Coding Agent Training Data Quality: Why Passing Tests Isn't Enough

239.AI · 239.AI · Nov 12, 2026

Why can a coding patch pass tests and still be wrong? A closer look at task specifications, verifier errors, reward reliability, and the limits of executable feedback in SWE RL.

In coding-agent evaluations, passing the tests often counts as completing the task. In some reinforcement learning (RL) setups built around executable feedback, that verdict also becomes a reward. Automated tests offer a practical advantage over reviewing every patch by hand: they exercise real software behavior, can be rerun, and scale much more easily.

But **a verifier accepting a patch is not the same as that patch satisfying the full task specification**. When the gap goes unnoticed, benchmarks can overstate an agent’s capabilities, and training can reward code that has not actually done the job.

This is not simply a matter of writing more tests. The specification may be incomplete; the tests may enforce incidental details of a reference implementation; an infrastructure failure may be mistaken for a coding failure; and the kinds of patches an agent produces can change as it learns from feedback.

A Small Bug: Limit First, or Filter First?

Suppose a task-listing API needs to filter records by source and return at most `limit` matches. The contract is explicit: **filter first, then apply the limit**. The following example uses `239.AI` as a fictional source label. It is not code from 239’s systems.

```
def select(rows, source, limit=1):
    return [r for r in rows[:limit] if r["source"] == source]

assert select([{"source": "239.AI"}], "239.AI")  # passes
assert select([{"source": "other"}, {"source": "239.AI"}], "239.AI")  # fails
```

The first assertion passes; the second fails. Instead of filtering the records and then limiting the result, the function takes the first record and filters that slice. A matching record outside the initial `limit` entries is never considered.

If the verifier only included the first case, this incorrect implementation would get a PASS. The test would not be lying: the particular input it checked does return a match. What it misses is **the interaction between filtering and limiting**.

In a real repository, the same pattern can be harder to spot. A new API parameter may break an old default. A cache change may compromise concurrency semantics. An exception-handling fix may change an error type that callers depend on. A local feature can work while the modified system still violates its contract.

What Does a Passing Test Actually Establish?

A software engineering task involves at least three distinct objects: a specification, a candidate implementation, and a test suite. Tests evaluate assertions on selected executions. Correctness is the stronger claim that the implementation satisfies the relevant behavioral contract.

Automated program repair research makes a useful distinction between a *plausible patch*, which passes the available tests, and a *correct patch*, which satisfies the intended behavior. One is an observation about a test suite; the other is a claim about program semantics.

Code coverage does not close that gap. Even if every relevant branch runs, the assertions may never examine operation order, state preservation, or interactions across modules. A test named `test_pagination` may check only the shape of the returned value. **Execution coverage, assertion strength, and requirements coverage are different properties.**

There is a second problem beyond missing tests: the tests may enforce a requirement that was never justified.

Return to the example. If the task merely said “support filtering by source and limiting results,” and neither the existing API nor its documentation established the order of those operations, the intended semantics would be ambiguous. A hidden test that insists on one order cannot, by itself, establish that the other order violates the disclosed task.

That does not mean every detail must appear in the prompt. Reading a repository and discovering existing interface contracts are part of a coding agent's job. The important boundary is whether **a requirement used to reject a solution can reasonably be derived from the task and the repository context available to the solver**. A private expectation known only to the test author should not silently become part of the specification.

Verifiers Can Be Wrong in Both Directions

Discussions of verifier reliability tend to focus on incorrect patches that slip through. The opposite error matters too: rejecting a correct implementation.

Error	Actual implementation	Verifier verdict	Consequence
False acceptance	Violates the task	PASS	Incorrect code receives positive feedback
False rejection	Satisfies the task	FAIL	A valid solution loses positive feedback

False rejection is especially relevant to repository-level work because many tasks admit more than one correct design. Yet a test may demand a particular private helper, internal call sequence, or code structure copied from the reference patch.

For example, filtering and limiting can be performed in memory or pushed down into a database query. If the task does not prescribe the mechanism, both should be judged by their externally observable behavior. A hidden assertion that a private helper must be called turns an implementation choice into an obligation.

A historical pull request is valuable evidence of one working approach, not a complete specification of all acceptable approaches. Differences from the reference patch may be worth investigating, but **different does not mean incorrect**. Conversely, seeing the reference patch pass cannot establish that the tests reject every incorrect alternative.

A credible verifier therefore needs evidence in both directions: it should reject candidates with independently established requirement violations and accept independently established correct alternatives. Those judgments must not simply be inferred from the verifier's own PASS or FAIL result.

Why Rejecting 95% of Wrong Patches May Still Leave Unreliable PASS Results

There is a statistical trap here: **a high rejection rate for incorrect patches does not by itself mean most accepted patches are correct**. The composition of the candidate population matters.

Imagine 1,000 candidate patches. Only 100 are truly correct; 900 are incorrect. The verifier accepts 90 correct patches but also accepts 45 incorrect ones. It rejects 855 of the 900 incorrect patches—a 95% rejection rate. Yet only 90 of its 135 PASS results are correct.

In this **hypothetical** population, roughly 67% of accepted patches are actually correct. This is not an estimate for any real benchmark or dataset. It illustrates why a verifier’ s error-detection rate, the correctness of accepted samples, and the distribution used to measure them must be reported separately.

That distribution is not fixed. Different models, task difficulties, and sampling policies produce different candidates. Even the same model may make different kinds of mistakes before and after training.

When PASS Becomes a Reward, the Error Changes Character

In some outcome-reward SWE RL setups, a verifier’ s PASS is mapped to a positive reward. The intended objective is to complete software engineering tasks correctly; the optimization algorithm, however, acts on the reward it is actually given. When a verifier accepts a patch that violates the requirements, the proxy and the objective diverge.

At first this resembles ordinary label noise. Coding agents make the situation more complicated: they inspect code, run commands, observe feedback, and revise their approach. **The population of candidate patches changes with the agent’ s abilities and search strategy; the verifier’ s failure modes may become more or less common as a result.**

A weaker agent may frequently submit code that does not run at all. A stronger one may satisfy the main requirement but miss subtle compatibility constraints, edge cases, or cross-module interactions. Such partial solutions are more likely to survive tests that exercise only

straightforward paths. Evidence that a suite handles obvious mistakes well does not automatically transfer to stronger candidates.

If optimization repeatedly favors implementations that receive reward without satisfying the task, a verifier blind spot may turn from an occasional mistake into a path the policy encounters more often. That is a conditional risk, not a claim that one mislabeled outcome inevitably degrades a model—or that every incorrect patch constitutes reward hacking.

It helps to distinguish an accidental coding error, overfitting to known test inputs, and active interference with test execution or result collection. All can create unreliable rewards, but they require different responses. More functional assertions do not necessarily prevent report tampering; stronger execution isolation does not fill gaps in behavioral coverage.

Better Evidence Than “The Reference Patch Passes”

A reference implementation passing while the unmodified starting code fails the new requirements is useful baseline evidence. It is not enough to show that the tests distinguish correct code from *partially* correct code. The limit-before-filter bug is a concrete counterexample: it runs, passes a simple test, and violates the contract only when two behaviors interact.

A more informative question is: **will the same verifier reject a candidate known, on independent grounds, to violate a particular requirement while otherwise behaving normally?** The rejection should be attributable to that requirement, not to a syntax error, a broken environment, or tests that never ran.

The reverse check matters as well: will the verifier accept a substantially different implementation that is independently known to satisfy the contract? This can reveal unwanted dependence on the reference patch’ s internal structure.

Neither check is a universal proof. Rejecting one wrong patch says nothing definitive about every other wrong patch. Accepting one alternative correct implementation cannot rule out all false rejections. Reliable reporting should identify the requirements and failure classes for which evidence exists, rather than present finite checks as absolute correctness.

FAIL Is Not the Same as ERROR

A patch may fail because it violates a functional requirement. It may also fail because dependencies cannot be installed, tests were not collected, a process timed out, a report is missing, or repeated executions produce inconsistent results.

A FAIL is useful negative evidence only when the verifier actually completed the intended checks and observed a relevant assertion failure. If tests never ran, an exit code says little about the patch’ s semantics. For trustworthy executable feedback, at least three outcomes need to remain distinct:

- **PASS:** The expected checks completed and accepted the candidate.
- **FAIL:** The expected checks completed and found one or more failing assertions.
- **ERROR:** Execution or result collection was insufficient for a reliable functional verdict.

PASS and FAIL still refer to the chosen tests; neither is an oracle for every possible behavior. Collapsing ERROR into a reward of zero confuses infrastructure failure with a model capability failure. Trusting an isolated report file without establishing what actually ran can be equally misleading.

Reliable Verification Is Only Part of Training Data Quality

Even with an unambiguous task, a sound verifier, and trustworthy execution, a sample may not be useful for a particular model’ s training objective.

A large collection of one-line conditional fixes may have accurate rewards yet contribute little to cross-module reasoning or complex repository navigation. At the other extreme, tasks far beyond the target model’ s current reach may offer limited useful feedback, even if each failure is judged correctly. Difficulty is relative to the model, its tools, and its budget—not merely the number of changed lines.

Distribution matters too. Highly repetitive tasks can cover a narrow slice of engineering behavior. Tasks drawn from public repositories may overlap with existing training material or evaluations, complicating claims about transferable improvement.

When a dataset includes complete agent trajectories, there is another distinction: a correct final patch does not imply every preceding action deserves imitation. A successful run can contain unproductive searches, mistaken edits, and lucky recoveries.

Three claims should therefore stay separate: **the task is valid, the feedback is trustworthy, and the sample is useful for the intended training objective**. The first two can be investigated through specifications, candidate behavior, and execution evidence. The last also requires attention to the target model, data distribution, and independent post-training evaluation.

Automated tests remain indispensable. The mistake is allowing a finite PASS observation to stand in for proof of code correctness, reward integrity, or training benefit. For coding-agent data, the useful question is not how confidently a label is printed, but **what evidence justifies that label—and where that evidence stops**.

Original article: https://239.ai/en/posts/coding-agent-training-data-quality/

239.AI

You may republish articles from this site without requesting separate permission. Credit the author and include a link to the original article.