2 Environment Anatomy: What the Robot Sees, Does, and Learns From

Vlad Prytula

2026-02-17

Contents

2 Environment Anatomy: What the Robot Sees, Does, and Learns From

This chapter covers:

- Inspecting the dictionary observation structure that every Fetch environment returns -- what each number means physically and why observations are dictionaries, not flat vectors
- Understanding what the 4D action vector controls: Cartesian deltas for the end-effector and a gripper open/close command
- Verifying reward computation for both dense (distance-based) and sparse (binary success/failure) variants, and proving the critical invariant that makes Hindsight Experience Replay possible
- Simulating goal relabeling by hand -- calling `compute_reward` with goals the environment never intended -- to see why the Fetch interface enables HER
- Establishing a random-policy baseline (success rate, mean return, goal distance) that every trained agent in later chapters must beat

In Chapter 1, you verified that the computational stack works: Docker launches, MuJoCo simulates, the rendering pipeline produces frames, and a training loop runs to completion. You have a proof of life -- evidence that the environment is alive and capable of producing results.

But alive is not the same as understood. You know the environment *works*, but not what it *says*. What do the 10 numbers in the observation vector mean? What happens when the robot takes action `[1, 0, 0, 0]`? Why does the reward function return -0.073 instead of -1? How does the environment know the robot ”succeeded”? Without answering these questions, you cannot debug training failures or choose appropriate algorithms.

This chapter provides a complete anatomy of Fetch environment observations, actions, rewards, and goals. You will inspect every component by hand, verify reward computation against the distance formula, simulate HER-style goal relabeling, and establish the random-policy baseline that every trained agent must beat. With the environment understood, Chapter 3 trains a real policy -- PPO on dense Reach. The observation shapes you document here determine the network architecture. The random baseline you establish here is the floor that PPO must exceed.

2.1 WHY: Why environment anatomy matters

Imagine you train a policy for 10 hours. The training loop completes without errors. You evaluate the checkpoint and find a success rate of 0%. You check the reward curve -- flat. You check the loss -- it looks normal. Nothing crashed. What went wrong?

Without understanding what the agent sees and what rewards mean, you cannot answer this question. The problem could be anywhere: the observations might have unexpected scales, the rewards might have the wrong sign, the success threshold might be different from what you assumed, or the goal structure might not match what the algorithm expects. You trained for 10 hours, but you never checked whether the environment's outputs make sense for the algorithm you chose.

This is not hypothetical. In our experience, environment misunderstandings are the single most common source of wasted compute in robotics RL. Not bad hyperparameters, not wrong algorithms -- wrong assumptions about what the environment is actually doing.

Three questions you cannot answer without anatomy

Here are three questions that come up in every training failure. Try to answer them without inspecting the environment:

1. ”Is the observation what the network expects?” When you create a policy with SB3's `MultiInputPolicy`, it reads the observation space to determine its input structure. If the observation is a dictionary with three keys, the network builds three separate encoders and concatenates them. If the observation were somehow a flat vector (due to a wrapper or version mismatch), the network architecture would be completely different -- and wrong for the task. You need to know exactly what the observation contains to understand what the network receives.

2. ”Is the reward on a reasonable scale?” Dense Fetch rewards are negative distances -- typical values in the range -0.01 to -0.3 for FetchReach. If you tuned your learning rate assuming rewards on the order of -1 to 0 (as with sparse rewards), your value function targets would be off by an order of magnitude. Reward scale affects everything: the value function's target range, the entropy coefficient in SAC, and the advantage normalization in PPO.

3. ”Can this environment support HER?” Hindsight Experience Replay -- which we introduce in Chapter 5 -- requires two specific properties from the environment: an explicit `achieved_goal` in the observation, and a `compute_reward` function that accepts arbitrary goals. If either is missing, HER cannot work. You might spend days trying to make HER work on an environment that does not support it, never realizing the problem is structural, not algorithmic.

What misunderstandings cost

To be concrete about the cost, here is a table of misunderstandings we have seen (some in our own work, some reported by others) and what they led to:

Misunderstanding	What happens
Assumed observations are flat vectors	Network architecture wrong
Did not know `achieved_goal` tracks object (not gripper) in FetchPush	Reward calculations wrong;
Assumed sparse reward is -1/+1 (it is -1/0)	Value function targets off;
Did not verify `compute_reward` matches `env.step()`	HER learns from corrupted
Did not check success threshold	Evaluated with wrong succ

The common thread: every one of these is preventable by inspecting the environment before training. This chapter does that inspection systematically.

The anatomy as a debugging foundation

There is a positive way to frame this too. When you understand what the environment returns, debugging becomes tractable. A flat reward curve is no longer a mystery -- you can check: does the policy's output fall within the action space bounds? Is the achieved goal moving in response to actions? Is the reward decreasing (getting closer to zero) even if success rate has not budged? Is `compute_reward` returning the same values as `env.step()`?

Each of these diagnostic checks has a specific prerequisite:

- **Checking action bounds** requires knowing the action space shape and range (section 2.4)
- **Checking goal movement** requires knowing what `achieved_goal` tracks -- gripper position for Reach, object position for Push (section 2.3)
- **Checking reward trends** requires knowing the reward formula and its range -- dense rewards are negative distances, sparse rewards are 0 or -1 (section 2.5)
- **Checking the `compute_reward` invariant** requires knowing the API signature and how it relates to `env.step()` (section 2.5)

These are not abstract skills. In Chapter 3, when PPO training stalls, the first thing you will do is check whether the reward is moving. In Chapter 5, when HER does not improve performance, the first thing you will do is check whether `compute_reward` handles relabeled goals correctly. The anatomy you learn here is the diagnostic toolkit for every later chapter.

The rest of this chapter gives you that knowledge, piece by piece. We start with what the agent sees (observations and goals), move to what it can do (actions), then to how performance is measured (rewards), and finally to the key insight that ties it all together (goal relabeling).

The goal-conditioned MDP: seven pieces

The Fetch environments implement a specific mathematical structure called a *goal-conditioned Markov Decision Process* (GCMDP). Here is what we mean by that -- not as a formal theorem, but as a concrete description of the interface you will work with.

A goal-conditioned MDP has seven pieces:

- **States** ($\mathcal{S}$): the full physical state of the robot and any objects on the table
- **Actions** ($\mathcal{A}$): what the robot can do (4D Cartesian deltas plus gripper)
- **Goals** ($\mathcal{G}$): the target positions the robot must achieve (3D Cartesian coordinates)
- **Transitions** (P): the physics -- how the state changes when the robot acts
- **Rewards** (R): the feedback signal, which depends on the goal
- **Goal-achievement mapping** (ϕ): a function that extracts ”what goal did we achieve?” from the current state
- **Discount factor** (γ): how much we value future rewards versus immediate ones

The critical insight is that the reward depends on the goal. The same state might yield reward -0.1 for one goal (you are close) and -0.5 for another (you are far away). And the goal-achievement mapping ϕ tells us what goal we *actually* achieved, regardless of what goal we were aiming for. These two properties -- goal-dependent rewards and an explicit achieved goal -- are what make Hindsight Experience Replay possible. We will see exactly why in section 2.6.

2.2 The Fetch task family

Chapter 1 introduced the four Fetch tasks: Reach, Push, PickAndPlace, and Slide. Here we add the physical details that matter for environment anatomy.

The simulated robot. The Fetch arm is a 7-degree-of-freedom manipulator modeled in MuJoCo, a rigid-body physics simulator. MuJoCo runs the physics at 500 Hz internally; the environment exposes control at 25 Hz (every 20 simulation steps). When you call `env.step(action)`, MuJoCo simulates 20 timesteps of physics, then returns the resulting state.

The workspace. The arm sits on a fixed base next to a table. The workspace -- the region where goals are sampled and the end-effector can reach -- spans roughly:

```
Fetch Workspace (approximate bounds in meters):

  z (height)
  0.6 |  +--------------------------+
      |  |   goal region (air)      |
  0.5 |  |                          |
      |  +--------------------------+ <-- table surface ~0.42
  0.4 |
      +----+----+----+----+----+----> x (forward)
         1.0  1.1  1.2  1.3  1.4  1.5

  y (sideways) spans roughly 0.4 to 1.1
```

These numbers matter because they give you a sense of scale. The entire workspace is about 60 cm x 70 cm x 20 cm. The success threshold is 5 cm (0.05 m), which means the end-effector must be within about 8% of the workspace width to ”succeed.” Random flailing is unlikely to land within 5 cm of an arbitrary target -- which is why random baselines have near-zero success rates.

Dense vs. sparse rewards. Each task comes in two reward variants. The environment ID encodes which: `FetchReachDense-v4` uses dense rewards (negative distance to goal), while `FetchReach-v4` uses sparse rewards (0 if goal reached, -1 otherwise). The observation structure and action space are identical between variants -- only the reward computation differs.

For this chapter, we work primarily with `FetchReachDense-v4` (dense Reach). It is the simplest variant: no objects, continuous feedback, and a workspace that the arm can easily cover. The anatomy principles transfer directly to Push, PickAndPlace, and Slide -- the interface is the same, and we verify that uniformity in section 2.7.

Episode structure. Every episode lasts exactly 50 steps (the environment truncates at this limit). At each step, the agent observes the state, chooses an action, and receives a reward. The episode ends when the step limit is reached -- there is no early termination on success. This means that even a successful policy continues to act for the full 50 steps, which affects how you interpret returns: a policy that reaches the goal on step 10 still receives rewards for the remaining 40 steps. For dense rewards, those remaining steps contribute small negative values (the agent is near the goal). For sparse rewards, they contribute 0s (the agent has already succeeded).

2.3 Build It: The observation dictionary

The first thing to understand about any RL environment is what the agent perceives. In Fetch environments, the agent does not see a flat vector. It sees a dictionary with three keys, each carrying a different kind of information. This structure is not just a data format -- it is the interface through which goal-conditioned learning operates.

What the dictionary contains

The observation dictionary has three entries:

- `obs["observation"]` -- the robot's proprioceptive state. For FetchReach, this is a 10-dimensional vector containing the gripper's position, velocity, and finger state.
- `obs["achieved_goal"]` -- where the relevant thing (end-effector for Reach, object for Push) currently is. Always 3D Cartesian coordinates.
- `obs["desired_goal"]` -- where we want that thing to be. Also 3D Cartesian coordinates.

The separation of achieved and desired goals from the main observation is the key design decision. It makes the goal-conditioning explicit: the environment tells the agent ”here is your state, here is where you are, here is where you need to be.”

Why not a flat vector? Many RL environments (CartPole, Atari, MuJoCo locomotion) return a single array as the observation. Fetch environments *could* concatenate everything into a flat vector of length 16 (for Reach) or 31 (for Push). But the dictionary structure serves two purposes. First, it makes the goal-conditioned interface explicit -- the achieved and desired goals are labeled, not buried at arbitrary indices in a flat vector. Second, it enables SB3's `MultiInputPolicy` to process each component through a separate encoder before concatenation, which gives the network a natural way to compare ”where I am” against ”where I should be.”

Inspecting the observation

> **Note:** All code listings in this chapter assume `import gymnasium as gym`, `import gymnasium_robotics`, and `import numpy as np`. The full runnable versions live in `scripts/labs/env_anatomy.py`.

Let's look at the code that inspects this structure:

```python
# Observation dictionary inspector
# (adapted from scripts/labs/env_anatomy.py:obs_inspector)

def obs_inspector(env_id="FetchReachDense-v4", seed=0):
    """Inspect obs dict: keys, shapes, dtypes, workspace bounds."""
    env = gym.make(env_id)
    obs, info = env.reset(seed=seed)
    summary = {}
    for key in ["observation", "achieved_goal", "desired_goal"]:
        arr = np.asarray(obs[key])
        summary[key] = {
            "shape": arr.shape, "dtype": str(arr.dtype),
            "min": float(arr.min()), "max": float(arr.max()),
            "finite": bool(np.all(np.isfinite(arr))),
        }
    # Check desired_goal is within workspace
    dg = np.asarray(obs["desired_goal"])
    summary["desired_goal_in_workspace"] = bool(
        1.1 <= dg[0] <= 1.5 and 0.5 <= dg[1] <= 1.0
        and 0.35 <= dg[2] <= 0.75
    )
    env.close()
    return summary
```

> **Checkpoint.** Run bash `docker/dev.sh` python `scripts/labs/env_anatomy.py --verify` and check the first output section. You should see:
>
> - `obs=(10,)`, `ag=(3,)`, `dg=(3,)`
> - All values finite
> - `desired_goal in workspace: True`
>
> If the observation shape is not `(10,)`, check that you are using `FetchReachDense-v4` (not a Push or PickAndPlace variant, which have 25D observations).

The 10D observation breakdown

What do those 10 numbers actually represent? Here is the breakdown for FetchReach:

Index	Semantic	Units	Typical range
0-2	Gripper position (x, y, z)	meters	[1.1, 1.5], [0.5, 1.0], [0.35, 0.75]
3-4	Gripper finger positions (right, left)	meters	[0.0, 0.05]
5-7	Gripper linear velocity (dx, dy, dz)	m/s	[-0.1, 0.1]
8-9	Gripper finger velocities (right, left)	m/s	[-0.01, 0.01]

For FetchReach, the first three elements of `obs["observation"]` -- the gripper position -- are the same as `obs["achieved_goal"]`. This is because the goal-achievement mapping ϕ for reaching is: ”where is the end-effector?” We can verify this:

The goal space: where goals live

The goal-achievement mapping ϕ extracts the ”achieved goal” from the current state. For Reach, $\phi(s)$ = gripper position. For Push, $\phi(s)$ = object position. This mapping is what determines achieved_goal in the observation dictionary.

```python
# Goal space verification
# (adapted from scripts/labs/env_anatomy.py:goal_space)

def goal_space(env_id="FetchReachDense-v4", n_resets=100, seed=0):
    """Sample goals via reset; check bounds; verify phi(s)=obs[:3]."""
    env = gym.make(env_id)
    desired_goals, phi_matches = [], []
    for i in range(n_resets):
        obs, _ = env.reset(seed=seed + i)
        desired_goals.append(np.asarray(obs["desired_goal"]))
        ag = np.asarray(obs["achieved_goal"])
        obs_vec = np.asarray(obs["observation"])
        # phi(s) = grip_pos = obs["observation"][:3] for FetchReach
        phi_matches.append(bool(np.allclose(ag, obs_vec[:3])))
    dg_arr = np.stack(desired_goals)
    env.close()
    return {
        "n_resets": n_resets,
        "goal_dim": int(dg_arr.shape[1]),
        "desired_goal_bounds": {
            "min": dg_arr.min(axis=0).tolist(),
            "max": dg_arr.max(axis=0).tolist(),
        },
        "all_phi_match": all(phi_matches),
    }
```

Checkpoint. The verification output should show:

- `goal_dim=3`
- `phi matches obs[:3]: True` (all 100 resets)
- `desired_goal range:` x roughly [1.05, 1.55], y roughly [0.40, 1.10], z roughly [0.42, 0.60]

If `phi matches obs[:3]` is False for any reset, there is a version mismatch in gymnasium-robotics. If the goal range is very narrow (all coordinates nearly identical), the environment is not randomizing goals correctly -- check your seed handling.

What the policy network will see

When we create an SB3 model with `MultiInputPolicy` in Chapter 3, the network reads this dictionary observation space and builds:

Network input	Source key	Dimensions
State encoder	`observation`	10
Achieved goal encoder	`achieved_goal`	3
Desired goal encoder	`desired_goal`	3
Total input	(concatenated)	**16**

The encoders process each key through separate MLP layers, then concatenate the results before passing through shared layers. The total input dimension -- 16 for FetchReach, 31 for FetchPush -- determines the network's capacity requirements. This is why we document shapes carefully: they directly affect architecture.

2.4 Build It: Action semantics

The agent does not control joint torques. Instead, it outputs a 4-dimensional vector of Cartesian commands that an internal controller translates into joint movements. This is an important design choice: the learning algorithm does not need to solve inverse kinematics, which makes the learning problem more tractable.

What each action dimension controls

Index	Semantic	Range	Physical effect
0	dx	[-1, 1]	End-effector moves in the x direction (forward/backward)
1	dy	[-1, 1]	End-effector moves in the y direction (left/right)
2	dz	[-1, 1]	End-effector moves in the z direction (up/down)
3	gripper	[-1, 1]	< 0 opens, > 0 closes the parallel-jaw gripper

Actions are clipped to [-1, 1] and then scaled by the environment's internal controller before being applied as forces. The scaling means that action `[1, 0, 0, 0]` does not move the end-effector by exactly 1 meter -- it applies a maximal positive displacement command along x, and the actual movement depends on the physics simulation (friction, inertia, joint limits).

Verifying action-to-movement mapping

We can verify that each action axis produces movement in the expected direction:

```python
# Action space explorer
# (adapted from scripts/labs/env_anatomy.py:action_explorer)

def action_explorer(env_id="FetchReachDense-v4", seed=0):
    """Step with axis-aligned actions, measure displacement."""
    env = gym.make(env_id)
    results = {
        "action_shape": env.action_space.shape,
        "action_low": env.action_space.low.tolist(),
        "action_high": env.action_space.high.tolist(),
        "displacements": {},
    }
    axis_names = ["x", "y", "z"]
    for axis_idx in range(3):
        obs, _ = env.reset(seed=seed)
        grip_before = np.asarray(obs["achieved_goal"]).copy()
        action = np.zeros(4, dtype=np.float32)
        action[axis_idx] = 1.0
        obs, _, _, _, _ = env.step(action)
        grip_after = np.asarray(obs["achieved_goal"])
        disp = grip_after - grip_before
        results["displacements"][axis_names[axis_idx]] = {
            "action": action.tolist(),
            "displacement": disp.tolist(),
            "moved_positive": bool(disp[axis_idx] > 0),
        }
    env.close()
    return results
```

> **Checkpoint.** The verification output should show:
>
> - `action_shape=(4,)`, bounds `[-1, 1]`
> - Action `[1,0,0,0]` produces positive x displacement
> - Action `[0,1,0,0]` produces positive y displacement
> - Action `[0,0,1,0]` produces positive z displacement
>
> The displacement magnitudes are small (typically 0.005-0.02 meters per step) because the action is applied for one 25 Hz control step. At 50 steps per episode, the end-effector can traverse roughly 0.25-1.0 meters total -- enough to cover the workspace.

For FetchReach, the gripper dimension (index 3) has no effect on the task -- there is no object to grasp. For Push and PickAndPlace, it becomes essential: the agent must learn to close the gripper around an object and open it at the target location.

> **Warning:** A subtle point about action semantics: the policy outputs continuous values in [-1, 1], but the mapping from these values to physical displace-

ment is not linear and depends on MuJoCo's internal controller. An action of [0.5, 0, 0, 0] does not produce exactly half the displacement of [1.0, 0, 0, 0]. In practice, this does not matter for training -- the policy learns the mapping implicitly -- but it means you should not try to hand-code a controller using these action values as if they were calibrated velocity commands.

2.5 Build It: Reward computation -- dense and sparse

Now that we know what the agent sees and what it can do, we need to understand how the environment evaluates performance. Fetch environments provide two reward variants, and understanding both is essential for the rest of the book.

Dense reward: negative distance

The dense reward for any Fetch environment is:

$$R_{\text{dense}} = -\|g_a - g_d\|_2$$

where g_a is the achieved goal (a 3D position), g_d is the desired goal (also 3D), and $\|\cdot\|_2$ is the Euclidean norm. The reward is always negative or zero, with zero meaning perfect goal achievement. A reward of -0.1 means the relevant point (end-effector for Reach, object for Push) is 10 cm from the target.

This reward provides continuous gradient information: every action that moves closer to the goal produces a less negative reward. Standard policy gradient methods like PPO can learn effectively from this signal because there is always a direction to improve.

Verifying the dense reward formula

We verify that three things match: the manual distance formula, the `compute_reward` API, and the reward returned by `env.step()`.

```python
# Dense reward check
# (adapted from scripts/labs/env_anatomy.py:dense_reward_check)

def dense_reward_check(env_id="FetchReachDense-v4", n_steps=100,
                       seed=0, atol=1e-10):
    """Verify R = -||ag - dg|| matches step_reward and compute_reward."""
    env = gym.make(env_id)
    obs, info = env.reset(seed=seed)
    mismatches = 0
    for t in range(n_steps):
        action = env.action_space.sample()
        obs, step_reward, terminated, truncated, info = env.step(action)
        ag = np.asarray(obs["achieved_goal"])
        dg = np.asarray(obs["desired_goal"])
        manual = -float(np.linalg.norm(ag - dg))
        cr = float(
            env.unwrapped.compute_reward(ag, dg, info))
        step_r = float(step_reward)
        if abs(manual - cr) > atol or abs(manual - step_r) > atol:
            mismatches += 1
        if terminated or truncated:
            obs, info = env.reset(seed=seed + t + 1)
    env.close()
    return {"n_steps": n_steps, "mismatches": mismatches}
```

This three-way comparison is the heart of the verification. We compute the reward ourselves (`manual`), ask the environment to compute it (`cr`), and compare both to what `env.step()` returned (`step_r`). All three must agree.

Why do we check all three? Because each catches a different failure mode. If `manual` differs from `cr`, our understanding of the reward formula is wrong -- maybe the reward is not just negative distance, but includes a scaling factor or an offset. If `cr` differs from `step_r`, there is a bug in the environment or a version mismatch between the wrapper and the base environment. If `manual` matches `cr` but not `step_r`, the wrapper is modifying the reward (perhaps adding a penalty term). Any of these mismatches would corrupt HER's relabeling in Chapter 5.

> **Checkpoint.** Over 100 steps: zero mismatches, tolerance 1e-10. The three values should match to floating-point precision. If they do not match, you likely have a version mismatch between gymnasium and gymnasium-robotics.

Sparse reward: binary success signal

The sparse reward uses a distance threshold $\epsilon = 0.05$ meters (5 cm):

$$R_{\text{sparse}} = \begin{cases} 0 & \text{if } \|g_a - g_d\|_2 \leq \epsilon \\ -1 & \text{otherwise} \end{cases}$$

Note the values: 0 for success, -1 for failure. Not +1 for success. Not -1/+1. The reward is always non-positive. This matters for value function initialization and for understanding what a ”good” return looks like.

Let's trace through the numbers. With sparse rewards, an episode of 50 steps where the goal is never reached has a return of $\sum_{t=0}^{49}(-1) = -50$. An episode where the goal is reached on step 40 and maintained for the remaining 10 steps has a return of $40 \times (-1) + 10 \times 0 = -40$. A perfect policy that reaches the goal immediately would have a return of $-1 \times (\text{steps to reach goal}) + 0 \times (\text{remaining steps})$. The best possible return on a 50-step episode is 0 (goal reached on step 0), but in practice even good policies take a few steps to reach the goal, so returns of -2 to -5 indicate strong performance.

For dense rewards, the numbers are different but the reasoning is similar. The return is the sum of negative distances across all 50 steps. A random policy averages about

-15 to -25 per episode. A well-trained policy that quickly reaches the goal and stays there might achieve -1 to -3.

Verifying the sparse reward formula

```
# Sparse reward check
# (adapted from scripts/labs/env_anatomy.py:sparse_reward_check)

def sparse_reward_check(env_id="FetchReach-v4", n_steps=100,
                        seed=0, atol=1e-10):
    """Verify R = 0 if ||ag-dg|| <= eps else -1; check threshold."""
    env = gym.make(env_id)
    eps = float(env.unwrapped.distance_threshold)
    obs, info = env.reset(seed=seed)
    mismatches, non_binary = 0, 0
    for t in range(n_steps):
        action = env.action_space.sample()
        obs, step_reward, terminated, truncated, info = env.step(action)
        ag = np.asarray(obs["achieved_goal"])
        dg = np.asarray(obs["desired_goal"])
        dist = float(np.linalg.norm(ag - dg))
        expected = 0.0 if dist <= eps else -1.0
        step_r = float(step_reward)
        cr_r = float(
            env.unwrapped.compute_reward(ag, dg, info))
        if step_r not in (0.0, -1.0):
            non_binary += 1
        if (abs(step_r - expected) > atol
                or abs(cr_r - expected) > atol):
            mismatches += 1
        if terminated or truncated:
            obs, info = env.reset(seed=seed + t + 1)
    env.close()
    return {"n_steps": n_steps, "threshold": eps,
            "mismatches": mismatches, "non_binary": non_binary}
```

Checkpoint. Over 100 steps:

- `threshold=0.05`
- `mismatches=0`
- `non_binary=0` (every sparse reward is exactly 0.0 or -1.0)

If the threshold is not 0.05, your gymnasium-robotics version may use a different default. If non-binary count is positive, something unexpected is happening with the reward computation.

The critical invariant

Both reward checks verify the same fundamental property -- the *critical invariant* for the entire book:

> The reward returned by `env.step()` must equal `env.unwrapped.compute_reward(achieved_g`
> `desired_goal, info)`.

This is not paranoid caution. Different versions of gymnasium-robotics have had bugs affecting reward computation, and API changes have altered the signature of `compute_reward`. Running these checks ensures that your specific installation behaves correctly.

More importantly, this invariant is the foundation of Hindsight Experience Replay. When HER relabels a trajectory with a different goal, it calls `compute_reward` with the new goal to get the relabeled reward. If `compute_reward` disagrees with `env.step()`, HER trains on incorrect labels. The policy would learn from corrupted data, and training could fail silently -- no error, no crash, just a policy that does not work.

> **Tip:** If you are ever unsure whether the reward invariant holds after upgrading gymnasium-robotics, run the quick check: `bash docker/dev.sh python scripts/labs/env_anatomy.py --verify`. The dense and sparse reward checks (components 4 and 5) specifically verify this invariant. It takes less than a minute and can save you days of debugging corrupted HER training.

2.6 Build It: Goal relabeling simulation

Everything we have built up to -- the dictionary observations, the separate achieved and desired goals, the `compute_reward` function -- converges here. We are going to do something that seems odd at first: call `compute_reward` with goals that the environment never set.

Why relabeling matters

The problem with sparse rewards is simple: if the agent never reaches the goal, the reward is -1 at every step of every episode. There is no gradient signal pointing toward success. The agent has no way to distinguish between a near-miss (3 cm from the goal) and a complete failure (30 cm away). Both get reward -1.

Hindsight Experience Replay (HER), which we cover in depth in Chapter 5, solves this by asking: ”You failed to reach the goal, but you *did* reach some position. What if that position had been the goal?” HER takes the `achieved_goal` from a failed trajectory and retroactively pretends it was the `desired_goal`. Then it recomputes the reward -- and because the agent actually reached that position, the recomputed reward is 0 (success).

For this to work, the environment must let us call `compute_reward` with arbitrary goals -- not just the goal that was originally set. Let's verify that this works:

Simulating relabeling by hand

```python
# Goal relabeling check
# (adapted from scripts/labs/env_anatomy.py:relabel_check)

def relabel_check(env_id="FetchReachDense-v4", n_goals=10,
                  seed=0, atol=1e-10):
    """Call compute_reward with arbitrary goals (HER simulation)."""
    env = gym.make(env_id)
    obs, info = env.reset(seed=seed)
    goal_sp = env.observation_space.spaces["desired_goal"]
    # Take one step to get a non-trivial achieved_goal
    obs, _, _, _, info = env.step(env.action_space.sample())
    ag = np.asarray(obs["achieved_goal"])
    reward_type = getattr(
        env.unwrapped, "reward_type", "dense")
    eps = float(env.unwrapped.distance_threshold)
    mismatches = 0
    for _ in range(n_goals):
        random_goal = goal_sp.sample()
        cr = float(
            env.unwrapped.compute_reward(ag, random_goal, info))
        dist = float(np.linalg.norm(ag - random_goal))
        expected = (-dist if reward_type == "dense"
                    else (0.0 if dist <= eps else -1.0))
        if abs(cr - expected) > atol:
            mismatches += 1
    env.close()
    return {"n_goals": n_goals, "mismatches": mismatches}
```

Here is what this code does, step by step:

1. **Take a step** to get a non-trivial achieved goal (the gripper moved somewhere).
2. **Sample 10 random goals** from the goal space -- positions the environment never intended as targets.
3. **Call `compute_reward`** with the actual achieved goal and each random goal.
4. **Verify** that the returned reward matches the distance formula.

This is exactly what HER does, in miniature. The agent ”failed” (it did not reach the original desired goal), but we can ask: ”what would the reward have been if the goal were somewhere else?” And `compute_reward` gives us a correct answer.

> **Checkpoint.** Over 10 random goals: zero mismatches. `compute_reward` correctly handles arbitrary goals -- not just the one the environment set. No errors, no NaN, no crashes.
>
> This proves the key property for HER: **we can recompute rewards for any goal without re-running the simulation.**

Why this works (and why it would not work everywhere)

The reason `compute_reward` accepts arbitrary goals is that the Fetch reward depends on the goal only through the distance $\|g_a - g\|$. The function does not need to access the physics state, the action history, or any internal simulation data. It takes two 3D vectors and computes a distance. This is why relabeling is cheap and exact.

Not all environments have this property. An environment where the reward depends on the trajectory (not just the endpoint), or where ”success” requires a specific sequence of actions, would not support this kind of relabeling. An environment that returns flat observations with no goal separation does not expose the structure HER needs -- you would not know what ”achieved goal” to relabel with, and you would have no function to recompute rewards for a different goal.

The Fetch interface was designed with HER in mind -- the dictionary observations, the separate goals, and the standalone `compute_reward` are all part of that design. The original HER paper (Andrychowicz et al., 2017) introduced these environments alongside the algorithm, specifically to provide a test bed where the relabeling mechanism works cleanly.

> **Note:** For a full treatment of HER's mechanism and the conditions under which it applies, see Chapter 5. For now, the important takeaway is practical: you have verified with your own hands that the Fetch environment supports goal relabeling, and you know exactly what that means -- calling `compute_reward` with goals the environment never set, and getting correct rewards back.

2.7 Build It: Cross-environment comparison

So far we have focused on FetchReach. But the Fetch family includes Push, PickAndPlace, and Slide, and the interface generalizes across all of them -- with important differences in the details.

How observations change across tasks

The key difference: environments with objects have larger observation vectors because they include the object's state (position, rotation, velocity).

```python
# Cross-environment comparison
# (adapted from scripts/labs/env_anatomy.py:cross_env_compare)

def cross_env_compare(seed=0):
    """Compare obs dims across Reach, Push, PickAndPlace."""
    env_configs = {
        "FetchReach-v4": {"expected_obs_dim": 10,
                          "ag_is_grip": True},
        "FetchPush-v4": {"expected_obs_dim": 25,
                         "ag_is_grip": False},
        "FetchPickAndPlace-v4": {"expected_obs_dim": 25,
                                 "ag_is_grip": False},
    }
    results = {}
    for env_id, cfg in env_configs.items():
        env = gym.make(env_id)
        obs, _ = env.reset(seed=seed)
        obs_vec = np.asarray(obs["observation"])
        ag = np.asarray(obs["achieved_goal"])
        grip_pos = obs_vec[:3]
        ag_matches_grip = bool(np.allclose(ag, grip_pos))
        results[env_id] = {
            "obs_dim": obs_vec.shape[0],
            "ag_matches_grip": ag_matches_grip,
        }
        env.close()
    return results
```

Checkpoint. Expected output:

- `FetchReach-v4`: `obs_dim=10`, `ag_matches_grip=True`
- `FetchPush-v4`: `obs_dim=25`, `ag_matches_grip=False`
- `FetchPickAndPlace-v4`: `obs_dim=25`, `ag_matches_grip=False`

For Push and PickAndPlace, achieved_goal is the **object position**, not the gripper position. This is because the task goal is about where the object ends up, not where the gripper is. The gripper position is still available in `obs["observation"][:3]`.

The 25D observation for manipulation tasks

Environments with objects (Push, PickAndPlace) include 15 additional dimensions:

Index	Semantic	Source
0-2	Gripper position	`grip_pos`
3-5	Object position	`object_pos`
6-8	Object relative position (object - gripper)	`object_rel_pos`
9-10	Gripper finger positions	`gripper_state`
11-13	Object rotation (Euler angles)	`object_rot`
14-16	Object linear velocity (relative to gripper)	`object_velp`
17-19	Object angular velocity	`object_velr`
20-22	Gripper linear velocity	`grip_velp`
23-24	Gripper finger velocities	`gripper_vel`

The crucial thing to notice: when `achieved_goal` changes from tracking the gripper (Reach) to tracking the object (Push, PickAndPlace), the goal-achievement mapping ϕ changes. For Reach, $\phi(s)$ = gripper position. For Push, $\phi(s)$ = object position. The environment handles this transparently -- you do not need to change your code. But you need to understand it, because it affects what ”success” means: in Push, the agent succeeds when the *object* (not the gripper) is within 5 cm of the target.

Uniform interface, changing semantics

Despite the dimension differences, the *interface* is identical across all Fetch environments:

Property	FetchReach	FetchPush	FetchPickAndPlace
`obs["observation"]` dim	10	25	25
`obs["achieved_goal"]` dim	3	3	3
`obs["desired_goal"]` dim	3	3	3
Action dim	4	4	4
`compute_reward` API	same	same	same
`distance_threshold`	0.05	0.05	0.05

This uniformity is what lets us develop algorithms on Reach (fast iteration, easy debugging) and then apply them to harder tasks. The same `MultiInputPolicy`, the same `compute_reward` invariant, the same HER relabeling -- all transfer directly. What changes is the task difficulty, not the interface.

In many RL domains, moving to a harder task means rewriting the environment wrapper, changing the observation preprocessing, and adjusting the reward function. In Fetch, you change one string -- the environment ID -- and everything else stays the same. The code that trains PPO on FetchReachDense-v4 in Chapter 3 will train SAC on FetchPush-v4 in Chapter 5 with no structural changes. The only differences will be algorithmic (SAC instead of PPO, HER for goal relabeling) and in hyperparameters (more training steps for harder tasks).

2.8 Bridge: Manual inspection meets the production script

We have now inspected every major component of the Fetch environment by hand:

1. **Observations**: dictionary with three keys, shapes (10,), (3,), (3,) for Reach
2. **Actions**: 4D Cartesian deltas in [-1, 1], each axis produces movement in the expected direction
3. **Goals**: 3D Cartesian positions within the workspace; ϕ maps states to achieved goals
4. **Dense reward**: $-\|g_a - g_d\|$, matches `compute_reward` and `env.step()`
5. **Sparse reward**: 0 if distance $\leq$ 0.05, else -1, all three sources agree
6. **Relabeling**: `compute_reward` accepts arbitrary goals and returns correct rewards
7. **Cross-environment**: interface is uniform, observation dimensions and ϕ change with task complexity

The production script `scripts/ch01_env_anatomy.py` performs the same checks -- the same three-way reward comparison, the same relabeling test, the same structure verification -- but in an automated pipeline that produces JSON artifacts.

You can run the bridging proof to confirm that the manual lab code and the production script agree:

```bash
bash docker/dev.sh python scripts/labs/env_anatomy.py --bridge
```

Expected output:

```
Environment Anatomy -- Bridging Proof
============================================================
Env: FetchReachDense-v4, seed=42, steps=100
Relabeled goals per step: 5
Tolerance: atol=1e-10

Step reward checks:    100 steps, mismatches=0
Relabel reward checks: 500 checks, mismatches=0

[MATCH] All rewards match within atol=1e-10
  manual_reward == compute_reward == step_reward  (100 steps)
  relabel_reward == -||ag - random_goal||  (500 checks)

[BRIDGE OK] Bridging proof passed
```

100 steps, 5 relabeled goals per step = 500 total reward checks. Zero mismatches. The manual computation and the environment's `compute_reward` agree to within 1e-10 on every single check.

This bridging proof serves the same purpose as unit tests in software engineering: it confirms that your understanding (the manual computation) matches the implementation (the environment's API). When you run the production script in the next section and it reports ”OK,” you know exactly what ”OK” means -- because you have done the same checks by hand.

In later chapters, the bridging proof will be more dramatic: in Chapter 3, you will compare your from-scratch PPO loss computation against SB3's internal implementation. In Chapter 4, you will compare SAC update targets. Here, the bridge is simpler -- reward computation is just a distance calculation -- but the principle is the same: never trust a pipeline you have not verified by hand.

2.9 Run It: The inspection pipeline

Now we run the full production inspection. The script `scripts/ch01_env_anatomy.py` automates everything we have done by hand, producing machine-readable JSON artifacts.

```
---------------------------------------------------------
EXPERIMENT CARD: Fetch Environment Inspection
---------------------------------------------------------
Algorithm:    None (inspection and verification only)
Environment:  FetchReachDense-v4, FetchReach-v4, FetchPush-v4

Run command (full inspection):
  bash docker/dev.sh python scripts/ch01_env_anatomy.py all \
    --seed 0

Time:         < 2 min (CPU or GPU)

Checkpoint track:
  N/A (no training; all commands produce results in seconds)

Expected artifacts:
  results/ch01_env_describe.json
  results/ch01_random_metrics.json

Success criteria:
  ch01_env_describe.json exists, contains observation_space
    with keys: observation (shape [10]), achieved_goal (shape [3]),
    desired_goal (shape [3])
  reward-check prints "OK:" (zero mismatches across 500 steps)
  ch01_random_metrics.json exists, success_rate near 0.0-0.1,
    return_mean in [-25, -10] for dense, ep_len_mean == 50
---------------------------------------------------------
```

Running the pipeline

The `all` subcommand runs three inspection stages:

Stage 1: Describe (`describe`). Creates `results/ch01_env_describe.json` documenting the observation and action spaces. This is the machine-readable version of section 2.3.

Stage 2: Reward check (`reward-check`). Runs 500 steps of random actions, verifying the critical invariant at each step: `env.step()` reward matches `compute_reward()` matches the distance formula. Also tests relabeling with random goals. This is the automated version of sections 2.5 and 2.6.

Stage 3: Random episodes (`random-episodes`). Runs 10 complete episodes with a random policy and records baseline metrics: success rate, mean return, mean episode length, and final goal distance. This establishes the performance floor.

Interpreting the artifacts

`results/ch01_env_describe.json` contains the observation and action space schema. Open it and verify:

- `observation_space.observation.shape` == `[10]`
- `observation_space.achieved_goal.shape` == `[3]`
- `observation_space.desired_goal.shape` == `[3]`
- `action_space.shape` == `[4]`
- `action_space.low` == `[-1, -1, -1, -1]`
- `action_space.high` == `[1, 1, 1, 1]`

`results/ch01_random_metrics.json` contains the random baseline. Expected values for FetchReachDense-v4:

Metric	Expected range	Meaning
`success_rate`	0.0-0.1	Random actions very rarely reach the goal
`return_mean`	-25 to -10	Sum of negative distances over 50 steps
`ep_len_mean`	50	Episodes always run to the truncation limit
`final_distance_mean`	0.05-0.15	Average distance to goal at episode end

These baseline numbers are your floor. In Chapter 3, PPO must produce a success rate well above 0.1 and a return much closer to 0. If a trained policy's metrics are not clearly better than these random baseline values, something is wrong with training.

The random baseline as diagnostic tool

The random baseline is more useful than it might appear. It answers a concrete question: ”how hard is this task for an agent that does nothing intelligent?” If random success rate is already 20-30%, the task might be too easy to test your algorithm. If it is 0.0% over hundreds of episodes, the task is genuinely hard -- the agent must learn something specific to succeed.

For FetchReachDense-v4 with a threshold of 0.05 meters, random success is typically 0-10%. The workspace is much larger than the success region, so stumbling into the goal by chance is rare but not impossible. This makes Reach a good development environment: hard enough that random does not solve it, easy enough that basic algorithms (PPO with dense rewards) reliably learn it.

Note the `ep_len_mean` of 50. This confirms something we mentioned in section 2.2: episodes are truncated at 50 steps, never terminated early. Even when the agent reaches the goal, the episode continues. This is a design choice in the Fetch environments -- the agent is rewarded for staying at the goal, not just reaching it. For dense rewards, staying at the goal produces rewards near 0 (distance near 0), which is the maximum possible. For sparse rewards, staying at the goal produces reward 0 (success), which is also the maximum. Both reward structures incentivize reaching and holding the goal position.

2.10 What can go wrong

Here are the most common issues when inspecting Fetch environments, along with diagnostics. We have encountered all of these during development.

`obs` is a flat array, not a dictionary

Symptom. `type(obs)` is `ndarray`, not `dict`. Calling `obs["observation"]` raises `TypeError`.

Cause. Old version of gymnasium-robotics (before 1.0), wrong environment ID, or a wrapper that flattens the dictionary.

Fix. Run `pip show gymnasium-robotics` and check the version is >= 1.0. Verify the environment ID matches a known Fetch environment. If you are wrapping the environment with SB3's `FlattenObservation`, remove that wrapper -- goal-conditioned environments must keep the dictionary structure.

`obs["observation"]` shape is not (10,) for FetchReach

Symptom. The observation has 25 dimensions instead of 10 (or some other unexpected number).

Cause. You are using FetchPush or FetchPickAndPlace (which have 25D observations), or there is a version mismatch.

Fix. Print `env.spec.id` to confirm the environment ID. Check `env.observation_space` for the full structure.

`compute_reward` raises `AttributeError`

Symptom. `env.compute_reward(...)` fails with `AttributeError: 'TimeLimit' object has no attribute 'compute_reward'`.

Cause. Calling on the wrapped environment instead of the unwrapped base environment. `gym.make()` wraps environments in `TimeLimit` and other wrappers that do not expose `compute_reward`.

Fix. Use `env.unwrapped.compute_reward(ag, dg, info)`.

Step reward and `compute_reward` disagree

Symptom. The three-way comparison shows mismatches. Manual reward, `compute_reward`, and `env.step()` return different values.

Cause. Version mismatch between gymnasium and gymnasium-robotics. Some versions changed the reward computation or the `compute_reward` API.

Fix. Upgrade both packages: `pip install --upgrade gymnasium gymnasium-robotics`. Then re-run the verification.

`desired_goal` is identical across resets

Symptom. Every call to `env.reset()` returns the same `desired_goal`.

Cause. Not passing different seeds, or passing the same seed every time.

Fix. Pass unique seeds: `env.reset(seed=42 + episode_number)`. If you want truly random goals, omit the seed argument.

`achieved_goal` does not match `obs["observation"][:3]` for FetchPush

Symptom. For FetchPush, `np.allclose(obs["achieved_goal"], obs["observation"][:3])` returns False.

Cause. This is correct behavior, not a bug. For Push and PickAndPlace, `achieved_goal` is the **object** position, not the gripper position. The gripper position is `obs["observation"][:3]`, but the goal is about where the object ends up.

Fix. No fix needed -- this is by design. See section 2.7 for the explanation.

`EnvironmentNameNotFound` for FetchReachDense-v4

Symptom. `gym.make("FetchReachDense-v4")` raises `gymnasium.error.NameNotFound`.

Cause. gymnasium-robotics is not installed, or the version does not include v4 environments.

Fix. Install with `pip install gymnasium-robotics`. If installed but v4 is not found, check the version -- v4 environments were introduced in gymnasium-robotics 1.2.0.

Random success rate much higher than 0.1

Symptom. Running 100 random episodes gives success_rate of 0.3 or higher.

Cause. The distance threshold might be larger than expected, or the goal space might be very small.

Fix. Check `env.unwrapped.distance_threshold` -- it should be 0.05. If it is different, your environment version uses a different default.

`is_success` is True immediately after reset

Symptom. The very first step of an episode reports `is_success: True` in the info dict, or the sparse reward is 0 right after reset.

Cause. The desired goal happened to be sampled at (or very near) the gripper's initial position. This is rare but normal -- it occurs in fewer than 5% of episodes.

Fix. No fix needed. Run multiple episodes and verify that success at reset is rare. If it happens consistently (every episode), there is likely a bug in goal sampling -- check that you are passing different seeds to `env.reset()`.

Workspace bounds look wrong

Symptom. Goal positions are near (0, 0, 0) or have very large values.

Cause. Different MuJoCo model version or coordinate frame issue.

Fix. Check `env.unwrapped.initial_gripper_xpos` -- typical values are around [1.34, 0.75, 0.53]. If these are very different, the MuJoCo model may have been modified or replaced.

2.11 Summary

You now understand the Fetch environment interface at the level needed to train and debug policies. Specifically:

- **Observations** are dictionaries with three keys: `observation` (proprioceptive state, 10D for Reach, 25D for Push/PickAndPlace), `achieved_goal` (where you are, 3D), and `desired_goal` (where you should be, 3D). This structure is what makes goal-conditioned learning explicit.
- **Actions** are 4D Cartesian deltas in [-1, 1]: three components for end-effector movement (dx, dy, dz) and one for gripper control. The agent operates in Cartesian space; an internal controller handles inverse kinematics.
- **Dense rewards** equal $-\|g_a - g_d\|$ -- negative distance. **Sparse rewards** are 0 if distance $\leq$ 0.05, else -1. Both can be recomputed for arbitrary goals via `compute_reward`.
- **The critical invariant** -- `env.step()` reward equals `compute_reward(ag, dg, info)` -- holds for all Fetch environments and is the foundation of HER.
- **Goal relabeling works**: calling `compute_reward` with goals the environment never set produces correct rewards, enabling the ”what if that had been the goal?” trick at the core of HER.
- **The interface is uniform across Fetch tasks**: same dictionary structure, same action space, same `compute_reward` API. What changes is the observation dimension and what `achieved_goal` tracks (gripper for Reach, object for Push/PickAndPlace).
- **The random baseline** (success_rate 0.0-0.1, return_mean in [-25, -10] for dense Reach) is the performance floor that any trained agent must beat.

With this anatomy understood, Chapter 3 trains a real policy. PPO on FetchReachDense-v4 will use the observation shapes you documented here to build its network, the reward signal you verified to drive learning, and the random baseline you established as the metric it must surpass.

Verify It

```
---------------------------------------------------------
VERIFY IT
---------------------------------------------------------
This chapter does not train any policies. The verification
commands below confirm that your environment installation
is correct and that all inspection outputs match expected
values.

  bash docker/dev.sh python scripts/ch01_env_anatomy.py all \
    --seed 0

Hardware:     Any machine with Docker (no GPU required)
Time:         < 2 min
```

```
Artifacts produced:
  results/ch01_env_describe.json
  results/ch01_random_metrics.json

Expected inspection outputs:
  describe:
    observation_space.observation.shape == [10]
    observation_space.achieved_goal.shape == [3]
    observation_space.desired_goal.shape == [3]
    action_space.shape == [4]
    action_space.low == [-1, -1, -1, -1]
    action_space.high == [1, 1, 1, 1]
    distance_threshold == 0.05
    reward_type == "dense"

  reward-check:
    "OK: reward checks passed" (500 steps, 3 random goals
    per step, atol=1e-6, zero mismatches)

  random-episodes (FetchReachDense-v4, 10 episodes):
    success_rate: 0.0-0.1
    return_mean: approx -15 to -25
    ep_len_mean: 50
    final_distance_mean: approx 0.05-0.15

Lab verification:
  bash docker/dev.sh python scripts/labs/env_anatomy.py --verify
    All checks pass in < 1 min on CPU.

  bash docker/dev.sh python scripts/labs/env_anatomy.py --bridge
    Manual compute_reward matches env.step() reward on 100 steps.
    Dense reward matches -np.linalg.norm(ag - dg) within atol=1e-10.
    Sparse reward matches threshold formula within atol=1e-10.
    Relabeled goals: compute_reward produces correct reward for
    arbitrary goals (simulating HER relabeling).

If any check fails, see "What Can Go Wrong" in this chapter.
----------------------------------------------------------
```

Exercises

1. (Verify) Confirm observation structure across seeds.

Reset FetchReachDense-v4 with 5 different seeds. For each reset, verify that the observation dictionary has the same three keys and the same shapes. Record the range of desired_goal values across resets.

Expected: goals span the workspace (x roughly 1.05-1.55, y roughly 0.40-1.10, z roughly 0.42-0.60). If all goals are identical, something is wrong with the seed handling -- check that you are passing different seeds to each `env.reset()` call.

2. (Tweak) Compare dense and sparse rewards on the same trajectory.

Create both `FetchReachDense-v4` and `FetchReach-v4` with the same seed. Take the same sequence of 50 random actions in both. Compare the reward sequences side by side. Questions:

- (a) Is the dense return always more negative than the sparse return?
- (b) What fraction of steps have sparse reward = 0?
- (c) At what distance does the sparse reward switch from -1 to 0?

Expected: sparse reward is 0 only when distance < 0.05. Dense return is typically -15 to -25; sparse return is typically -50 (all -1s, since random actions rarely reach the goal).

3. (Extend) Observation breakdown for FetchPush.

Create `FetchPushDense-v4` and inspect the 25D observation vector. Using the observation component table from section 2.7, identify:

- (a) Which indices correspond to the object position
- (b) What `achieved_goal` represents (hint: it is the object position, not the gripper position)
- (c) What happens to the object position when you take action `[0, 0, 0, 0]` for 50 steps -- does the object move?

Expected: the object stays roughly in place (gravity keeps it on the table); `achieved_goal` tracks the object, not the gripper.

4. (Challenge) Estimate success rate as a function of distance threshold.

Run 100 random episodes on `FetchReachDense-v4`. For each episode, record the final distance between `achieved_goal` and `desired_goal`. Then tabulate what the success rate *would be* at thresholds of 0.01, 0.02, 0.05, 0.10, and 0.20 meters. How does the "difficulty" of the task change with the threshold?

Expected: at threshold 0.01, random success is roughly 0%; at 0.20, it may be 5-15%. The default threshold of 0.05 makes random success very rare (0-5%). This exercise gives you intuition for how the success threshold determines task difficulty -- a concept we revisit when discussing sparse rewards in Chapter 5.