Developers Training Handbook for ByteDance Model Training Project

Module 4: Testing & Evaluation

Introduction

Testing in the context of training Large Language Models is a discipline of strict verification. Unlike traditional QA, where the goal is to ensure a product is good enough for users, testing for AI training must ensure that every single requirement, constraint, and nuance in a prompt is perfectly fulfilled. The test suite serves as the ground-truth judge that evaluates if the model has truly followed the pedagogical contract set by the developer.

1. Testing as Requirement Fulfillment

The core philosophy of this module is that a test suite is the executable mirror of the prompt. If a prompt specifies that a function must handle a specific data type, run within 50ms, and avoid using the `regex` library, the test suite must contain dedicated checks for all three.

1.1 The Mapping Principle

Every requirement in your prompt should have a corresponding "test anchor." If the model generates a solution that passes a generic test but fails to meet a specific constraint (e.g., using a forbidden library), the test suite has failed its primary mission. In AI training, a "pass" must mean that the model has followed *all* instructions, not just achieved the functional result. This mapping ensures that the Reinforcement Learning (RL) signal is "clean"—rewarding the model only when it obeys the full set of engineering constraints.

1.2 Scenario-Based Logic Coverage

Models often fail not on the "happy path," but in the "interaction of constraints." For example, a model might correctly implement an *O(n)* algorithm (Requirement A) but fail to handle a null input (Requirement B) simultaneously. Your tests must simulate these intersectional scenarios. You should design tests that combine constraints, such as "processing a maximum-sized input that is also malformed," to ensure the model's logic is robust across the entire problem space defined in the prompt.

2. Test Design for Complex Scenarios

A comprehensive test suite must be tailored to the specific category of the task. Different engineering problems require different verification strategies to ensure the prompt's intent is fully realized.

2.1 Functional and Structural Verification

For feature generation tasks, your tests must go beyond output checking to verify structural compliance. If a prompt requires a class to implement a specific interface or pattern (like the Singleton or Observer pattern), the test suite should use reflection or introspection (e.g., `hasattr` or `isinstance` in Python) to verify that the generated code actually possesses the required architectural traits. This prevents the model from "faking" a behavior through a global variable when a specific structure was requested.

2.2 Performance and Scalability Stress

When a prompt includes efficiency requirements—such as "must process 1 million records in under 1 second"—the test suite must act as a high-fidelity benchmark. You should generate synthetic data that precisely matches the scale mentioned in the prompt. A common failure in AI testing is using a sample that is too small (e.g., 10 records), which allows a *O(n^2)* algorithm to pass even if the requirement was *O(n)*. Tests must trap the model by providing inputs large enough to expose sub-optimal complexity.

2.3 Comprehensive Edge-Case Exhaustion

A good test is one that covers every edge case that might affect the fulfillment of the requirements. This includes:

- **Empty and Extreme Values:** Testing with 0, -1, empty strings, and maximum 64-bit integers.
- **Concurrency and State:** If the prompt mentions thread-safety, the tests must launch multiple threads to probe for race conditions.
- **Environmental Constraints:** If the prompt specifies a certain Python version or library, the tests should verify that no prohibited modern syntax or external dependencies were sneaked in.

3. Meta-Testing: Evaluating the Evaluator

Meta-testing is the process of verifying the quality of the tests themselves, especially when the model's task is to *generate* a test suite. Instead of relying on automated mutation, we perform a deep verification of Requirement Traceability and Implementation Integrity.

- **Requirement Traceability Verification:** ensures that the written tests cover every single logical branch and constraint mentioned in the prompt. A meta-test in this context acts as a requirement auditor. It checks that there is a one-to-one mapping between the prompt's constraints and the test cases. If a prompt mentions "handling invalid API keys," but no test case simulates an invalid key, the test suite is incomplete. This phase validates that the tests provide 100% requirement coverage, which is often more important than simple line coverage in AI training.
- **Implementation Integrity Checks:** focus on whether the tests are written correctly according to the prompt's environmental standards. We verify that the tests are not false positives—tests that pass for the wrong reasons. This involves checking that the test assertions are specific enough to catch subtle logical errors. For instance, a meta-test would flag a test that only checks if a function returns a `list` when it should be checking the specific content of that list.

4. Advanced Adversarial Testing

Adversarial testing is a proactive attempt to break almost correct code. This is the most difficult but highest-value part of the evaluation process.

AI models often provide a solution that works for the provided examples but fails for any variation. Adversarial testing involves input perturbation—slightly modifying a valid input in a way that *should* change the output or cause an error, and verifying that the model's code handles it correctly. For instance, if the model writes a parser, an adversarial test might inject unusual Unicode characters or hidden control sequences to see if the parser crashes or misinterprets the data.

Developers must look for common AI failure modes to inform their adversarial tests. For example,

- Instruction forgetfulness occurs when the model solves the main problem but ignores a do not use X library rule.
- Logical shortcuts happen when the model uses a `try-except` block to hide a bug instead of fixing the underlying logic.
- Hardcoded Answers involve the model detecting specific values in your examples and returning them as hardcoded constants.

Adversarial tests should be designed specifically to catch these "lazy" behaviors, ensuring that the only way for the model to pass is to truly understand and implement the requested logic.