetchReachDense-v4")
obs, info = env.reset(seed=42)

# Take a random action
action = env.action_space.sample()
next_obs, step_reward, terminated, truncated, info = env.step(action)

# Recompute the reward manually
manual_reward = env.unwrapped.compute_reward(
    next_obs["achieved_goal"],
    next_obs["desired_goal"],
    info,
)

print(f"Step reward:   {step_reward:.6f}")
print(f"Manual reward: {manual_reward:.6f}")
print(f"Match: {np.isclose(step_reward, manual_reward)}")
```

Expected output (your exact values will differ -- the important result is `Match: True`):

```
Step reward:   -1.058329
Manual reward: -1.058329
Match: True
```

The specific reward value depends on the random action sampled, which varies across library versions and seeds. What matters is that the two values match exactly -- we call this the critical invariant: `compute_reward(achieved_goal, desired_goal, info)` must equal the reward from `env.step()`, since every chapter that uses HER depends on it.

Why does this matter so much? When HER relabels a failed trajectory -- "you did not reach the goal at position (1.3, 0.7, 0.5), but you did reach position (1.2, 0.6, 0.42)" -- it needs to recompute the reward as if that alternate position had been the goal all along. It does this by calling compute_reward(achieved_goal=actual_position, desired_goal=relabeled_goal, info). If this function returns a different value than

`env.step()` would have returned for the same state, then HER is training on incorrect reward labels. The policy learns from corrupted data, and training may silently fail or converge to a bad policy.

> **Checkpoint.** Run this check with several different seeds and actions. The rewards should always match exactly (not approximately -- exactly). If they do not, there is a version incompatibility between gymnasium and gymnasium-robotics that may cause problems in Chapter 5.

For dense rewards, the manual reward equals the negative Euclidean distance between the achieved goal and the desired goal:

```python
distance = np.linalg.norm(
    next_obs["achieved_goal"] - next_obs["desired_goal"]
)
print(f"Negative distance: {-distance:.6f}")
print(f"Dense reward:      {step_reward:.6f}")
```

These should also match exactly. The dense reward equals `-distance` -- nothing more.

You can also verify the sparse reward variant. If you create `FetchReach-v4` (without "Dense") and perform the same check, the reward will be either -1.0 (goal not achieved) or 0.0 (goal achieved). The same `compute_reward` invariant holds for sparse rewards -- the function just returns a different value:

```python
sparse_env = gym.make("FetchReach-v4")
sparse_obs, _ = sparse_env.reset(seed=42)
action = sparse_env.action_space.sample()
next_obs, reward, _, _, info = sparse_env.step(action)
print(f"Sparse reward: {reward}")  # -1.0 (most likely)
```

1.5.3 Parsing the success signal

The `info` dictionary returned by `env.step()` contains an `is_success` field that indicates whether the goal was achieved. For FetchReach, "achieved" means the distance between the end-effector and the target is below 5 cm (0.05 meters):

```python
distance = np.linalg.norm(
    next_obs["achieved_goal"] - next_obs["desired_goal"]
)
print(f"Distance: {distance:.4f} m")
print(f"Success:  {bool(info['is_success'])}")
print(f"Threshold: 0.05 m")
print(f"Below threshold: {distance < 0.05}")
```

The success signal is what we measure during evaluation. When later chapters report "94% success rate," they mean that across many episodes, `info['is_success']` was True at the end of 94% of them. This is the metric that matters most -- not the reward, not the return, not the loss value, but the fraction of episodes where the robot actually achieved the goal. A high return with a low success rate means the robot is getting

close but not close enough; a high success rate with a modest return means the robot succeeds but takes a roundabout path.

Notice how the three things we have inspected fit together: observations tell the policy where it is and where it should be, rewards give the policy a training signal based on that information (dense: how close? sparse: success or failure?), and the success metric is the binary outcome we ultimately care about. In other words, the reward drives learning, while the success signal measures whether learning worked. In dense-reward environments, a well-trained policy will have both high reward (close to 0) and high success rate (close to 100%). In sparse-reward environments, the relationship is starker: the reward is -1 until the moment of success, then 0. This makes dense rewards more informative for learning but sparse rewards more honest about what we actually want.

> **Checkpoint.** On a random action immediately after reset, the distance is typically 0.05-0.15 meters and `is_success` is usually False (the robot's initial position is rarely on top of the goal). If `is_success` is True on the first step with a random action, something is unusual -- check that the environment is creating diverse goal positions.

1.5.4 The bridge: Build It meets Run It

We have now verified three things by hand:

1. The observation dictionary has the expected structure and shapes
2. `compute_reward(ag, dg, info)` matches the reward from `env.step()`
3. The success signal corresponds to goal distance below a threshold

Each of these checks connects to the production pipeline in specific ways.

When SB3 creates a PPO or SAC model with `MultiInputPolicy`, it reads the observation space to determine the input structure -- the same `Dict` with three `Box` entries that you inspected in section 1.5.1. If the shapes were wrong or the keys were missing, model creation would fail silently or produce a network with the wrong architecture. By inspecting the observation yourself, you know exactly what the model will see.

When HER (Chapter 5) relabels goals, it calls `compute_reward` with different goal values and trusts the returned reward to be consistent with what `env.step()` would have returned. The check in section 1.5.2 verifies that this trust is warranted. If you ever upgrade gymnasium-robotics and want to confirm that nothing broke, this is the check to run.

When `eval.py` reports a success rate, it counts how many episodes ended with `info['is_success'] == True` -- the same signal you examined in section 1.5.3. If the threshold were wrong, or if `is_success` did not correspond to the distance you measured, the evaluation numbers would be meaningless.

In later chapters, the Build It sections are more substantial -- you will implement entire algorithms from scratch. But the principle is the same: understand the pieces by hand before trusting the production code to assemble them correctly.

1.6 Run It: The proof-of-life pipeline

Now we run the full verification sequence. The script `scripts/ch00_proof_of_life.py` implements four tests, each verifying a necessary condition for the experiments that follow.

```
------------------------------------------------------------
EXPERIMENT CARD: Proof of Life
------------------------------------------------------------
Environment:  FetchReachDense-v4 (smoke test only)
Fast path:    ~5 min (GPU) / ~10 min (CPU)

Run command:
  bash docker/dev.sh python scripts/ch00_proof_of_life.py all

Artifacts:
  smoke_frame.png           (headless render validation)
  ppo_smoke.zip             (training loop validation)
  ppo_smoke.meta.json       (smoke run metadata: versions, seed, device)

Success criteria:
  smoke_frame.png exists, non-empty, shows Fetch robot
  ppo_smoke.zip exists, loadable by PPO.load()
  ppo_smoke.meta.json exists, contains env_id and versions
  All 4 subtests pass (gpu-check, list-envs, render, ppo-smoke)
------------------------------------------------------------
```

The four-test verification sequence

The `all` subcommand runs these tests in order. If a test fails, diagnose and fix it before proceeding -- later tests depend on earlier ones.

Test 1: GPU check (`gpu-check`)

Verifies that PyTorch can see the GPU via `torch.cuda.is_available()`. On a Linux system with NVIDIA hardware, this should report CUDA available with the device name and count. You should see something like:

`OK: CUDA available -- 1 device(s), primary: NVIDIA A100-SXM4-80GB`

On Mac, this correctly reports "CUDA not available" -- training proceeds on CPU, which is expected and functional:

`WARN: CUDA not available; training will use CPU (this is expected on Mac)`

This test always passes (it warns rather than fails on Mac or CPU-only systems) because CPU-only operation is valid. But on a system where you expect a GPU, treat a "not available" warning as a real problem: check that Docker was invoked with `--gpus all` and that the NVIDIA Container Toolkit is installed.

We verify GPU access early because later chapters need it -- but not all chapters, and not for the same reasons:

Chapters	Workload	CPU viable?
1-8	State-based RL (MuJoCo + small MLPs on 25D vectors)	Yes (~60-100 fps Mac, ~600 fp
9	Pixel-based RL (CNN on 84x84x12 images)	Slow but possible
App. E	Isaac Lab (GPU-parallel PhysX)	No

For Chapters 1-8, the bottleneck is MuJoCo physics simulation, which runs on CPU regardless of platform. Training runs that take minutes on a Linux GPU machine take tens of minutes on a Mac laptop -- not days.

Test 2: Fetch environment registry (`list-envs`)

Imports `gymnasium_robotics` and lists all registered Fetch environments. You should see a list that includes:

```
FetchPickAndPlace-v3
FetchPickAndPlaceDense-v3
FetchPush-v3
FetchPushDense-v3
FetchReach-v3
FetchReachDense-v3
FetchReach-v4
FetchReachDense-v4
...
```

The `-v3` and `-v4` variants are both present; we use `-v4` throughout this book (the latest version at time of writing). If no Fetch environments appear, `gymnasium-robotics` is not installed correctly.

Test 3: Headless rendering (`render`)

Creates a Fetch environment with `render_mode="rgb_array"`, calls `env.render()`, and saves the frame as `smoke_frame.png`. This tests the entire rendering pipeline: MuJoCo physics initialization, scene construction, camera setup, and offscreen rendering via EGL or OSMesa.

Why is rendering non-trivial? On a typical workstation or laptop, rendering uses the display server (X11 on Linux, the window system on Mac) to manage the OpenGL context. But DGX systems and cloud servers are headless -- they have no monitor attached and no display server running. Rendering must happen entirely offscreen, which requires either EGL (using the GPU's rendering capability directly, without a display) or OSMesa (a software-only renderer that produces images using the CPU). Both approaches have system library requirements that can fail silently.

The script implements a fallback chain: it first tries EGL (hardware-accelerated, preferred on NVIDIA systems), then OSMesa (software rendering, slower but more compatible). The fallback works by re-executing the entire script process with different environment variables -- so if EGL fails, you may see the script's startup output appear twice in your terminal. This is the re-exec mechanism switching backends, not an error.

On success, you see:

```
OK: wrote /workspace/smoke_frame.png
```

Test 4: PPO smoke training (`ppo-smoke`)

Runs PPO (Proximal Policy Optimization, a policy gradient method we derive and implement from scratch in Chapter 3) for 50,000 timesteps on `FetchReachDense-v4` with 8 parallel environments and saves a checkpoint as `ppo_smoke.zip`. This is enough to verify that the entire training pipeline (environment interaction, gradient computation, parameter updates, checkpoint saving) works end-to-end, though far too short to learn a good policy.

Why 50,000 steps and not 1,000,000? Because this is a smoke test, not a training run. We want to verify the machinery in under 5 minutes, not produce a useful policy. A full training run for FetchReachDense takes about 500,000 steps (Chapter 3). For now, we just need the loop to execute without crashing.

We use PPO for this smoke test rather than SAC (Soft Actor-Critic, an off-policy method we introduce in Chapter 4) because PPO is simpler and surfaces errors faster. It has fewer moving parts -- no replay buffer, no twin critics, no entropy tuning. If PPO runs, we can be confident the core infrastructure is sound. SAC-specific issues can be debugged separately when we get to Chapter 4.

> **WARNING:** If the PPO smoke test hangs or takes much longer than expected, check whether you are accidentally running on CPU when you expected GPU. The script's GPU check output at the top tells you which device is in use. On CPU, 50,000 steps may take 5-10 minutes; on GPU, about 1-2 minutes.

> **NOTE:** Low GPU utilization (~5-10%) during training is expected and normal. The bottleneck is CPU-bound MuJoCo physics simulation, not GPU-bound neural network operations. With small networks and small batch sizes, the GPU finishes its work in microseconds and waits for the CPU. This is the nature of RL with physics simulators, not a problem to solve.

Interpreting the artifacts

After `all` completes, you should have three new files in your repository root:

`smoke_frame.png` -- Open this file. You should see the Fetch robot arm on a table with a target marker (a small red sphere floating in the air near the arm). The table surface, the robot's silver links, and the red target should all be clearly visible. If the image exists but is black or empty, the rendering pipeline has a partial failure -- check the rendering backend output in the terminal. If it looks correct, headless rendering works.

This image is your visual confirmation that MuJoCo is simulating the robot correctly and that the rendering pipeline can turn the physics state into pixels. In later chapters, you will generate evaluation videos using the same pipeline.

`ppo_smoke.zip` -- This is a Stable Baselines 3 checkpoint containing the neural network weights and optimizer state, along with the observation/action space schema needed to load the model. You can verify it is loadable:

```
from stable_baselines3 import PPO
model = PPO.load("ppo_smoke.zip")
print(f"Policy type: {type(model.policy).__name__}")
print(f"Observation space: {model.observation_space}")
print(f"Action space: {model.action_space}")
```

The output should look like:

```
Policy type: MultiInputPolicy
Observation space: Dict('achieved_goal': Box(-inf, inf, (3,), ...),
   'desired_goal': Box(-inf, inf, (3,), ...),
   'observation': Box(-inf, inf, (10,), ...))
Action space: Box(-1.0, 1.0, (4,), float32)
```

A few things to notice here. The policy is a `MultiInputPolicy` -- not a `MlpPolicy` -- because observations are dictionaries, not flat vectors. SB3 handles the dictionary structure internally by processing each key separately and concatenating them before feeding into the neural network. The observation space is a `Dict` with three `Box` entries matching the shapes we saw in section 1.5. The action space is a `Box` with shape `(4,)` and bounds `[-1, 1]`, matching the Cartesian velocity + gripper action we described in section 1.2.

This checkpoint is not worth evaluating for performance -- it trained for only 50,000 steps, far too few to learn useful behavior on any task. Its purpose is to prove the loop runs, not that it learns; actual learning starts in Chapter 3.

ppo_smoke.meta.json -- A small metadata file capturing the environment ID, seed, training step count, device, and library versions used for the smoke run. Later chapters produce the same `.meta.json` alongside checkpoints in `checkpoints/`.

The dependency chain

The four tests form a logical dependency chain:

```
GPU check  ->  Fetch env registry  ->  Headless rendering  ->  Training loop
```

Each test assumes the previous ones work. Rendering depends on MuJoCo initializing correctly (Test 2). Training depends on rendering being at least attempted (the training test disables rendering, but it still needs MuJoCo and Gymnasium working). And training performance depends on the compute environment (Test 1) -- for state-based chapters (2-8), CPU is adequate; for pixel chapters (9+), GPU helps but RAM is often the binding constraint.

When something fails, diagnose in order: if rendering fails, check Test 2 first, since MuJoCo itself may not be initializing; if training is unexpectedly slow, check Test 1, since CUDA may not be available. The `all` subcommand runs the tests sequentially, so the output makes it clear which test failed.

What "proof of life" means

After these four tests pass, you know that the container has access to the GPU (or is correctly falling back to CPU), that MuJoCo and Gymnasium-Robotics are installed and functional, that headless rendering produces valid images, and that the full training loop (env -> policy -> training -> checkpoint) executes without error.

Together with the Build It checks from section 1.5, you also know that observations have the expected dictionary structure and shapes, that `compute_reward` matches `env.step()` (the critical invariant), and that the success signal correctly reflects goal distance.

This is what "alive" means: the environment can produce valid results. Producing good results -- that is what the rest of the book is for.

> **SIDEBAR: Regenerating figures and videos**
>
> Every figure in this book is generated from code -- no hand-drawn diagrams, no screenshots from external tools. This means you can regenerate any figure yourself. The figures you see (annotated environment screenshots, reward diagrams, learning curves) are produced by:
>
> ```bash
> bash docker/dev.sh python scripts/capture_proposal_figures.py all
> ```
>
> This creates annotated PNGs in the `figures/` directory. Individual subcommands are available: `env-setup` for environment screenshots, `reward-diagram` for reward function plots, `ppo-clipping` for the PPO clipping diagram, and `ppo-demo-curve` for learning curves.
>
> For evaluation videos showing trained policies in action, use:
>
> ```bash
> bash docker/dev.sh python scripts/generate_demo_videos.py \
>   --ckpt checkpoints/ppo_FetchReachDense-v4_seed0.zip \
>   --out videos/ppo_FetchReachDense-v4_seed0 \
>   --gif
> ```
>
> This produces `videos/ppo_FetchReachDense-v4_seed0.mp4` and `videos/ppo_FetchReachDen v4_seed0.gif`. We use these to verify that trained policies behave as expected -- not just that the numbers look right, but that the robot actually moves to the target. Later chapters provide specific video commands alongside their Experiment Cards.

1.7 What can go wrong

Here are the most common failures and how to fix them. We have encountered all of these during development. The list is roughly ordered by how early in the pipeline the failure occurs -- Docker issues first, then rendering, then training. If you hit a problem not listed here, the most productive diagnostic approach is to run the four tests individually (`gpu-check`, `list-envs`, `render`, `ppo-smoke`) and identify which one fails.

"Permission denied" when running Docker

Symptom. docker: Got permission denied while trying to connect to the Docker daemon socket.

Cause. Your user is not in the docker group.

Fix. Run sudo usermod -aG docker $USER, then log out and back in. Alternatively, prefix Docker commands with sudo (but this can cause file ownership issues).

"I have no name!" in the container shell

Symptom. The shell prompt shows I have no name!@<container-id>.

Cause. The container runs as your numeric UID (to match file ownership), and that UID has no entry in the container's /etc/passwd.

Impact. None. This is cosmetic. File permissions, training, and everything else work correctly. You can safely ignore it.

EGL initialization failure

Symptom. Errors mentioning gladLoadGL, eglQueryString, or libEGL.so.

Cause. The EGL rendering library is missing or the GPU driver does not expose EGL support.

Fix. The proof-of-life script automatically falls back to OSMesa. If you see the script output appear twice, that is the fallback mechanism -- not an error. If both EGL and OSMesa fail, check that libegl1 and libosmesa6 are installed in the container image. Rebuilding the image with bash docker/build.sh usually resolves this.

No Fetch environments found

Symptom. The list-envs test shows no output, or gym.make("FetchReachDense-v4") raises EnvironmentNameNotFound.

Cause. The gymnasium-robotics package is not installed.

Fix. Inside the container, check with pip list | grep gymnasium. You should see both gymnasium and gymnasium-robotics. If either is missing, run pip install gymnasium-robotics or rebuild the image.

CUDA not available (on a system with a GPU)

Symptom. The gpu-check test reports "CUDA not available" on a machine that has an NVIDIA GPU.

Cause. Either Docker was not invoked with --gpus all, or the NVIDIA Container Toolkit is not installed, or there is a driver mismatch between the host and the container.

Fix. First, verify the NVIDIA runtime works at all:

```
docker run --rm --gpus all nvidia/cuda:12.0-base nvidia-smi
```

If that command fails, the NVIDIA Container Toolkit is not installed or not configured. Follow the installation instructions from NVIDIA's documentation for your distribution. If `nvidia-smi` works but PyTorch still cannot see the GPU, there may be a CUDA version mismatch -- the container's CUDA toolkit version must be compatible with the host driver version. Run `nvidia-smi` on the host to check the driver version and CUDA compatibility.

On Mac, "CUDA not available" is expected and correct -- training uses CPU. This is not a problem to fix.

PPO smoke training crashes with shape/dtype errors

Symptom. Errors about tensor shapes or dtypes during training.

Cause. Usually a version mismatch between `stable-baselines3`, `gymnasium`, and `gymnasium-robotics`. The observation space handling changed between major versions.

Fix. Delete .venv and re-run `bash docker/dev.sh` to recreate the environment with correct versions. Check that `requirements.txt` specifies compatible versions.

Dependencies reinstall every time

Symptom. `docker/dev.sh` reinstalls packages on every run, adding several minutes of overhead.

Cause. The hash file `.venv/.requirements.sha256` is missing or corrupted. The script uses this hash to detect when `requirements.txt` has changed; if the file is missing, it assumes dependencies need updating.

Fix. Delete .venv entirely (`rm -rf .venv`) and re-run `docker/dev.sh`. The venv will be recreated from scratch and the hash file will be written correctly. Subsequent runs should skip installation.

Docker build fails or times out

Symptom. `docker/build.sh` (or the automatic build inside `docker/dev.sh`) fails with network errors, timeout errors, or "unable to pull" messages.

Cause. The base image (`nvcr.io/nvidia/pytorch:25.12-py3`) is large (several GB) and hosted on NVIDIA's container registry, which can be slow or require authentication for some images. Alternatively, `pip install` during the build may fail if your network blocks PyPI.

Fix. For network issues, retry the build -- transient failures are common with large downloads. If the NVIDIA registry requires authentication, run `docker login nvcr.io` first (a free NVIDIA developer account is sufficient). If pip install fails, check that your network allows HTTPS traffic to pypi.org. On corporate networks with proxy servers, you may need to configure Docker's proxy settings.

TIP: If the build succeeds once, the image is cached locally and you will not need to download it again unless you delete the image. You can verify your images with `docker images | grep robotics-rl`.

1.8 Summary

You now have a working laboratory. Specifically, you can trust four things:

- **The physics engine works.** MuJoCo initializes, Gymnasium-Robotics registers the Fetch environments, and observations have the expected structure: a dictionary with `observation` (10D), `achieved_goal` (3D), and `desired_goal` (3D).
- **Headless rendering works.** You can generate images of the robot without a display, using EGL or OSMesa as the rendering backend. This is the pipeline that will produce evaluation videos in later chapters.
- **The training loop works.** A complete cycle -- environment interaction, policy forward pass, gradient computation, parameter update, checkpoint save -- executes without error. The checkpoint is loadable by SB3.
- **The reward invariant holds.** `compute_reward(achieved_goal, desired_goal, info)` matches the reward returned by `env.step()`. This is the foundation for Hindsight Experience Replay (HER) in Chapter 5.

You also have a set of habits that will carry through the entire book: the three diagnostic questions (can it be solved? is it reliable? is it stable?), the experiment contract (artifacts on disk that you can inspect and share), and the willingness to inspect the environment directly rather than trusting that "it probably works."

What comes next is where things get interesting. The PPO smoke test ran for only 50,000 steps -- enough to verify the loop, not enough to learn. The remaining chapters will show you how to read training diagnostics (what does a healthy reward curve look like?), how to evaluate policies rigorously (across how many seeds? with deterministic or stochastic actions?), and how to tell whether a flat reward curve means "broken" or "needs more time."

Chapter 2 takes a deeper look at what the robot actually sees -- inspecting all observation components, understanding what the action space means physically, exploring reward semantics for both dense and sparse variants, and establishing the metrics schema (success rate, mean return, goal distance, action smoothness) that you will use for evaluation in every later chapter. Where this chapter asked "does the environment work?", Chapter 2 asks "do we understand what the environment is telling us?"

Reproduce It

```
------------------------------------------------------------
REPRODUCE IT
------------------------------------------------------------
The artifacts in this chapter come from this run:
```

```
   bash docker/dev.sh python scripts/ch00_proof_of_life.py all

Hardware:       Any machine with Docker (GPU optional)
Time:           ~5 min (GPU) / ~10 min (CPU)

Artifacts produced:
  smoke_frame.png
  ppo_smoke.zip
  ppo_smoke.meta.json

Results summary:
  gpu-check:    PASS (CUDA available on NVIDIA; CPU fallback on Mac)
  list-envs:    PASS (Fetch environments listed)
  render:       PASS (smoke_frame.png created, shows Fetch robot)
  ppo-smoke:    PASS (ppo_smoke.zip created, loadable by PPO.load())
                PASS (ppo_smoke.meta.json created)

This chapter's pipeline is fast enough to run in full
every time. No checkpoint track is needed.
------------------------------------------------------------
```

Exercises

1. Change the seed and confirm reproducibility.

Run the proof-of-life pipeline with a different seed:

```
bash docker/dev.sh python scripts/ch00_proof_of_life.py all --seed 7
```

Confirm that the artifacts (smoke_frame.png, ppo_smoke.zip, and ppo_smoke.meta.json)
are still produced. The rendered frame should look slightly different (the goal position
is randomized), but the robot and table should be identical.

2. Force a rendering backend.

By default, the script tries EGL first, then falls back to OSMesa. Force each backend
explicitly and document what happens on your system. We set the variable inside the
container (after bash -c) to ensure it takes effect, since docker/dev.sh sets its own
MUJOCO_GL value:

```
# Force EGL
bash docker/dev.sh bash -c 'MUJOCO_GL=egl python scripts/ch00_proof_of_life.py ren

# Force OSMesa
bash docker/dev.sh bash -c 'MUJOCO_GL=osmesa python scripts/ch00_proof_of_life.py
```

Which backend works on your machine? If only one works, note which one and why
(hint: EGL requires NVIDIA drivers; OSMesa is software-only). The rendered image
should be identical regardless of backend.

3. Inspect the observation dictionary across multiple resets.

Write a short script that creates `FetchReachDense-v4`, resets it 100 times, and for each reset prints or accumulates:

- The shape and dtype of each observation component
- The min and max values across all resets for each array
- Whether any values are NaN or infinite

This gives you a feel for the value ranges that the policy network will need to handle. Expect all values to be finite and float64. The `observation` array contains positions in meters and velocities in meters/second -- typical ranges are roughly [-0.1, 0.1] for velocities and [1.0, 1.5] for positions. The `desired_goal` should cover the workspace bounds (roughly x in [1.05, 1.55], y in [0.40, 1.10], z in [0.42, 0.60]) with uniform-looking coverage. If the goals are clustered or identical across resets, the environment may not be randomizing correctly.

This is also a good opportunity to check that different seeds produce different goal positions (they should -- the goal is sampled uniformly within the workspace at each reset).

4. (Challenge) Modify the success threshold.

The default success threshold for FetchReach is 0.05 meters (5 cm). What happens if you change it? The threshold is defined in the environment's XML configuration. Explore:

- Can you find where the threshold is set? (Hint: look at the `distance_threshold` parameter in the Gymnasium-Robotics source code, or check `env.unwrapped.distance_thres` after creating the environment.)
- If you made the threshold larger (say, 0.10 m), would a random policy succeed more often? Estimate by running 100 random episodes and checking `is_success` at the final step.
- If you made it smaller (say, 0.01 m), what would happen to training difficulty? Consider: how precisely would the end-effector need to be positioned?

You do not need to actually modify the threshold to answer these questions -- running random episodes and measuring goal distances will give you the intuition. We will revisit this question when we discuss sparse rewards in Chapter 5, where the threshold directly determines how much of the goal space produces zero reward.