

RESEARCH MARCH 11, 2026

AGENTIC RUBRICS: TEACHING AI TO VERIFY CODE THE WAY DEVELOPERS DO

By Mohit Raghavendra, Anisha Gunjal, Bing Liu, Yunzhong He

TL;DR

- We introduce **Agentic Rubrics**, a verification method where an expert agent explores a repository and produces structured, context-grounded checklists for scoring candidate patches, without executing any tests.
- On SWE-Bench Verified (K=16 rollouts), Agentic Rubrics achieve **54.2%** on Qwen3-Coder-30B-A3B and **40.6%** on Qwen3-32B, outperforming all baselines by **+3.5 points or more**.
- Rubric scores align strongly with ground-truth tests (ROC-AUC=0.886) while surfacing failure modes that tests miss entirely.
- Repository interaction during rubric generation is what makes this work. Remove it, and performance drops significantly.

Read the full paper [here](#).

The Verification Bottleneck

Building AI agents that write code is hard. But there's a less obvious problem that matters just as much: once an agent produces a patch, how do you know it's correct?

Verification sits at the center of two capabilities the field urgently needs. First, it provides the reward signal for reinforcement learning. A good reward signal helps train agents to write better code, by reliably distinguishing good responses from bad ones. Second, it enables test-time scaling: generate multiple candidate patches, or leverage multiple agents to work in parallel, and pick the best one. The quality of your verifier directly determines how much value you extract from additional compute at inference time.

The dominant approach is execution-based: run the patch against a test suite and check if it passes. This works, but it's expensive, and doesn't cover everything. Each problem instance requires a sandboxed environment, dependency installation, and test execution. The signal is also surprisingly sparse: a test either passes or fails, telling you nothing about *how close* a failing patch came or *why* it failed. New issues that need test generation inherit all the brittleness of automated testing.

The alternative is asking an LLM to judge a patch directly, which is cheap but shallow. Without grounding in the actual codebase, these judges rely on surface-level cues: Does the patch look reasonable? Does it touch plausible files? This is roughly equivalent to reviewing a pull request by reading only the diff and the issue title.

Neither approach mirrors how experienced developers actually verify code.

How Developers Think About Correctness

When a senior engineer reviews a patch for a bug they haven't seen before, they don't start by running tests. They start by *understanding the problem in context*. They read the issue, trace the relevant code paths, check how similar patterns are handled elsewhere in the codebase, and mentally construct a set of criteria: Does this patch modify the right files? Does it respect the existing interfaces? Does it actually address the root cause, or just suppress the symptom? Could it break something else? They then review code with this mental model.

Agentic Rubrics formalize this workflow. Instead of a human building a mental model of correctness, an agent explores the repository, gathers context, and produces an explicit, structured rubric. Subsequent patches are then scored against this rubric by a separate judge with no test execution required. This offers a scalable oversight mechanism to evaluate the outputs of coding agents.

How It Works

The pipeline has two phases, rubric generation and rubric grading.

Rubric Generation: A rubric agent receives the problem statement and full access to the repository through a standard coding scaffold, like SWE-Agent or OpenHands. It can search files, read code, inspect directory structures, anything a developer would do when familiarizing themselves with an unfamiliar codebase. After exploration, it produces a rubrics file: a structured checklist organized along four axes.

- **File Change:** Are the edits minimal, correctly scoped, and sufficient? Does the patch avoid unnecessary modifications?
- **Spec Alignment:** Does the patch satisfy the requirements stated in the issue?
- **Integrity:** Does it avoid shortcuts like hardcoding expected values, disabling tests, or bypassing validation?
- **Runtime:** Will the patch execute correctly? Are there import issues, type mismatches, or unhandled edge cases?

Each item is a natural-language criterion with an importance weight.

Rubric Grading: A separate LLM judge scores each candidate patch against the rubric, assigning a binary pass/fail per criterion. The weighted aggregate produces a score in $[0, 1]$ used to rank candidates.

Generating rubrics requires deep reasoning and repository exploration, necessitating a frontier model. However, it is a one time investment that gets amortized every time the task is used. Grading against rubrics is comparatively straightforward, so that even a lightweight model performs nearly as well as a frontier one, thus saving on costs at every use.

What We Found

We evaluated Agentic Rubrics on 500 problems from SWE-Bench Verified, generating $K=16$ independent candidate patches per problem from two generators (Qwen3-32B and Qwen3-Coder-30B-A3B). The verifier's job: pick the best patch.

Agentic Rubrics outperform everything else

Across both generators, Agentic Rubrics achieve the highest Best@16 resolution rates, beating both execution-free baselines (self-consistency, patch classifiers) and agentic baselines that generate their own artifacts (test suites, reference patches).

Against the best non-agentic baseline (Patch Classifier), rubrics gain +3.5 points on Qwen3-32B and +4.0 on Qwen3-Coder. Against Agentic Patch Similarity, the gains are +5.6 and +4.6 points respectively.

Why do rubrics outperform even agentic test generation? Generated tests must be syntactically valid, execute cleanly, and discriminate correctly between good and bad patches. Reference patches suffer a different problem: a semantically correct fix that takes a different approach than the reference will score poorly on similarity.

Rubrics sidestep both failure modes by stating *what* correctness looks like rather than encoding it in executable or reference artifacts.

Rubric Scores are Meaningful, Not Just Predictive

It's one thing for a verifier to pick winners. It's more interesting when its scores are *interpretable*.

Ground-truth-passing patches concentrate near scores of 0.85–1.0. Failing patches spread across 0.4–0.5. But the axis-level breakdown tells a richer story. Incorrect patches score low on File Change and Spec Alignment.

They edit the wrong files or miss requirements. But they often score well on Integrity, meaning they're not cheating; they're genuinely *trying* and falling short. Correct patches nearly saturate Spec Alignment and Integrity but occasionally lose points on File Change due to over-scoped edits.

Rubrics Catch What Tests Miss

Perhaps the most striking finding: when rubrics disagree with ground-truth tests, they're often *right*.

We analyzed cases where tests accepted a patch but rubrics scored it below our acceptance threshold. In 54% of these disagreements, the rubric failures were substantive. They flagged issues like root causes missed (the patch suppresses the symptom but doesn't fix the underlying bug) and missing edge cases that the test suite doesn't cover.

When rubrics and tests agree, 78% of rubric judgments target core semantics, API compatibility, and structural correctness. This evaluates the core components of what makes a patch right or wrong.

What We Learned

Beyond the headline results, several findings shaped our understanding of why this approach works and where it breaks.

Repository grounding is not optional

We ablated the agentic component by generating rubrics from the problem statement alone. We used the same model, same rubric format, and took away the repository access, performance dropped by 4.0 points on Qwen3-32B rollouts and 1.4 on Qwen3-Coder.

The qualitative difference is stark. An agentic rubric for a Django issue might specify: *"The patch must modify `django/db/models/query.py` and preserve the existing `QuerySet.filter()` interface while correctly handling the edge case where `related_name` is `None`."*

A non-agentic rubric for the same issue says: *"The patch should touch the right code path and fix the filtering behavior."* The former makes grading easy, and unambiguous, while the latter needs the judge to be capable enough to understand and interpret correctness from it.

Better Models Write More Granular Rubrics

We tested rubric generation across a range of models, from Qwen3-32B to Claude Opus-4.5. Frontier coding models produce substantially more rubric items per instance (>20 for Sonnet-4.5 vs. ~10 for Qwen3-32B), and they align more closely with ground-truth patches.

Granularity leads to finer discrimination between candidates. Rubrics can distinguish a patch that gets 18 of 22 things right from one that gets 15. With only 5 items, both patches might look identical.

The Best@16 performance follows accordingly: frontier models as rubric agents reach ~54%, open-weight models ~45%, and non-coding models ~43%.

What's Next?

There are several interesting future work this work unlocks:

1. Integrating rubric signals into training loops as reward signals for reinforcement learning is extremely exciting. The benefits are clear and the granular, interpretable reward signals lends itself to this task.
2. We fine-tuned Qwen3-32B on just 2,000 rubric-generation trajectories distilled from Sonnet-4.5, as a proof-of-concept of an open-weight agentic rubric generator that works well. Structured, context-grounded agentic rubric generation is a learnable skill, and we hope we inspire more work in this.
3. Rubric quality also has room to grow. While the majority of generated judgments are substantive, a subset falls into low-utility modes: over-specification (demanding implementation details beyond what correctness requires), redundancy, and occasional mismatches with ground-truth intent. Human-in-the-loop refinements like lightweight review, template reuse, targeted failure-mode prompts could improve fidelity while preserving the auditability that makes rubrics attractive in the first place.