

# Contents

<b>1 Chapter 0: A Containerized "Proof of Life" on Spark DGX</b>	<b>1</b>
1.1 Abstract	1
1.2 Run It (TL;DR)	1
1.3 Part I: The Problem	2
1.3.1 1.1 WHY: The Reproducibility Crisis and Its Resolution	2
1.3.2 1.2 The Specific Problem of This Chapter	2
1.4 Part II: The Method	3
1.4.1 2.1 HOW: Containerization as Environment Specification	3
1.4.2 2.2 The Verification Protocol	3
1.4.3 2.3 The Container Architecture	6
1.4.4 2.4 The Virtual Environment Within the Container	7
1.5 Part III: The Implementation	7
1.5.1 3.1 WHAT: The Concrete Steps	7
1.5.2 3.2 Mac M4 (Apple Silicon) Support	9
1.6 Part IV: Analysis and Verification	10
1.6.1 4.1 Interpreting the Results	10
1.6.2 4.2 On the Meaning of "Proof of Life"	10
1.6.3 4.3 What This Chapter Does Not Verify	11
1.7 Part V: Deliverables	11
1.8 Appendix A: Troubleshooting	11
1.8.1 A.1 "Permission denied" When Running Docker	11
1.8.2 A.2 "I have no name!" in the Container Shell	11
1.8.3 A.3 EGL Initialization Failures	11
1.8.4 A.4 Dependency Hash Mismatch	12
1.9 Appendix B: Environment Variable Reference	12

## 1 Chapter 0: A Containerized "Proof of Life" on Spark DGX

### 1.1 Abstract

This chapter establishes the foundational experimental environment upon which all subsequent work depends. We address a problem that is logically prior to reinforcement learning itself: the problem of *reproducible computation*. Our deliverables are not trained policies but verified infrastructure--a container that runs, a renderer that produces images, a training loop that completes without error.

We borrow the term *proof of life* from hostage negotiations, where it means evidence that someone is still alive. Here, it means evidence that the computational environment is functional -- capable of producing valid, reproducible results.

We find it tempting to skip "setup" chapters, but this one does more than configure tools -- it establishes the laboratory in which all subsequent results are produced. A result that cannot be reproduced because the environment cannot be reconstructed is not a result at all. This chapter ensures that our laboratory is sound.

### 1.2 Run It (TL;DR)

*# From the host, in your repo clone:*

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

Done when smoke\_frame.png and ppo\_smoke.zip are created.

---

## 1.3 Part I: The Problem

### 1.3.1 1.1 WHY: The Reproducibility Crisis and Its Resolution

Consider the following scenario, which is unfortunately common in empirical machine learning research. A researcher trains a policy that achieves impressive results. They write a paper, submit it, and it is accepted. Six months later, a colleague attempts to reproduce the results. The code is available, but it fails to run: dependencies have changed, the CUDA version is different, an environment variable is missing. After days of debugging, the colleague achieves results that are qualitatively different from the original paper. Was there an error in the original work? Or is the discrepancy due to environmental differences? It is impossible to know.

This scenario illustrates what we might call the *reproducibility problem* in empirical machine learning:

**Problem (Reproducibility).** *Given a computational experiment  $E$  that produces result  $R$  on machine  $M$  at time  $t$ , under what conditions can we guarantee that  $E$  produces result  $R'$  with  $\|R - R'\| < \epsilon$  on machine  $M'$  at time  $t' > t$ ?*

The problem is harder than it appears. The result  $R$  depends not only on the code but on the entire computational environment: the operating system, the installed libraries, the GPU driver version, the CUDA toolkit, the Python interpreter, and dozens of other components. Any of these may change between  $t$  and  $t'$ , and any change may affect  $R$ .

In the tradition of Hadamard, we ask: Is the reproducibility problem *well-posed*?

**Definition (Well-Posedness for Reproducibility).** *The reproducibility problem is well-posed if: (1) there exists an environment specification  $S$  such that running  $E$  in any environment satisfying  $S$  produces consistent results; (2) the specification  $S$  is unique up to equivalence; (3) small perturbations to  $S$  produce small perturbations to  $R$ .*

Condition (1) requires that we can *specify* an environment precisely enough to guarantee consistency. Condition (2) requires that the specification be *canonical*--that there not be multiple incompatible specifications claiming to represent the same environment. Condition (3) requires *stability*--that the result not be arbitrarily sensitive to minor environmental variations.

Containerization addresses all three conditions: a Docker image provides a complete, self-contained specification of the computational environment; the image is identified by a content-addressable hash, ensuring uniqueness; and the layered filesystem ensures that small changes to the specification (adding a package, changing a configuration) produce small changes to the resulting environment.

This is why we containerize: not for convenience, but for *scientific validity*.

### 1.3.2 1.2 The Specific Problem of This Chapter

Within the general reproducibility framework, this chapter addresses a specific sub-problem:

**Problem (Environment Verification).** *Construct and verify a containerized environment  $S$  such that: (1) GPU computation is available within  $S$ ; (2) MuJoCo physics simulation runs correctly within  $S$ ; (3) headless rendering produces valid images within  $S$ ; (4) a complete training loop executes without error within  $S$ .*

Each condition is necessary for the reinforcement learning experiments that follow. GPU access accelerates training for pixel-based chapters (Ch9) and is required for GPU-physics chapters (Appendix E), though for state-based chapters (Ch1-8) the CPU is the actual bottleneck and GPU adds little. MuJoCo is the physics engine underlying our Fetch robot, rendering is how we

generate evaluation videos, and the training loop is the pipeline through which policies are learned -- so a failure in any one of these blocks an entire class of experiments.

We recommend completing all verification steps before proceeding. In our experience, skipping this chapter and proceeding directly to training eventually surfaces as rendering errors, CUDA misconfigurations, or import failures -- and debugging those in context takes longer than verifying the environment systematically from the start.

---

## 1.4 Part II: The Method

### 1.4.1 2.1 HOW: Containerization as Environment Specification

Our approach is to specify the environment as a Docker container and to verify each requirement through explicit tests.

**Definition (Container).** *A container is a lightweight, isolated execution environment defined by an image. The image specifies the filesystem contents, environment variables, and default commands. Containers share the host kernel but have isolated process and network namespaces.*

**Definition (Image).** *An image is an immutable template from which containers are instantiated. Images are identified by a content-addressable hash (digest) and may be tagged with human-readable names (e.g., `robotics-rl:latest`).*

The key property of containers for our purposes is *isolation with specification*. The container is isolated from the host environment--packages installed on the host do not affect the container--but the isolation is precisely specified by the image, which can be versioned, shared, and reconstructed.

**Remark (On the Choice of Docker).** *We use Docker rather than alternatives (Singularity, Podman, etc.) because Docker is ubiquitous, well-documented, and fully supported by the NVIDIA Container Toolkit. For HPC environments where Docker is unavailable, the concepts transfer to Singularity with minor modifications.*

### 1.4.2 2.2 The Verification Protocol

Verification is not bureaucracy -- it is the empirical side of our well-posedness analysis. Each test corresponds to a necessary condition for the experiments that follow, so that if any test fails, some class of experiments becomes impossible. Understanding *why* each test matters (not just *what* it checks) is essential for diagnosing failures when they occur.

**1.4.2.1 Test 1: GPU Access What we verify.** The container can access the host GPU via the NVIDIA runtime.

**Why this matters.** We verify GPU access early because later chapters need it -- but not all chapters, and not for the same reasons. The compute requirements vary significantly across the curriculum:

Chapters	Workload	CPU viable?
0-8	State-based RL (MuJoCo + 256x256 MLPs on 25D vectors)	Yes (~60-100 fps Mac, ~600 fps DGX)
9	Pixel-based RL (CNN on 84x84x12 images)	Slow but possible
App. E	Isaac Lab (GPU-parallel PhysX, 64-128 envs)	No

For Chapters 0-8, the bottleneck is MuJoCo physics simulation, which runs on CPU regardless of platform. A 256x256 MLP processing a 25D vector completes forward and backward passes in microseconds -- the GPU has almost nothing to do. Training runs that take minutes on DGX take tens of minutes on a Mac laptop, not days.

The GPU becomes important at Chapter 9, where a CNN processes 84x84 pixel images every step, and essential at Appendix E, where Isaac Lab runs physics itself on the GPU. We verify GPU access now so that readers on GPU-equipped machines catch driver or toolkit issues early, before they matter.

But GPU access inside a container is not automatic. The container runs in an isolated namespace; it cannot see host devices unless explicitly granted access. The `--gpus all` flag instructs Docker to use the NVIDIA Container Toolkit, which mounts the GPU device files and driver libraries into the container.

**What failure means.** If this test fails, either:

1. The NVIDIA driver is not installed on the host
2. The NVIDIA Container Toolkit is not installed
3. Docker was not invoked with `--gpus all`
4. The GPU is in use by another process with exclusive access

Training will still *run* on CPU -- and for Chapters 0-8 CPU is perfectly adequate, while for Chapter 9 training will be 2-3x slower but workable. Only Appendix E (Isaac Lab) has no CPU path and requires an NVIDIA GPU.

**The test.** The script checks `torch.cuda.is_available()` inside the container. If CUDA is available, it reports the device name and count. If not, it prints a warning but does not halt the test sequence -- training can still proceed on CPU (this is the expected path on Mac). On a DGX system where CUDA *should* be available, treat a "CUDA not available" warning as a real problem: check that Docker was invoked with `--gpus all` and that the NVIDIA Container Toolkit is installed.

#### 1.4.2.2 Test 2: MuJoCo and Gymnasium-Robotics Functionality What we verify.

The physics simulator initializes correctly, and the Fetch environments are registered and functional.

**Why this matters.** The Fetch environments are implemented on top of MuJoCo, a physics engine that simulates rigid body dynamics with contact. MuJoCo is not a pure Python library; it includes compiled C code that interfaces with system libraries. If these libraries are missing or incompatible, MuJoCo fails to initialize.

Furthermore, Gymnasium-Robotics must register its environments with Gymnasium's registry at import time, which means that if the import fails silently or the registration is incomplete, `gym.make("FetchReach-v4")` will raise `EnvironmentNameNotFound`.

**What failure means.** If this test fails, either:

1. MuJoCo's compiled extensions cannot find required system libraries
2. The gymnasium-robotics package is not installed
3. There is a version incompatibility between gymnasium, gymnasium-robotics, and mujoco

Without functional Fetch environments, the entire curriculum is blocked.

**The test.** Import `gymnasium` and `gymnasium_robotics`, then call `gym.make("FetchReachDense-v4")` (a variant where the reward is proportional to distance from the goal -- we formalize dense vs. sparse rewards in Chapter 1) and `env.reset()`. If the environment returns a valid observation dictionary with keys `observation`, `achieved_goal`, and `desired_goal`, the physics stack is functional.

**1.4.2.3 Test 3: Headless Rendering What we verify.** The environment can produce RGB frames without a display.

**Why this matters.** Evaluation often requires visual inspection: Does the robot reach the goal? Is the motion smooth or jerky? Does the gripper close at the right moment? These questions are answered by watching videos, which requires rendering.

But DGX systems are headless--they have no monitor attached. Rendering typically requires a display server (X11) to manage the graphics context. On a headless system, we must use *offscreen* rendering: EGL (hardware-accelerated via the GPU) or OSMesa (software rasterization).

This is where many setups fail, because EGL requires specific driver support and library versions while OSMesa requires Mesa to be compiled with offscreen support -- and if neither backend works, rendering is impossible.

**What failure means.** If this test fails, either:

1. EGL libraries (`libEGL.so`) are missing or incompatible
2. The GPU driver does not expose EGL support
3. OSMesa libraries (`libOSMesa.so`) are missing
4. Environment variables (`MUJOCO_GL`, `PYOPENGL_PLATFORM`) are misconfigured

Training can proceed without rendering, but evaluation will be limited to numerical metrics. You will not be able to generate videos or visually debug policy behavior.

**The test.** Create a Fetch environment with `render_mode="rgb_array"`, call `env.render()`, and save the resulting numpy array as a PNG image. If the image file exists and is non-empty, offscreen rendering works.

**Remark (The Fallback Chain).** The proof-of-life script implements a fallback chain: it first attempts EGL (preferred, hardware-accelerated), then OSMesa (slower, but compatible), then disables rendering entirely. The test passes if *any* backend produces a valid image. The fallback works by calling `os.execvpe`, which *replaces* the current process entirely with a new invocation using different environment variables -- this is not a retry within the same process, but a full re-exec. As a result, on DGX systems where EGL fails, you may see the script's startup output appear twice in logs -- once for the EGL attempt and once for the OSMesa retry. This is expected behavior, not an error. On systems without an NVIDIA driver stack (no `nvidia-smi`), the `all` subcommand starts with OSMesa, so the double-output pattern does not occur unless you explicitly force `MUJOCO_GL=egl`.

**1.4.2.4 Test 4: Training Loop Completion What we verify.** A complete training loop--environment interaction, gradient computation, parameter updates, checkpoint saving--executes without error.

**Why this matters.** The previous tests verified components in isolation -- GPU access, physics simulation, rendering -- but reinforcement learning combines these components in ways that create new failure modes. Data flows from the environment to the replay buffer to the neural network and back, which means that shapes must match, dtypes must be compatible, and memory must not leak across the entire pipeline.

Many bugs only manifest when components interact: a shape mismatch between the observation space and the policy network, a dtype incompatibility between numpy arrays and PyTorch tensors, or a memory leak that only appears after thousands of environment steps. These bugs do not surface in unit tests; they appear when you run training.

**What failure means.** If this test fails, either:

1. The policy network architecture is incompatible with the observation space

2. There is a dtype or device mismatch (CPU vs. CUDA tensors)
3. The training loop has a bug that manifests only after some number of steps
4. The checkpoint serialization format is incompatible with the model architecture

Until this test passes, you cannot train policies.

**The test.** Run PPO (Proximal Policy Optimization, a policy gradient method we formalize in Chapter 2) for 50,000 timesteps on FetchReachDense-v4 with 8 parallel environments, then save a checkpoint. If the checkpoint file exists and is loadable by `PP0.load()`, the training loop is functional.

**Remark (Why PPO, Not SAC).** We use PPO for this smoke test because it is simpler and fails faster if something is wrong. SAC (Soft Actor-Critic, an off-policy algorithm introduced in Chapter 3) involves additional components (replay buffer, twin critics, entropy tuning) that could mask or compound errors. Once PPO works, we have confidence that the core training infrastructure is sound; SAC-specific issues can be debugged separately.

#### 1.4.2.5 The Logical Structure

The four tests form a dependency chain:

GPU Access -> MuJoCo Functionality -> Headless Rendering -> Training Loop

Each test assumes the previous tests pass, so it helps to diagnose in order -- rendering issues are harder to debug if MuJoCo itself cannot initialize, and training performance is hard to evaluate without verifying the compute environment.

**Run the tests in order.** If a test fails, diagnose and fix it before proceeding. The all sub-command respects this ordering and stops at the first failure. The one exception is `gpu-check`, which always exits with status 0 even when CUDA is unavailable, so it warns but does not block subsequent tests. This is intentional: CPU-only operation is valid on Mac and other non-NVIDIA platforms.

These tests are implemented in `scripts/ch00_proof_of_life.py`. The script provides sub-commands for running each test individually (`gpu-check`, `list-envs`, `render`, `ppo-smoke`) or all tests in sequence (`all`).

### 1.4.3 2.3 The Container Architecture

Our container architecture consists of two layers:

**Base Layer.** We use the NVIDIA PyTorch image (`nvcr.io/nvidia/pytorch:25.12-py3`) as the base. This image provides CUDA, cuDNN, PyTorch, and other deep learning infrastructure pre-configured and tested by NVIDIA.

**Project Layer.** On top of the base, we install system dependencies for MuJoCo rendering (EGL, OSMesa) and Python dependencies for the project (Gymnasium, Gymnasium-Robotics, Stable Baselines 3). These are specified in the `docker/Dockerfile`, which builds the `robotics-rl:latest` image:

- **System packages:** `libegl1`, `libgl1`, `libosmesa6` (headless rendering), `libglfw3` (windowed rendering), `ffmpeg` (video encoding)
- **Python packages:** `gymnasium`, `gymnasium-robotics`, `mujoco`, `stable-baselines3`, `tensorboard`, `imageio`

The `docker/dev.sh` script automatically builds this image on first run if it does not exist locally, falling back to the raw NVIDIA base image if the build fails (e.g., due to network issues) -- though rendering may be unavailable in that case.

**Remark (On Version Pinning).** The `requirements.txt` file specifies minimum version constraints (e.g., `stable-baselines3>=2.4.0`) rather than exact pins. For strict reproducibility,

the Docker image digest is the true specification--it freezes the entire dependency tree. Readers who want to capture exact versions for a paper can run `pip freeze` inside the container and save the output.

**Remark (On the Two-Layer Architecture).** The separation into base and project layers reflects a design principle: heavyweight, stable dependencies (CUDA, PyTorch) belong in the base layer; lightweight, project-specific dependencies belong in the project layer. This separation enables faster iteration--changing project dependencies does not require rebuilding the entire CUDA stack--while maintaining reproducibility.

#### 1.4.4 2.4 The Virtual Environment Within the Container

A subtle point deserves elaboration: we use a Python virtual environment *inside* the container, even though the container already provides isolation.

**Proposition.** A virtual environment inside a container provides additional benefits: (1) it enables `pip install -e .` for editable installs of the project; (2) it allows project dependencies to shadow container dependencies when necessary; (3) it makes the dependency specification explicit in `requirements.txt` rather than implicit in the `Dockerfile`.

On DGX/NVIDIA, the virtual environment is created with `--system-site-packages`, which allows it to inherit packages from the container's system Python. This avoids reinstalling PyTorch (which is large and CUDA-specific) while still allowing project-specific packages to be installed separately. On Mac, the container uses a plain `python:3.11-slim` base; `docker/dev.sh` installs CPU PyTorch into the venv (since `requirements.txt` intentionally does not pin torch), then installs the remaining dependencies from `requirements.txt`.

---

## 1.5 Part III: The Implementation

### 1.5.1 3.1 WHAT: The Concrete Steps

We now describe the concrete steps to establish and verify the environment.

**1.5.1.1 Step 1: Verify Host Prerequisites** Before using containers, we must verify that the host system is properly configured.

**Verification (Docker Availability).** Run `docker --version` on the host. The test passes if Docker reports a version number.

**Verification (NVIDIA Runtime).** Run `docker run --rm --gpus all nvcr.io/nvidia/pytorch:25.12-py3 nvidia-smi` on the host. The test passes if `nvidia-smi` output appears showing at least one GPU.

If either test fails, consult your system administrator. This tutorial does not cover Docker installation or NVIDIA runtime configuration.

**1.5.1.2 Step 2: Enter the Development Environment** The repository provides a wrapper script that automates container setup:

```
cd /path/to/robotics # your local clone
bash docker/dev.sh
```

This script performs the following operations:

1. Checks for the `robotics-rl:latest` image; builds it if missing
2. Launches a container with GPU access and appropriate environment variables

3. Mounts the repository at /workspace
4. Runs as your host UID/GID to avoid permission issues
5. Creates .venv if it does not exist and installs requirements.txt
6. Activates the virtual environment and drops you into a shell

Alternatively, to run a single command without entering an interactive shell:

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

**1.5.1.3 Step 3: Verify Fetch Environment Availability** Inside the container (or via docker/dev.sh), run:

```
python scripts/ch00_proof_of_life.py list-envs
```

**Expected Output.** A list of environment IDs including FetchReach-v4, FetchReachDense-v4, FetchPush-v4, FetchPickAndPlace-v4, and their variants.

**Failure Mode.** If no Fetch environments appear, gymnasium-robotics is not installed correctly. Verify with `pip list | grep gymnasium`.

**1.5.1.4 Step 4: Verify Headless Rendering**

```
python scripts/ch00_proof_of_life.py render --out smoke_frame.png
```

**Expected Output.** A message indicating successful rendering and the creation of smoke\_frame.png.

**Failure Mode.** If rendering fails with EGL errors, the script attempts a fallback to OSMesa; if both fail, rendering is disabled entirely, which means you can still train policies but cannot generate evaluation videos.

**Remark (Rendering Backend Hierarchy).** *The script implements a fallback chain: EGL (hardware-accelerated) -> OSMesa (software) -> disabled. EGL requires NVIDIA drivers and EGL libraries; OSMesa requires the Mesa library. The robotics-rl:latest image includes both.*

**1.5.1.5 Step 5: Verify Training Loop**

```
python scripts/ch00_proof_of_life.py ppo-smoke --n-envs 8 --total-steps 50000 --out ppo_smoke.zip
```

**Expected Output.** Training progress messages followed by the creation of ppo\_smoke.zip.

**Remark (Hyperparameters).** *The smoke test uses n\_steps=1024 (half of SB3's default of 2048, to keep the test short) and batch\_size=256. These are configurable via --n-steps and --batch-size but the defaults are fine for verification. Chapter 2 discusses how these parameters affect learning.*

**Failure Mode.** CUDA errors indicate GPU misconfiguration, shape mismatch errors point to problems with observation space handling, and import errors indicate missing dependencies.

**1.5.1.6 Step 6: Combined Verification** For convenience, all tests can be run in sequence:

```
python scripts/ch00_proof_of_life.py all
```

This is the recommended way to verify a fresh installation.



### 1.5.2 3.2 Mac M4 (Apple Silicon) Support

The platform also supports development on Apple Silicon Macs (M4, M3, M2, M1). The same `docker/dev.sh` command works on both platforms--the script auto-detects the host and configures appropriately.

#### 1.5.2.1 Platform Differences

Aspect	DGX / NVIDIA	Mac M4 (Apple Silicon)
<b>Architecture</b>	x86_64	ARM64
<b>Docker image</b>	robotics-rl:latest	robotics-rl:mac
<b>Dockerfile</b>	docker/Dockerfile	docker/Dockerfile.mac
<b>Base image</b>	nvcr.io/nvidia/pytorch:25.12-py3	python:3.11-slim
<b>Compute device</b>	CUDA (GPU)	CPU only
<b>Rendering backend</b>	EGL (hardware)	OSMesa (software)
<b>Typical throughput</b>	~600 fps	~60-100 fps

**1.5.2.2 Why CPU-Only on Mac?** Apple's Metal Performance Shaders (MPS) backend for PyTorch exists but has edge cases with certain operations, so for maximum compatibility we default to CPU on Mac -- which is perfectly adequate for development and debugging, since the physics simulation in MuJoCo is CPU-bound anyway.

**Remark (On Performance).** *The 6-10x throughput gap between Mac and DGX is mostly about faster CPUs and more cores on DGX, not GPU acceleration. For state-based RL (Ch1-8), both platforms are CPU-bound -- MuJoCo physics dominates, and the 256x256 MLP completes in microseconds regardless of device. Mac is fully viable for Chapters 1-8; training runs that take seconds on DGX take tens of minutes on Mac, which is fine for learning and iteration. For pixel-based RL (Ch9), the gap widens because CNNs benefit from GPU parallelism -- but RAM, not GPU speed, is usually the binding constraint there (see Ch9 for buffer-size guidance).*

**1.5.2.3 Usage on Mac** The commands are identical:

```
# Build image (auto-detects Mac, uses Dockerfile.mac)
bash docker/build.sh
```

```
# Run proof of life (auto-detects Mac, uses robotics-rl:mac)
bash docker/dev.sh python scripts/ch00_proof_of_life.py all
```

The platform detection uses `uname -s` (Darwin for Mac) and `uname -m` (arm64 for Apple Silicon).

**1.5.2.4 Expected Results by Platform** All tests should pass on both platforms, and all artifacts should be generated correctly. The policies learned on Mac are interchangeable with those trained on DGX -- the learned weights are platform-independent.

Aspect	Mac M4	DGX / NVIDIA
Device reported	cpu	cuda
Rendering backend	osmesa	egl
Training throughput	~60-100 fps	~600 fps
smoke_frame.png	Yes (Fetch robot visible)	Yes (Fetch robot visible)
ppo_smoke.zip	Yes (loadable)	Yes (loadable)
All tests pass	Yes	Yes

### 1.5.2.5 Known Limitations

1. **State-based RL (Ch1-8):** Mac is fully viable -- training runs complete in tens of minutes rather than seconds, which is fine for learning and iteration, and no GPU is needed.
2. **Pixel-based RL (Ch9):** RAM is the binding constraint, not GPU speed. A 500K-transition pixel buffer uses ~80 GB; Mac laptops with 32 GB should use `--buffer-size 100000` (see Ch9 for per-tier guidance). Training is 2-3x slower without a GPU but workable.
3. **Isaac Lab (Appendix E):** Not available on Mac, since Isaac Lab requires Linux + NVIDIA GPU for GPU-accelerated physics (PhysX) and has no CPU fallback.
4. **Rendering quality:** OSMesa (software rendering) produces identical images to EGL but is slower, which matters only for video generation, not for training.
5. **Docker Desktop memory:** You may need to increase Docker Desktop's memory allocation (Settings -> Resources -> Memory) to 8GB+ for large batch sizes or long training runs.
6. **No MPS (Apple GPU) support:** MPS (Metal Performance Shaders) cannot work in Docker containers because Docker runs Linux, not macOS. MPS is a macOS-only API that requires direct access to Apple's Metal framework. Inside a Linux container, `torch.backends.mps.is_available()` always returns False, regardless of the host being an M4 Mac.

### 1.5.2.6 Docker Desktop Configuration for Mac If you encounter out-of-memory errors:

1. Open Docker Desktop
2. Go to Settings (gear icon)
3. Select "Resources"
4. Increase "Memory" to at least 8 GB
5. Click "Apply & restart"

---

## 1.6 Part IV: Analysis and Verification

### 1.6.1 4.1 Interpreting the Results

Upon successful completion of all tests, the following artifacts should exist:

Artifact	Purpose	Expected Properties
<code>.venv/</code>	Python virtual environment	Contains <code>gymnasium</code> , <code>stable-baselines3</code> , etc.
<code>smoke_frame.png</code>	Rendered frame	Non-empty PNG, shows Fetch robot
<code>ppo_smoke.zip</code>	Trained checkpoint	Non-empty ZIP, loadable by SB3

The existence and properties of these artifacts constitute empirical verification that the environment is correctly configured.

### 1.6.2 4.2 On the Meaning of "Proof of Life"

As noted in the Abstract, a misconfigured environment is, from the perspective of reproducible science, *dead*: it cannot produce results that can be trusted or reproduced. The tests in this chapter establish that our environment is *alive* -- capable of producing valid, reproducible results.

### 1.6.3 4.3 What This Chapter Does Not Verify

This chapter verifies that the environment is functional; it does not verify that training produces good policies. The PPO smoke test runs for only 50,000 timesteps, which is insufficient for convergence. The purpose is to confirm that the training loop executes, not that it produces useful results.

Verification of learning performance is deferred to Chapter 2, where we establish PPO baselines with proper evaluation protocols.

---

## 1.7 Part V: Deliverables

Upon completion of this chapter, the following must be true:

**D1.** The command `bash docker/dev.sh` enters a container shell with an activated virtual environment.

**D2.** The file `smoke_frame.png` exists and contains a valid image (viewable in an image viewer, showing the Fetch robot).

**D3.** The file `ppo_smoke.zip` exists and is a valid Stable Baselines 3 checkpoint (loadable via `PPO.load("ppo_smoke.zip")`).

**D4.** All four tests in `scripts/ch00_proof_of_life.py` all pass without error.

We recommend satisfying all four conditions before proceeding. Skipping ahead tends to surface as mysterious failures in later chapters that are harder to diagnose than fixing them here.

---

## 1.8 Appendix A: Troubleshooting

### 1.8.1 A.1 "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.

**Resolution.** Either add your user to the docker group (`sudo usermod -aG docker $USER`, then log out and back in) or run Docker commands with `sudo`.

### 1.8.2 A.2 "I have no name!" in the Container Shell

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

**Cause.** The container is running as your numeric UID, which has no entry in `/etc/passwd` inside the container.

**Impact.** None -- this is purely cosmetic, and file permissions work correctly regardless.

### 1.8.3 A.3 EGL Initialization Failures

**Symptom.** `mujoco.FatalError: gladLoadGL error or similar EGL errors.`

**Cause.** EGL libraries are missing or the GPU driver does not support EGL.

**Resolution.** The `robotics-rl:latest` image includes EGL libraries. If using a different base image, install `libegl1` and `libgl1`. If EGL is unavailable, the script falls back to OSMesa.

#### 1.8.4 A.4 Dependency Hash Mismatch

**Symptom.** `docker/dev.sh` reinstalls packages on every run.

**Cause.** The requirements hash file (`.venv/.requirements.sha256`) is missing or corrupt.

**Resolution.** Delete `.venv` and re-run `docker/dev.sh` to recreate the environment.

---

### 1.9 Appendix B: Environment Variable Reference

Variable	Value	Purpose
MUJOCO_GL	egl	Selects EGL rendering backend
PYOPENGL_PLATFORM	egl	Configures PyOpenGL for EGL
NVIDIA_DRIVER_CAPABILITIES	all	Enables full GPU access in container

These variables are set automatically by `docker/dev.sh`. If you launch containers manually, you must set them explicitly.

---

**Next.** With the experimental environment verified, proceed to Chapter 1 to examine the structure of goal-conditioned Fetch environments.