# Contents



# 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.

The reader who dismisses this chapter as mere ”setup” misunderstands its purpose. In empirical machine learning, the experimental environment is not scaffolding to be discarded; it is the laboratory in which 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 Part I: The Problem

### 1.2.1 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*.

### 1.2.2 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. Without GPU access, training is prohibitively slow. Without MuJoCo, we cannot simulate the Fetch robot. Without rendering, we cannot generate evaluation videos. Without a working training loop, we cannot learn policies.

The verification is not optional. A researcher who skips this chapter and proceeds directly to training will eventually encounter failures--rendering errors, CUDA misconfigurations, import failures--and will spend more time debugging than if they had verified the environment systematically from the start.

---

## 1.3 Part II: The Method

### 1.3.1 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.*

### 1.3.2 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. 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.

#### 1.3.2.1 Test 1: GPU Access

**What we verify.** The container can access the host GPU via the NVIDIA runtime.

**Why this matters.** Reinforcement learning with neural network function approximators is computationally intensive. A single training run may require $10^6$-$10^7$ gradient updates, each involving forward and backward passes through networks with $10^5$-$10^6$ parameters. On CPU, this takes days or weeks. On GPU, it takes hours.

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, but it will be 10-100× slower, making iterative experimentation impractical.

**The test.** Run `nvidia-smi` inside the container. This command queries the NVIDIA driver for GPU status. If it succeeds, the container has GPU access. If it fails with "command not found" or "NVIDIA-SMI has failed," the container cannot see the GPU.

#### 1.3.2.2 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. This registration happens at import time. 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")` and `env.reset()`. If the environment returns a valid observation dictionary with keys `observation`, `achieved_goal`, and `desired_goal`, the physics stack is functional.

#### 1.3.2.3 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. EGL requires specific driver support and library versions. OSMesa requires Mesa to be compiled with offscreen support. If neither 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.

#### 1.3.2.4 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 complex ways. Data flows from the environment to the replay buffer to the neural network and back. Shapes must match. Dtypes must be compatible. Memory must not leak.

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. A memory leak that only appears after thousands of environment steps. These bugs do not appear 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 for 50,000 timesteps on `FetchReachDense-v4` with 8 parallel environments, then save a checkpoint. If the checkpoint file exists and is loadable by `PPO.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 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.

#### 1.3.2.5 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. There is no point testing rendering if MuJoCo cannot initialize. There is no point testing training if the GPU is inaccessible (training would "work" but be too slow to iterate).

**Run the tests in order.** If a test fails, diagnose and fix it before proceeding. The proof-of-life script's `all` subcommand respects this ordering and stops at the first failure.

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

### 1.3.3 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`, `wandb`

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

**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.*

#### 1.3.4 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.*

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.

---

### 1.4 Part III: The Implementation

#### 1.4.1 3.1 WHAT: The Concrete Steps

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

**1.4.1.1 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.

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

```
cd /home/vladp/src/robotics
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
```

#### 1.4.1.3 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`.

#### 1.4.1.4 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 fallback to OSMesa. If both fail, rendering is disabled. You can still train policies, but you cannot generate 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.*

#### 1.4.1.5 Step 5: Verify Training Loop

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

**Expected Output.** Training progress messages followed by the creation of `ppo_smoke.zip`.

**Failure Mode.** CUDA errors indicate GPU misconfiguration. Shape mismatch errors indicate problems with observation space handling. Import errors indicate missing dependencies.

#### 1.4.1.6 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.

### 1.4.2 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.

#### 1.4.2.1 Platform Differences

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

**1.4.2.2 Why CPU-Only on Mac?** Apple's Metal Performance Shaders (MPS) backend for PyTorch exists but has edge cases with certain operations. For maximum compatibility and to avoid subtle bugs, we use CPU on Mac. This is perfectly adequate for development and debugging--the physics simulation in MuJoCo is CPU-bound anyway.

**Remark (On Performance).** *The 6-10x slower throughput on Mac is expected. The bottleneck is MuJoCo physics simulation, which runs on CPU regardless of platform. On DGX, the GPU handles neural network operations in microseconds, so the CPU is the bottleneck. On Mac, both physics and neural networks run on CPU, compounding the slowdown. This is acceptable for development; use DGX for serious training runs.*

**1.4.2.3 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).

**1.4.2.4 Expected Output Differences** When running on Mac, you will see:

- Device reported as `cpu` instead of `cuda`
- Rendering backend as `osmesa` instead of `egl`
- Slower training throughput (~100 fps instead of ~600 fps)
- Slightly different timing for operations

All tests should pass, 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.

#### 1.4.2.5 Known Limitations

1. **Performance**: CPU training is ~10-20x slower than CUDA. Mac is suitable for development, debugging, and small experiments. For serious training (>100k timesteps), use DGX.
2. **Rendering quality**: OSMesa (software rendering) produces identical images to EGL but is slower. This matters only for video generation, not for training.
3. **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.

4. **No MPS support**: We intentionally avoid Metal Performance Shaders despite its availability. MPS has edge cases with certain PyTorch operations that can cause silent numerical issues. CPU is slower but more reliable.

#### 1.4.2.6 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”

---

## 1.5 Part IV: Analysis and Verification

### 1.5.1 4.1 Interpreting the Results

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

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

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

### 1.5.2 4.2 On the Meaning of ”Proof of Life”

The term ”proof of life” is borrowed from hostage negotiations, where it refers to evidence that a hostage is still alive. In our context, it refers to evidence that the computational environment is functional.

This is not mere metaphor. 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.

### 1.5.3 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.

---

## 1.6 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.

A reader who cannot satisfy all four conditions has not completed this chapter and should not proceed.

---

## 1.7 Appendix A: Troubleshooting

### 1.7.1 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`.

### 1.7.2 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 cosmetic. File permissions work correctly.

### 1.7.3 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.

### 1.7.4 A.4 Dependency Hash Mismatch

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

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

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

---

## 1.8 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.