Developers Training Handbook for ByteDance Model Training Project

Module 2: Project Structure & Dockerization Standards

Introduction

In high-performance AI training, code is treated as data. Unlike a standard web application that runs on a specific server, the code you write for this project will be executed millions of times across distributed clusters, often months after you submit it. This requires a shift in mindset from "making it work" to "making it reproducible."

Part 1: Theoretical Foundations

Before writing a single line of code, developers must understand the systems engineering concepts that drive our quality standards.

1.1 Containerization & The Hermetic Standard

Containerization allows us to package an application with all of its parts—libraries, dependencies, and system-level tools—into a single, lightweight unit called a container. Unlike a Virtual Machine (VM) which simulates hardware, a container shares the host's OS kernel but isolates the application execution.

We use Docker to solve the "Matrix of Hell"—the problem where code works on a developer's MacBook with Python 3.11 but fails on a Linux training node running Python 3.9. Docker ensures that the operating system (e.g., Debian vs. Alpine), system libraries (e.g., `gcc`, `openssl`), and language runtimes are identical for every user, every time. This is critical for security and stability. The code running inside the container cannot affect the host machine, and the host machine's configuration cannot affect the container.

For this project, we strive for Hermetic Builds. This means the execution is sealed off from the outside world. The code must not rely on anything installed on your local computer. If it’s not in the `Dockerfile`, it doesn't exist. Ideally, once the container is built, the tests should run without needing to fetch data from the internet. This prevents flakey tests caused by network timeouts or external API changes.

In this project, you will see a requirement for a `trajectory` folder. A trajectory is the recording of the "Chain of Thought" (CoT). It represents the logical path a human takes to solve a problem. Current LLMs are good at guessing the answer, but they struggle with *reasoning*. By recording your intermediate steps—your analysis, your failures, and your resource lookups—we provide the model with a map of *how to think*, not just *what to write*.

Part 2: Practical Implementation Guide

This section details the rigid standards you must follow. Automated scripts will reject any submission that deviates from this structure.

2.1 The Three-Command Rule

The golden rule of this project is Zero Manual Intervention. Any stakeholder must be able to clone your repository and run the entire lifecycle using only three commands. You must configure your `docker-compose.yml` or `Makefile` to support these distinct operations:

1. A command that sets up the environment and runs the code *before* the solution is applied.
2. A command that runs your Ground Truth solution.
3. A command that executes the tests and generates the evaluation report.

If a reviewer has to manually create a folder, install a generic library, or fix a path error, the task is immediately rejected.

2.2 Detailed Directory Architecture

You will organize your work into the following strict folder hierarchy.

1. `repository_before/` (The Problem State)

- This folder contains the codebase exactly as it exists *before* the specific task is addressed.
- **Content:**
  - **Refactoring Tasks:** The messy, unoptimized, or buggy code.
  - **Test Generation Tasks:** The working code *without* the requested tests.
  - **Feature Generation Tasks:** This folder might be empty or contain a minimal scaffold/skeleton.
- Do not include any part of your solution here.

2. `repository_after/` (The Ground Truth)

- This folder contains the "After" state—the definitive solution to the prompt.
- It is essentially `repository_before` + your changes. It must contain the clean, optimized, verified code that solves the problem.
- This code must pass all tests and be production-ready.

3. `tests/` (The Verification Layer)

- This folder contains the tests that prove the validity of the Ground Truth.
- Unit tests, integration tests, or end-to-end suites.
- **The Logic Flow:**
  - These tests should ideally *fail* when run against `repository_before` (proving the problem exists).
  - These tests must *pass* when run against `repository_after` (proving the problem is solved).
- If the prompt asks the model to write tests, your *solution tests* go in `repository_after`. The `tests/` folder will then contain **Meta-Tests**—scripts that check if the tests in `repository_after` are valid and cover the code correctly.

4. `evaluation/`

- This folder handles the output of the testing process.

- **Content:**
    - Scripts to execute the tests.
    - A generated file containing detailed metrics: execution time, pass/fail status, and error logs.
- The tests folder contains the *logic*, while evaluation contains the *runner and report*.

5. instances/

- A JSON file (instances.json) that acts as the "ID Card" for the task. It links the prompt to the specific technical verification steps.
- **Schema Fields:**
    - problem_statement: The exact text prompt given to the AI.
    - base_commit: Points to repository_before.
    - repo: The URL of the repository.
    - FAIL_TO_PASS: A list of specific test cases that fail in the "Before" state but pass in the "After" state. This proves you did work.
    - PASS_TO_PASS: Regression tests that pass in both states.

6. patches/

- A .patch file generated by git.
- The exact difference between repository_before and repository_after.
- This allows the model to learn the specific edit actions required to solve the problem (e.g., "Delete lines 10-15, Insert lines 10-12").

7. trajectory/

- A trajectory.md file documenting your engineering process.
- **Content:**
    - **Analysis:** How you deconstructed the prompt.
    - **Strategy:** Why you chose this specific algorithm or pattern.
    - **Execution:** Step-by-step implementation details.
    - **Resources:** Links to documentation or concepts used.

8. Root Level Configs

- **README.md:** Must describe the problem and list the Docker commands.
- **Dockerfile:** The blueprint for the environment.
- **Dependencies:** requirements.txt, package.json, etc. These files ensure the environment is reproducible.