# Agentic Rubrics as Contextual Verifiers for SWE Agents

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

Scale AI
*Equal Contribution
**Correspondence:** mohit.raghavendra@scale.com

## Abstract

Verification is critical for improving agents: it provides the reward signal for Reinforcement Learning and enables inference-time gains through Test-Time Scaling (TTS). Despite its importance, verification in *software engineering (SWE)* agent settings often relies on code execution, which can be difficult to scale due to environment setup overhead. Scalable alternatives such as patch classifiers and heuristic methods exist, but they are less grounded in codebase context and harder to interpret. To this end, we explore **Agentic Rubrics**: an expert agent interacts with the repository to create a context-grounded rubric checklist, and candidate patches are then scored against it without requiring test execution. On SWE-Bench Verified under parallel TTS evaluation, Agentic Rubrics achieve a score of **54.2%** on `Qwen3-Coder-30B-A3B` and **40.6%** on `Qwen3-32B`, with at least a **+3.5** percentage-point gain over the strongest baseline in our comparison set. We further analyze rubric behavior, showing that rubric scores are consistent with ground-truth tests while also flagging issues that tests do not capture. Our ablations show that agentic context gathering is essential for producing codebase-specific, unambiguous criteria. Together, these results suggest that Agentic Rubrics provide an efficient, scalable, and granular verification signal for SWE agents.

## 1 Introduction

Large Language Models (LLMs) have rapidly advanced on coding tasks, enabling increasingly capable software engineering (SWE) agents for realistic code editing and bug fixing (Yang et al., 2024; Hui et al., 2024; Wang et al., 2024). A central bottleneck in training and evaluating such agents is *verification*: determining whether a candidate patch is correct, complete, safe, and aligned with the intended behavior. *Verifier's Law* links the ease of training AI systems on a task to the efficiency and reliability of verifying candidate solutions (Wei, 2025). In SWE agent, strong verification plays a dual role. It provides supervision for post-training with verifiable rewards (OLMo et al., 2025), and it improves inference through test-time scaling by sampling multiple candidates and selecting the best one using a verifier (Brown et al., 2024).

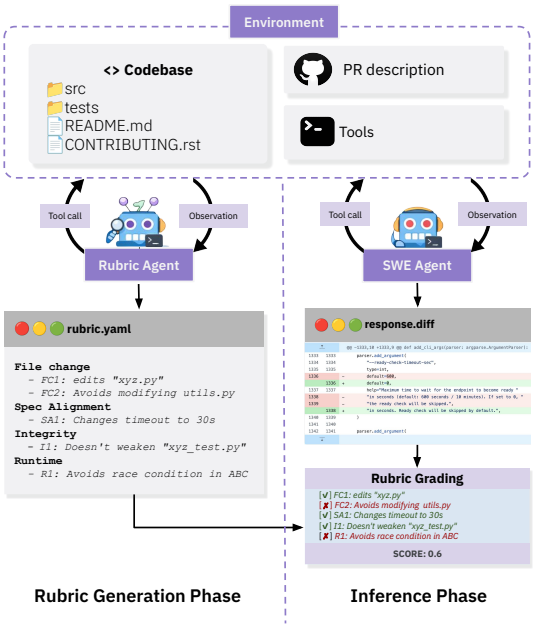



Figure 1: **Agentic rubric pipeline. (a) Rubric generation (left):** given two inputs—the *codebase repository* and the *problem statement* (PR description)—a rubric agent uses repository tools (file navigation, search, and lightweight commands) to inspect task-relevant code and synthesizes a structured rubric checklist organized along four axes: *File Change*, *Spec Alignment*, *Integrity*, and *Runtime*. The rubric is generated *once per task instance* and reused across all candidates. **(b) Inference (right):** a SWE agent proposes a candidate patch; an LLM judge grades the patch against each rubric criterion and aggregates these scores into a single execution-free verifier score, which is used to rank candidates and select a final submission.

Current approaches use a range of verifiers,



*Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15265–15290

including unit tests (human or LLM-generated), learned patch classifiers, similarity metrics, and LLM judges (Luo et al.; Wei et al., 2025a; Jain et al., 2025; Wei et al., 2025b). Verification via code execution is environment-aware, but can be costly to scale due to per-instance setup (e.g., sandbox initialization), and may yield sparse or brittle signals, including limited distinguishability and test toxicity (Ehrlich et al., 2025; Jain et al., 2025). In contrast, execution-free signals are operationally lightweight, but can be less reliable (Crupi et al., 2025), less interpretable, and prone to shallow cues. As SWE agents expand to more open-ended, goal-driven tasks and long-tail repositories, verifiers must become both scalable and codebase-specific.

To close this gap, we explore **Agentic Rubrics**. In our setup, illustrated in Figure 1, an expert rubric agent first interacts with a sandboxed repository to synthesize *context-grounded* rubric criteria; after rubric generation, candidate patches are scored without executing code, enabling scalable verification. We build on rubric-based verification (Shao et al., 2025; Wu et al., 2025), which decomposes correctness into interpretable criteria that capture partial progress and surface failure modes. For SWE, rubrics written from the problem statement alone are often under-specified because they lack repository-specific context. Our rubric generation is therefore *agentic*: the verifier actively explores the repository to ground criteria in relevant code paths, interfaces, and project conventions, yielding rubric items that are more specific and consistently gradable. We evaluate Agentic Rubrics via best-of-$K$ selection under parallel test-time scaling on SWE-Bench Verified, ablate different design decisions and provide detailed analyses of rubric alignment and utility.

Our contributions are: (1) We study **Agentic Rubrics**, a repository-grounded rubric generation paradigm with execution-free scoring for patch selection and post-training. (2) We show that Agentic Rubrics consistently outperform strong test-based and execution-free verifier baselines under parallel test-time scaling on SWE-Bench Verified. (3) We analyze why Agentic Rubrics work, demonstrating alignment with ground-truth tests and showing that rubrics surface diagnostic concerns (e.g., unnecessary edits or missing edge-case handling) even when tests pass. (4) We demonstrate that agentic rubric generation can be distilled into smaller open-weight models, enabling scalable deployment.

## 2 Preliminaries

### 2.1 Verification for SWE Agents

We consider a *verifier* as a procedure that assigns a score to a candidate patch for a given issue, with the goal of selecting or training toward higher-quality solutions. Prior work in SWE Agent settings commonly uses two broad classes of verification signals. **Execution-based** methods verify patches by executing code, most often by running unit tests (human-authored ground-truth or LLM-generated) (Ehrlich et al., 2025). **Execution-free** methods assess patch quality without running the repository, by reranking candidates using learned patch classifiers/verifiers, similarity metrics, or LLM judges (Wei et al., 2025a). These approaches occupy different points in the trade-off space between repository grounding, operational cost, and reliability (Jain et al., 2025). Execution-based verification is environment-aware but can require per-instance setup (e.g., sandbox initialization) and may yield sparse or brittle signals (e.g., limited distinguishability or test toxicity). Execution-free verification is operationally lightweight, but can be less reliable (Crupi et al., 2025), less interpretable, and sometimes sensitive to surface-level cues (e.g., stylistic patterns, non-semantic artifacts) rather than functional correctness.

### 2.2 Rubric-based Verification

A *rubric* verifies a candidate patch by decomposing correctness into a small set of explicit criteria (Arora et al., 2025). Concretely, a rubric consists of criteria texts (optionally grouped by axes) with per-criterion weights, and a scoring rule that aggregates criterion-level judgments into a single verifier score. Given a problem and a candidate patch, a judge assigns each criterion a score (e.g., binary or graded) and aggregates them to obtain an overall patch score used for selection or learning.

For SWE tasks, a key practical consideration is *grounding*. Criteria written solely from the problem statement can omit repository-specific interfaces, constraints, and conventions, which makes judgments less precise and less consistent across patches. This motivates verifiers whose criteria are grounded in the right task-relevant repository context, while still allowing lightweight scoring once criteria are generated.

## 3 Experimental Design

### 3.1 Agentic Rubrics

**Rubric Generation.** We implement a rubric-generation agent on top of the SWE Agent scaffold, which provides tools for repository navigation, file inspection/editing, and shell command execution (Yang et al., 2024; Wang et al., 2024). The agent receives two inputs: (1) the *codebase repository* (accessed via the sandboxed environment) and (2) the *problem statement*, i.e., the GitHub issue description. We modify the scaffold's `SYSTEM PROMPT`, instructing the agent to explore the repository, gather task-relevant context, and produce a patch that adds a structured rubric file, `rubrics.yaml` (prompt in Appendix A.10). Crucially, the rubric is generated *once per task instance* and then reused to score all $K$ candidate patches in the BEST@K setting, amortizing the generation cost across candidates. This workflow mirrors how developers validate fixes when comprehensive tests are unavailable: inspecting surrounding code and contracts, tracing call sites, and reasoning about edge cases.

Each rubric item is a tuple $(t_i, w_i)$ consisting of a short natural-language criterion $t_i$ and an importance weight $w_i \in \{1, 2, 3\}$ (nice-to-have / important / must-have), and is assigned to one of the following *axes*:
(i) **File Change** (4–8 items): edits are minimal, local, and sufficient for the fix;
(ii) **Spec Alignment** (3–6): the patch satisfies the requirements in the issue description;
(iii) **Integrity** (3–6): "no-cheating" and hygiene constraints (e.g., no test weakening, broad refactors, mass renames, or dependency churn);
(iv) **Runtime** (3–6): the changes imply the intended runtime behavior and avoid obvious execution-time issues.
We parse the submitted `rubrics.yaml`; if parsing fails, we consider the generation attempt invalid.

The rubric axes were hand-crafted by the authors based on frequent failure modes of coding models, and we let the agent generating the rubric assign weights to the rubric items, to mimic experts determining the importance of rubrics. We leave a more thorough analysis of rubric creation and weighting for future work.

**Rubric Grading.** Given a problem and a candidate patch, an LLM judge assigns each rubric item a binary score $s_i \in \{0, 1\}$. We aggregate scores as $S = \frac{\sum_i w_i s_i}{\sum_i w_i}$, yielding a verifier score $S \in [0, 1]$ used for reranking candidates.

### 3.2 Test-time Scaling with Agentic Rubrics

**Setup** Given a SWE problem statement description $D$, a SWE agent produces a rollout trajectory $T^{(j)}$ and candidate patch $P^{(j)}$ for $j = 1, \ldots, K = 16$ independent rollouts. The verifier's goal is to assign a score $S^{(j)} \in [0, 1]$ to each candidate. These are then reranked to select the best candidate patch.

**Candidate patch generation** We use the SWE-Agent scaffold by Yang et al. (2024) as the agent harness for the coding model to interact with the repository in a sandboxed environment to generate patch. For each of the 500 SWE-Bench Verified problems (OpenAI, 2024), we sample 16 independent rollouts and extract candidate patches, from a fixed generator model. We run experiments with two generators: `Qwen3-32B` (Instruct version, max 30 turns) and `Qwen3-Coder-30B-A3B` (max 50 turns), both at temperature 1.0 (Yang et al., 2025).

**Evaluation protocol (BEST@K).** A problem is considered *resolved* if the candidate patch passes the ground-truth Fail-To-Pass and Pass-to-Pass tests. We score all $K$ candidates and select the highest-scoring patch for BEST@K resolution calculation. In cases where $K < 16$, we repeat sample 100 trials to make this robust. As reference points, we report ORACLE PASS@K (selecting using ground-truth tests; an upper bound) and RANDOM@K (uniform selection).

### 3.3 Baselines

We group baseline verifiers into two categories based on whether they rely on an *externally generated verification artifact* (e.g., tests, a reference patch, or a rubric) produced via repository interaction. **Non-agentic verifiers** score candidate patches directly from the problem statement and patch, without generating any additional artifact or inspecting the repository. **Agentic verifiers** first interact with the repository via an agentic scaffold to produce an artifact that is then used to score and re-rank candidates. We select baselines that represent the strongest methods in their respective categories from recent SWE-agent literature: Agentic Tests for execution-based verification (Jain et al., 2025), Patch Similarity for execution-free repository/patch-signal methods (Singhi et al., 2025; Wei et al., 2025a), and Patch Classifiers for learned verifiers (Pan et al.,

| Method | Execution Free | Expert Artifact | Qwen3 32B | Qwen3 Coder |
|---|---|---|---|---|
| Oracle Pass@16 | – | – | 51.4 | 65.6 |
| Random Pass@16 | – | – | 22.6 | 39.6 |
| **Non-Agentic Verifiers** | | | | |
| Self-Consistency | ✓ | – | 33.2 | 47.6 |
| Patch Classifier | ✓ | – | 37.1 | 50.2 |
| **Agentic Verifiers** | | | | |
| Agentic Tests | ✗ | Tests | 33.6 | 49.0 |
| Agentic Patch Similarity | ✓ | Patch | 35.0 | 49.6 |
| **Agentic Rubrics** (ours) | ✓ | Rubric | **40.6** | **54.2** |

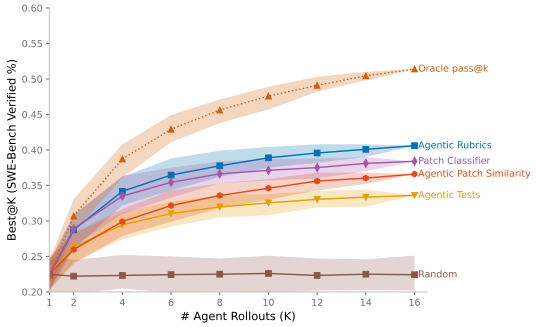



Figure 2: (Left) Best@16 resolution (%) with $K = 16$ rollouts for `Qwen3-32B` and `Qwen3-Coder-30B-A3B`. Verifier signals are generated with `Claude Sonnet-4.5`; LLM judging uses `GPT-5 (low reasoning)`. (Right) Best@K scaling curves for `Qwen3-32B` rollouts under different verifiers, with numbers averaged over 100 trials.

2024; Jain et al., 2025). Prompts used for all the baselines are provided in Appendix A.10.

**Non-agentic Verifiers (no artifact).**

(i) *Self-Consistency* (Wang et al., 2023; Wei et al., 2025a; Singhi et al., 2025): select the candidate patch whose diff has the highest average similarity to the remaining $K-1$ candidates.[1]

(ii) *Patch Classifier* (Pan et al., 2024; Jain et al., 2025): an LLM judge predicts patch correctness and outputs a continuous score in $[0, 1]$.

**Agentic Verifiers (artifact-based).**

(i) *Agentic Tests* (Jain et al., 2025): an expert agent generates a problem-specific `test_issue.py` with repository interaction; candidates are scored by executing these testcases.

(ii) *Agentic Patch Similarity*: an expert agent generates a context-grounded *proxy reference patch*; candidates are reranked by similarity to this patch (scored by an LLM judge on a 1–5 scale).

(iii) ***Agentic Rubrics (our method)***: an expert agent gathers repository context and synthesizes a structured `rubrics.yaml`; candidates are graded against rubric criteria to obtain a final verifier score.

**Implementation details.** For methods that require a verification artifact (tests, proxy patch, or rubrics), we use `Claude Sonnet-4.5` as the expert agent (30-turn budget) to generate the artifact via repository interaction; whenever scoring requires an LLM judge (patch classification, similarity scoring, rubric grading), we use `GPT-5 (low reasoning)`. All verifiers run in the SWE-Bench Verified sandbox with the repository reset to a pre-PR snapshot: agents may inspect the codebase and existing tests, but cannot access or execute the hidden ground-truth evaluation tests, reference patches, or git history. If an artifact wasn't correctly produced (e.g., missing `test_issue.py` or invalid `rubrics.yaml`), we assign a score of 0. Prior work shows that verifier decisions can be unduly influenced by agent's thinking trace (Jain et al., 2025). So to keep evaluation uniform and reduce verifier hacking, scoring for all methods use only on the problem statement, verifier artifact and the final submitted patch, not the full rollout trajectory or tool traces.

[1]We compute similarity using `difflib`'s `SequenceMatcher.ratio()` between unified-diff strings.

## 4 Results

### 4.1 Test Time scaling with Agentic Rubrics

**Agentic Rubrics improve BEST@K selection.** Figure 2 reports BEST@16 for rollouts from two generator models, `Qwen3-32B` and `Qwen3-Coder`, grouping verifiers into *non-agentic* methods that score patches directly (no artifact) and *agentic* methods that first generate a verification artifact (tests, a proxy patch, or a rubric) via repository interaction. The right panel shows how BEST@K scales with $K$ for `Qwen3-32B`.

At K=16, *Agentic Rubrics* is the top-performing verifier in both settings. For `Qwen3-32B`, Agentic Rubrics achieves 40.6% BEST@16, improving over the best performing non-agentic baseline (Patch Classifier, 37.1%) by +3.5 points and over the strongest artifact-based alternative (Agentic Patch Similarity, 35.0%) by +4.6 points. For `Qwen3-Coder-30B-A3B`, Agentic Rubrics attains 54.2%, improving over the best non-agentic baseline (50.2%) by +4.0 points and over the best agentic baseline (49.6%) by +4.6 points. The scaling curve further shows that rubric-based scoring maintains an advantage as K increases, indicating that the gain is not confined to a single operating point.

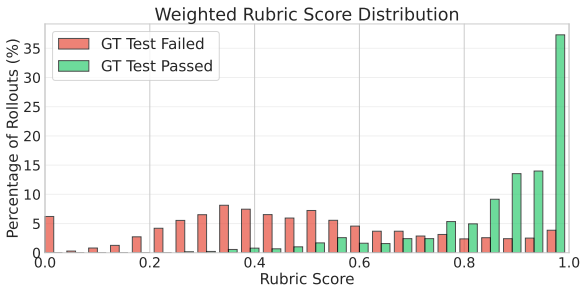



Figure 3: Distribution of Weighted Rubric score for `Qwen3-32B` rollouts on `Sonnet-4.5` generated agentic rubrics, for both correct (Ground Truth Tests Pass - Green) and incorrect (Ground-Truth tests Fail - Red). Rubric scores are well aligned with the GT Test correctness signal, awarding lower score for incorrect patches and higher score for correct ones, while providing a denser score distribution.

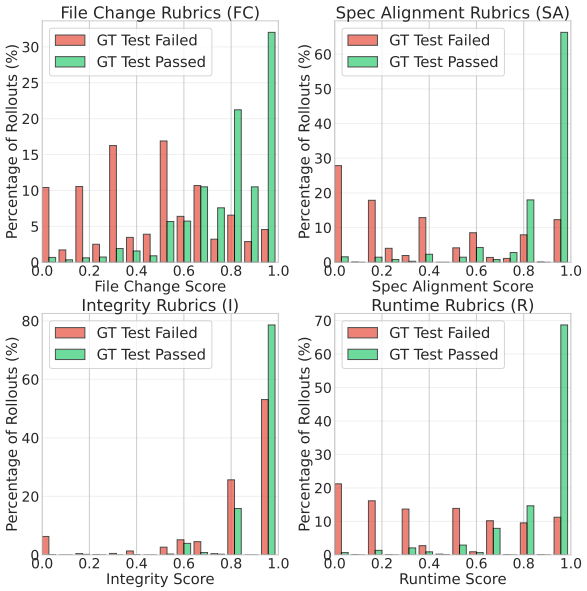



Figure 4: Category-wise distribution of `Sonnet-4.5` rubric scores on `Qwen3-32B` rollouts. Incorrect patches (GT Test Failed, in red) score lower on File Change (Edit scope) and Spec Alignment (Satisfying prompt requirements) and Runtime issues, but still good preserving codebase integrity and avoid cheating. Patches that pass ground-truth tests (GT Test Passed, in green) have a very high spec-alignment and integrity score but still suffer from edit scope and in some cases, have issues in runtime checks.

**Rubrics provide most effective agentic signal.** While other agentic baselines also inject repository context, they rely on more brittle intermediate steps. *Agentic Tests* must generate *runnable* tests in the sandbox (including setup/compilation) and those tests must cleanly discriminate between candidates. *Agentic Patch Similarity* scores "closeness" to a proxy reference patch, which can underrank semantically correct but stylistically different fixes. Rubric artifacts instead use repository interaction to state *what should hold* (file change, spec alignment, integrity, runtime) and then score candidates execution-free against these criteria, yielding a more robust and interpretable scoring signal.

We provide example problems, rubrics, and their grading of responses in Appendix A.9.

### 4.2 Analysis of Agentic Rubrics

We further examine rubric-based verification beyond aggregate BEST@K. We first study *score alignment*: whether agentic rubric scores agree with ground-truth tests and patches in §4.2.1. We then audit *utility* by categorizing judgments into high- vs. low-utility modes, especially when rubrics are stricter than tests, to understand when they add signal beyond the available tests in §4.2.2.

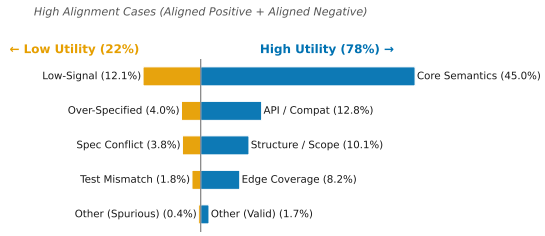
#### 4.2.1 Rubric Score Alignment Analysis

**Rubric scores separate passing vs. failing patches.** Figure 3 plots `Sonnet-4.5`'s weighted rubric score distribution on SWE-Bench Verified rollouts by `Qwen3-32B`, split by whether the candidate patch passes the GT test suite. Rubric scores for GT-Pass rollouts concentrate near high rubric scores (typically 0.85–1.0), while GT-Fail rollouts receive much lower scores on average and spread across a wide range (often around 0.4–0.5). This spread suggests that rubrics can distinguish partial progress from fully-correct solutions, rather than providing only a binary signal like test-suite pass/-fail. Quantitatively, Rubric scores have an ROC-AUC score of **0.886** and PR-AUC of **0.722** against GT test Pass/Fail prediction. High PR-AUC suggests that rubrics prioritize true GT-passing patches in the high-precision regime, consistent with providing a more informative graded signal.

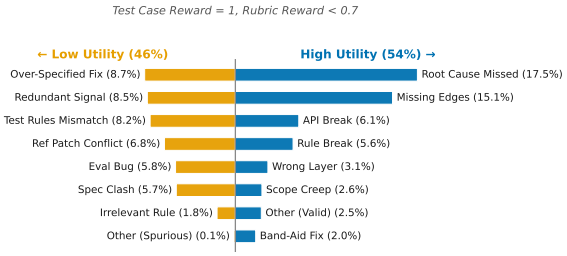
Figure 4 dissects this across *File Change* (scope), *Spec Alignment*, *Integrity*, and *Runtime*. GT-Failing patches tend to score lower because they make unnecessary edits (*File Change*), miss requirements (*Spec Alignment*), or have runtime issues (*Runtime*), while often remaining strong on *Integrity*. For GT-passing patches, *Spec Alignment* and *Integrity* are near-saturated, but we still see penalties from over-scoped edits and occasional runtime-check issues.

**Ground-Truth Patch Agreement** We also score human-written *Ground-Truth patches* from the original pull requests, which all pass the SWE-Bench Verified Ground-Truth tests in Appendix A.1. Ta-

ble 3 and Figure 8 shows that GT patches receive consistently high rubric scores (mean $> 0.8$) across axes from frontier models, suggesting that expert-generated rubrics are broadly compatible with high-quality human fixes. One recurring exception is the *File Change* axis, where rubrics can be more prescriptive about exact edit location/scope than the GT implementation.




(a) High-alignment cases (ground-truth test case reward = 1, rubric reward >= 0.7 threshold).




(b) Low alignment cases (ground-truth test case reward = 1, rubric reward < 0.7 threshold).

Figure 5: Qualitative breakdown of Agentic Rubric utility relative to SWE-Bench Verified ground-truth tests. (a) In high-alignment cases, 78% of rubrics are high-utility (core semantics, API/compatibility, structure, edge coverage) and 22% are low-utility (low-signal, over-specified, spec- or test-mismatched). (b) When tests pass but rubric scores are low, 54% of rubric failures are high-utility (often missed root causes or edge-case coverage), while 46% reflect low-utility mode.

**Takeaway** *Rubric signals are highly correlated with human written Ground Truth tests and patches. Rubric scores also provide a denser signal than test pass/fail by assigning intermediate credit to partially-correct patches and provides detailed technical feedback across different axes.*

#### 4.2.2 Rubric Utility Analysis

Since rubrics are generated synthetically without a canonical correctness check, we further scrutinize them for true utility versus spurious signals. We study *when* rubric judgments are useful by labeling them as *high-utility* (spec-consistent, semantically meaningful checks such as core semantics, API/-compatibility, structure/scope, edge coverage) or *low-utility* (redundant, over-prescriptive, or misaligned). For a subset of 100 SWE-Bench Verified instances, we prompt `GPT-5 (medium reasoning)` to assign each case a High-/Low-Utility tag and a sub-category from Table 5, using the problem statement and Ground-Truth tests and patches as the reference spec (refer Appendix A.12).

**When rubrics and tests agree.** Figure 5(a) aggregates cases where rubric and test outcomes agree (both accept or both reject, using a rubric acceptance threshold of 0.7). In this regime, **78%** of rubric judgments are high-utility, primarily reflecting *Core Semantics*, *API/Compatibility*, *Structure/Scope*, and *Edge Coverage*. The remaining **22%** are low-utility (e.g., low-signal, over-specified, test-mismatched rubrics, etc.)

**When rubrics are stricter than tests.** Figure 5(b) considers cases where tests accept a patch but the rubric score is below 0.7. Even here, **54%** of rubric failures are high-utility, most often flagging *Root Cause Missed* and *Missing Edges* - issues that may not be covered by the available GT tests. The remaining **46%** low-utility cases highlight failure modes like over-specified fixes, redundant signals, and rubric–test mismatches, which can be mitigated using human-in-the-loop rubric refinement in future work.

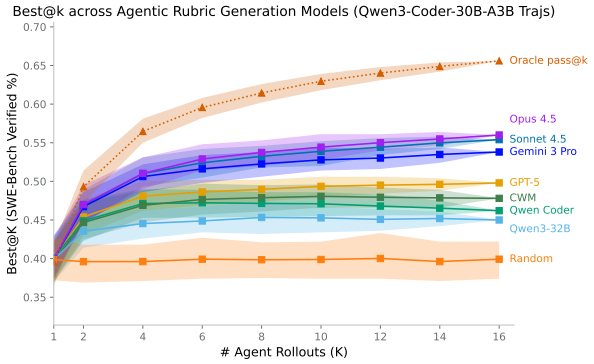
**Takeaway** *Across both regimes, most rubric judgments are* substantive*: when rubrics agree with GT-tests, they do so mainly for core semantic and interface reasons (78% high-utility), & when rubrics disagree by rejecting GT-test passing patches, over half of these rejections (54%) flag plausible under-tested issues (root cause missed or missing edges).*

## 5 Ablations

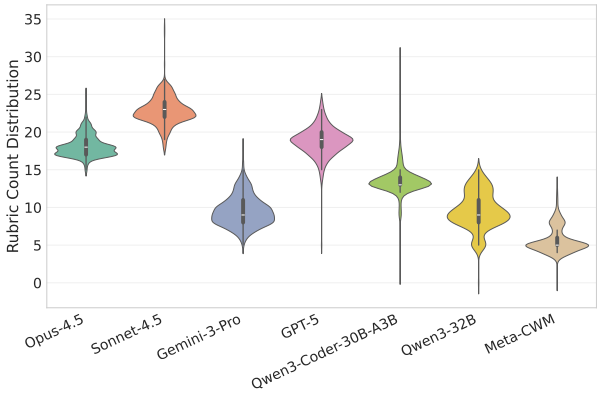
In this section, we ablate key components of the Agentic Rubrics pipeline to isolate the impact of (i) the rubric-agent model, (ii) repository context gathering during rubric construction, and (iii) the judge model used for rubric grading.

### 5.1 Rubric-Agent Model Choice

We investigate the performance of various frontier and Open-weight models in generating rubrics, by studying their test-time performance on rollouts by a fixed policy model (`Qwen3-Coder-30B-A3B`) and a fixed judge model (`GPT-5 low reasoning`). The models use their default reasoning effort when applicable.




(a) Comparing models as rubric creation agents




(b) Rubric Count Distribution

Figure 6: (a) TTS of rubrics generated by various frontier expert and open models, on rollouts by Qwen-Coder-30B-A3B. (b) Distribution of rubric count per instance generated by various models.

Figure 6(a) shows the BEST@16 resolution rate when using each model's rubrics on SWE-Bench Verified. We see that the capability of the rubric generation model directly impacts their TTS performance. Frontier coding models (`Claude Opus-4.5`, `Claude Sonnet-4.5` and `Gemini-3-Pro`) achieve the highest BEST@16 resolution rates of 54%. Open-weight coding models like `Qwen3-Coder-30B-A3B` and `Code World Model` are not as effective ($\tilde{4}5\%$), and finally, non-coding agentic model like `Qwen3-32B` rubrics yield around 43%. Figure 6(b) provides some insight into this performance gap: more capable models generate sometimes substantially more rubrics per instance. For instance, `Sonnet-4.5` averages over 20 rubrics per instance, twice that of `Qwen3-32B` and `CWM`, although exceptions like `Gemini-3-Pro` exist. Increased granularity can enable finer-grained differentiation between candidate solutions, explaining the correlation between model capability and selection performance. In addition, rubrics from

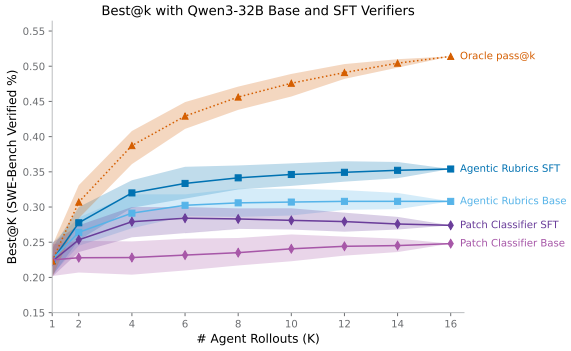



Figure 7: Finetuning (SFT) open-weight models like `Qwen3-32B` as Agentic Rubric Generator outperforms finetuning them as Patch Classifier for SWE verification.

expert frontier models rubrics are better aligned with ground-truth reference patches, and we analyze this in Appendix A.1. We also report the model's success rate in using the agentic scaffold and producing parseable rubrics in Appendix A.2.

**Takeaway** *Rubric-agent capability matters: stronger frontier models generate more granular rubrics with higher human alignment, which translates into higher* BEST@16 *performance.*

### 5.1.1 Training Open-Weight Rubric Agents

Motivated by this, we study whether such rubric generation capability can be distilled from expert frontier models like `Sonnet-4.5` into smaller open-models. We fine-tune `Qwen3-32B` to *generate agentic rubrics* using the agentic scaffold, and compare against fine-tuning the same model as a *patch classifier* from the same expert model - a common execution-free verifier used in prior works (Jain et al., 2025; Pan et al., 2024).

**Training Setup.** Following Jain et al. (2025), for the **patch classifier**, we fine-tune `Qwen3-32B` to output a `YES/NO` judgment given a problem statement and candidate patch. We sample 2,000 prompts from R2E-Gym and collect 4,696 test-labeled examples (approximately balanced) drawn from both expert (`Sonnet-4.5`) and on-policy (`Qwen3-32B`) rollouts. During verification, we extract the `YES/NO` token probability as the score for the patch. For the **agentic rubric generator**, we fine-tune `Qwen3-32B` use the agentic harness and emit a rubric file, using 2,000 rubric-generation trajectories produced by `Sonnet-4.5` (no on-policy rubric samples). Similar to Jain et al. (2025), we use the AdamW optimization for 2 epochs, a $1.0e-5$ learning rate with cosine scheduling and

| Non-Agentic Rubric | Agentic Rubric | Tool Calls |
|---|---|---|
| *"Targets code paths handling relational operators in the parser without touching unrelated operators"* | *"Adds a* `visit_Compare` *method to* `EvaluateFalseTransformer` *class"* | - `find -path "*/parsing/*"`<br>- `str_replace_editor view`<br>- `view -view_range 1090 1194` |
| *"Modifies the kbd role implementation file that contains HTML generation logic"* | *"Modifies* `KeyboardTransform` *class in* `transforms.py`*"* | - `find -exec grep -l "kbd"`<br>- `grep -r "kbd" transforms.py`<br>- `str_replace_editor view` |

Table 1: Comparison of Non-Agentic vs Agentic Rubrics with the agent's tool calls that gather relevant context

| Judge Model | Best@16 |
|---|---|
| GPT-5-mini | $52.6 \pm 2.20$ |
| GPT-5 Low Reasoning | $54.2 \pm 2.22$ |
| GPT-5 Medium Reasoning | $54.3 \pm 2.25$ |
| GPT-5 High Reasoning | $55.0 \pm 2.21$ |

Table 2: Best@16 accuracy for different judge model capabilities for scoring `Sonnet-4.5` rubrics on `Qwen3-Coder-30B-A3B` rollouts.

a batch size of 32, over 4 nodes of 8xH100 GPUs.

**Results.** Figure 7 shows that the agentic rubric generator substantially outperforms the patch-classifier verifier, and also the non-finetuned base models. This indicates the ability to produce structured, context-grounded rubrics is trainable, and is a stronger and more robust objective than binary classification for execution-free verification. It also demonstrates distilled open-weight models as a pathway to reproducibility and a way to mitigate constant reliance on proprietary APIs

### 5.2 Impact of Repository Grounding

To isolate the value of repository interaction, we compare **Agentic Rubrics** to **Non-Agentic Rubrics** generated by the same model from the problem statement alone, without access to the agentic harness or codebase (prompt in Appendix A.10). As shown in Table 1, agentic rubrics use targeted tool calls (search and file inspection) to ground criteria in concrete repository entities (files, classes, methods), making items more specific and consistently gradable; non-agentic rubrics are often high-level (e.g., "touches the right code path"), which increases ambiguity and can lead to false positives. Empirically, on **SWE-Bench Verified**, using `Sonnet-4.5` for rubric generation without repository access reduces BEST@16 by **4.0** points on `Qwen3-32B` rollouts and **1.4** points on `Qwen3-Coder-30B-A3B` rollouts, showing that agentic context gathering improves both rubric quality and downstream selection.

### 5.3 Sensitivity to Judge Model Choice

In table 2, we analyze how the capability of the judge model affects rubric grading on `Sonnet-4.5` rubrics for `Qwen3-32B` rollouts. We use three increasing reasoning efforts on the `GPT-5` model. We find that judge model capability has a small but non-trivial effect on performance. We don't require high reasoning efforts from our judge models, since rubrics are designed to be self-contained and atomic for easy grading. We also measure the flakiness of rubric grading in A.4.

## 6 Related Work

**Coding Agents and Test Time Scaling** Real world coding for SWE problems have become a popular domain for applying LLMs. SWE environments from open-source GitHub repos provide a good testbed for training (Jain et al., 2025; Jimenez et al., 2023; Pan et al., 2024; Wei et al., 2025a; Luo et al.) and evaluation of coding agents (OpenAI, 2024; Deng et al., 2025). Agentic scaffolds like Agentless, SWEAgent, Mini SWEAgent and OpenHands define a standardized interface for using these models on such tasks (Xia et al., 2024; Yang et al., 2024; SWE-agent Team, 2024; Wang et al., 2024). Test-Time Scaling (TTS) is a way to leverage inference-time compute to improve performance on verifiable tasks like SWE Agents. (Yao et al., 2023; Wang et al., 2023; Brown et al., 2024; Zhu et al., 2025). In addition, some works study the use of verifiers for TTS and RL, but they are limited to training a reward/scoring model like a patch classifier or a testing agent (Pan et al., 2024; Jain et al., 2025; Luo et al.; Wei et al., 2025b).

**Rubrics as verifiers for LLMs** Rubrics have become the predominant way to evaluate LLMs on several key capabilities (Arora et al., 2025; Akyürek et al., 2025). They have also been used as a reward signal to train LLMs during RL (Gunjal et al., 2025; Viswanathan et al., 2025; Goel et al., 2025). In this work, we describe how context-aware rubrics are an effective verifier that can holis-

tically verify candidates for SWE tasks, and demonstrate their value through TTS.

## 7 Conclusion

Automatic, high-quality verification is essential for improving SWE agents. We study **Agentic Rubrics**, a context-grounded yet execution-free verification signal for SWE patches. Under the standard parallel test-time scaling setting on SWE-Bench Verified, Agentic Rubrics consistently outperform strong non-agentic and agentic baselines. Beyond selection performance, rubrics provide interpretable natural-language feedback and are well-aligned with human-written ground-truth tests and reference patches, while also surfacing failure modes that the available tests may not capture. Finally, our ablations study key design choices in the agentic rubric pipeline, including the role of repository interaction, the rubric agent, and the judge model. We hope these findings motivate future work on improving rubric quality and integrating rubrics as reward signals for RL.

## 8 Limitations

Agentic Rubrics provide an interpretable, codebase-grounded verification signal that can be applied *without test execution* once the rubric is produced. We study them in the parallel test-time scaling setting, which offers a clean and widely used way to evaluate verifiers by holding the generator fixed and varying only the selection rule. A natural next step is to use rubric signals in *post-training* pipelines as rewards for RLVR-style optimization. Doing so introduces challenges such as reward hacking, non-stationarity as policies improve, and credit assignment across multi-step agent behavior, which we leave to future work.

Rubric quality is another key axis. While most automatically generated rubric judgments are *high-utility* and substantively grounded, some exhibit low-utility failure modes such as over-specification, redundancy, or rubric–test mismatch. We also evaluate on a curated SWE-Bench Verified setting following prior work, while real-world tasks may be messier or more ambiguous. This motivates *human-in-the-loop* refinement as a practical next step: lightweight review, rubric-template reuse, and targeted prompts for common failure modes could improve rubric fidelity while preserving auditability, and also provide supervision for stronger rubric-generation models.

## References


Afra Feyza Akyürek, Advait Gosai, Chen Bo Calvin Zhang, Vipul Gupta, Jaehwan Jeong, Anisha Gunjal, Tahseen Rabbani, Maria Mazzone, David Randolph, Mohammad Mahmoudi Meymand, Gurshaan Chattha, Paula Rodriguez, Diego Mares, Pavit Singh, Michael Liu, Subodh Chawla, Pete Cline, Lucy Ogaz, Ernesto Hernandez, and 5 others. 2025. PRBench: Large-scale expert rubrics for evaluating high-stakes professional reasoning. *arXiv preprint arXiv:2511.11562*.

Rahul K Arora, Jason Wei, Rebecca Soskin Hicks, Preston Bowman, Joaquin Quiñonero-Candela, Foivos Tsimpourlas, Michael Sharman, Meghan Shah, Andrea Vallone, Alex Beutel, Johannes Heidecke, and Karan Singhal. 2025. Healthbench: Evaluating large language models towards improved human health. *arXiv preprint arXiv:2505.08775*.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *Preprint*, arXiv:2407.21787.

Giuseppe Crupi, Rosalia Tufano, Alejandro Velasco, Antonio Mastropaolo, Denys Poshyvanyk, and Gabriele Bavota. 2025. On the effectiveness of llm-as-a-judge for code generation and summarization. *Preprint*, arXiv:2507.16587.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. 2025. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. Codemonkeys: Scaling test-time compute for software engineering. *Preprint*, arXiv:2501.14723.

Shashwat Goel, Rishi Hazra, Dulhan Jayalath, Timon Willi, Parag Jain, William F. Shen, Ilias Leontiadis, Francesco Barbieri, Yoram Bachrach, Jonas Geiping, and Chenxi Whitehouse. 2025. Training ai co-scientists using rubric rewards. *Preprint*, arXiv:2512.23707.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. 2025. Rubrics as rewards: Reinforcement learning beyond verifiable domains. *arXiv preprint arXiv:2507.17746*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Shang Zhu Tarun Venkat, Ben Athiwaratkun, and 1 others. Deepswe: Training a fully open-sourced, state-of-the-art coding agent by scaling rl.

Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, and 24 others. 2025. 2 olmo 2 furious. *Preprint*, arXiv:2501.00656.

OpenAI. 2024. Introducing swe-bench verified. https://openai.com/index/introducing-swe-bench-verified/. OpenAI Blog.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2024. Training software engineering agents and verifiers with SWE-gym. *arXiv preprint arXiv:2412.21139*.

Rulin Shao, Akari Asai, Shannon Zejiang Shen, Hamish Ivison, Varsha Kishore, Jingming Zhuo, Xinran Zhao, Molly Park, Samuel G Finlayson, David Sontag, and 1 others. 2025. Dr tulu: Reinforcement learning with evolving rubrics for deep research. *arXiv preprint arXiv:2511.19399*.

Nishad Singhi, Hritik Bansal, Arian Hosseini, Aditya Grover, Kai-Wei Chang, Marcus Rohrbach, and Anna Rohrbach. 2025. When to solve, when to verify: Compute-optimal problem solving and generative verification for llm reasoning. *arXiv preprint arXiv:2504.01005*.

SWE-agent Team. 2024. mini-swe-agent: A 100-line software engineering agent. https://github.com/SWE-agent/mini-swe-agent. GitHub repository.

Vijay Viswanathan, Yanchao Sun, Shuang Ma, Xiang Kong, Meng Cao, Graham Neubig, and Tongshuang Wu. 2025. Checklists are better than reward models for aligning language models. *arXiv preprint arXiv:2507.18624*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *Proceedings of the International Conference on Learning Representations (ICLR)*. ArXiv:2203.11171.

Jason Wei. 2025. Asymmetry of verification and verifier's rule. Blog post.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. 2025a. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. 2025b. Toward training superintelligent software agents through self-play swe-rl. *Preprint*, arXiv:2512.18552.

Mian Wu, Gavin Zhang, Sewon Min, Sergey Levine, and Aviral Kumar. 2025. Rlac: Reinforcement learning with adversarial critic for free-form generation tasks. *arXiv preprint arXiv:2511.01758*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *Preprint*, arXiv:2407.01489.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems*. NeurIPS 2023, arXiv:2305.10601.

Kang Zhu, Hanhao Li, Siwei Wu, Tianshun Xing, Dehua Ma, Xiangru Tang, Minghao Liu, Jian Yang, Jiaheng Liu, Yuchen Eleanor Jiang, Changwang Zhang, Chenghua Lin, Jun Wang, Ge Zhang, and Wangchunshu Zhou. 2025. Scaling test-time compute for LLM agents. *arXiv preprint arXiv:2506.12928*.

# A Appendix

## A.1 Analyzing agentic rubric scores against Ground-Truth patch

Table 3: Average Weighted Scores by Agentic Rubrics produced by different models on human written Ground-Truth Patches

| Model | Avg Weighted Score |
|---|---|
| Opus-4.5 | 0.8658 |
| GPT-5 | 0.8413 |
| Sonnet-4.5 | 0.8233 |
| Gemini-3-Pro | 0.8082 |
| Meta-CWM | 0.8015 |
| Qwen3-Coder-30B-A3B | 0.8037 |
| Qwen3-32B | 0.6729 |

In table 3, we show the average scores by rubrics generated by different models over the Ground-Truth patches for the tasks. We also show the breakdown across different rubric axis for a representative subset in 8. Frontier Coding Models have higher alignment with human-written Ground-Truth patches than open-weight models.

## A.2 Agentic abilities of rubric generation models

Figure 9 shows the percentage of instances where each model successfully produces parseable YAML rubric files. Frontier models demonstrate near-perfect adherence to the structured output format: Sonnet-4.5 and Gemini-3-Pro generate well-formatted rubric files 97.8% and 96.8% respectively, with zero parse errors. In contrast, smaller models exhibit degraded format compliance, due to weaker tool calling and instruction following. Qwen3-32B produces valid rubrics for only 74.6% of instances with an 18.2% parse error rate, while Meta-CWM succeeds on 69.4% of instances but fails to generate rubrics entirely for 29.8% of cases.

In figure 10, we show how finetuning Qwen3-32B on Sonnet-4.5 rubric agent trajectories leads to better use of the Agentic harness as demonstrated by reduced errors and improved rubric distribution, matching the original teacher model. This demonstrates that we can train rubric generator models, which unlocks their use for creating preference data and reward models.

## A.3 Cost analysis for agentic verification methods

In table 4, we describe the cost analysis of different agentic methods. Note that the cost of Grading for Tests is spinning up a sandboxed environment for grading and running the test suite after applying the patch. We use Modal to run sandboxed containers. All costs are averaged over the entire dataset. The cost of artifact is once per instance (a fixed cost) while cost of grading per rollout, and will scale with the number of rollouts used. Therefore the average total cost of BEST@16 calculation (Agentic artifact generation + grading 16 rollouts) per instance is $0.736 for Patch Similarity, $0.515 for Test Generation and $0.293 for Rubrics.

| Method | Artifact Cost($) | Grading Cost($) | API Calls | Qwen3-32B Best@16 |
|---|---|---|---|---|
| Patch Sim. | 0.640 | 0.006 | 48.5 | 36.6 |
| Tests | 0.499 | 0.001 | 29.2 | 33.6 |
| Rubrics | 0.245 | 0.003 | 22.9 | 40.6 |

Table 4: Cost vs Performance (Qwen3-32B Best@K) comparison of different agentic verifier methods that all use the same underlying model, Sonnet-4.5. Artifact Cost is the average cost in USD ($) for the total input and output tokens in the entire trajectory. Note that agentic patch generation was run with 50 steps as the limit to improve solution patch quality since the reference patches produced with the 30 turn limit was too restrictive to get any comparable result, while the other methods are limited to 30 turns.

We see that rubric generation is very cost efficient, requiring fewer tool calls and tokens while also achieving higher performance.

## A.4 Rubric Flakiness Study

To assess the reliability of LLM-based rubric evaluation, we do a flakiness study measuring grading determinism across repeated trials. We randomly sampled 20 instances for rubric generator models `Sonnet-4.5`, and `Qwen3-32B`, selecting 5 rubric items per instance (100 items per model). Each rubric item was then scored 5 independent times using `GPT-5 (low reasoning)` as the judge, producing binary assessments of whether the candidate patch satisfied the rubric criterion.

A rubric item is considered "flaky" if any of its 5 trial scores differ from the others. Our results demonstrate high scoring determinism, and stronger models write better, less flaky rubrics: `Sonnet-4.5` generated rubrics exhibited only 2% flakiness (98% of items scored identically across all trials), while `Qwen3-32B` had 9% flakiness. We attribute such low levels of flakiness to our instructions that mandate rubrics to be atomic and self-contained, reducing the scope for the Judge's inter-

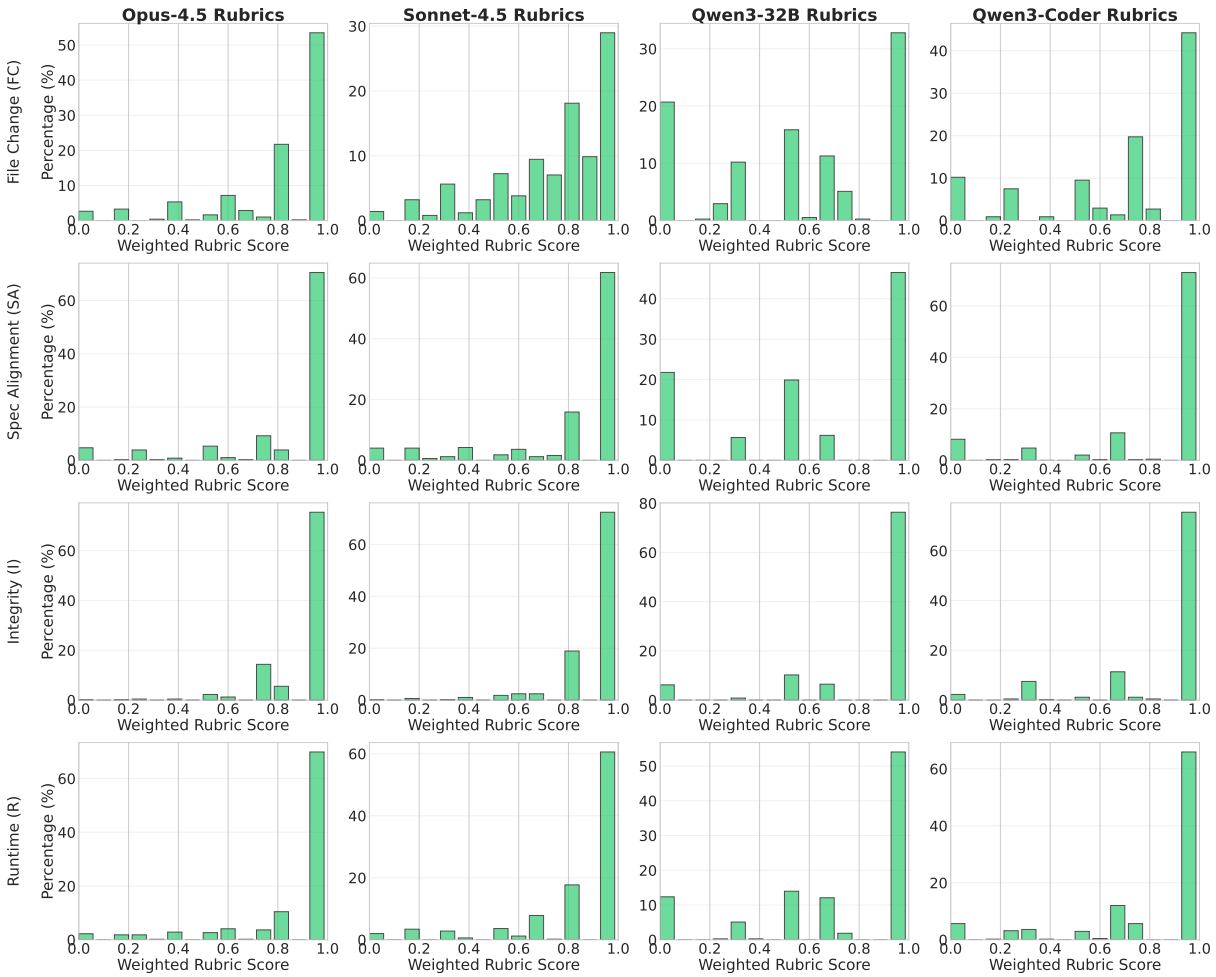



Figure 8: Distribution of rubric scores on reference patches comparing good vs bad models

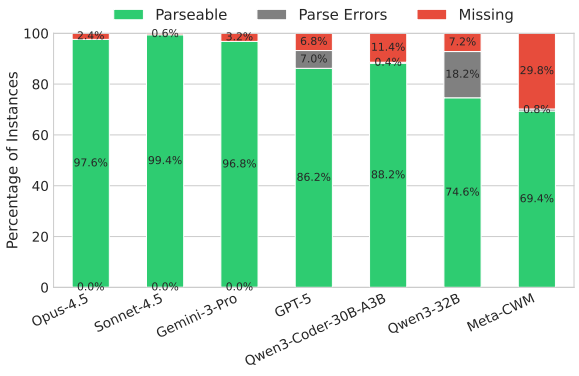


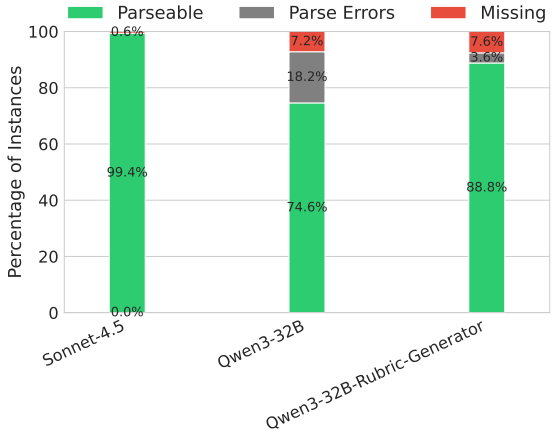

Figure 9: Performance of different models in using the Rubric Generation scaffold to create parseable rubric files.

pretation for grading.

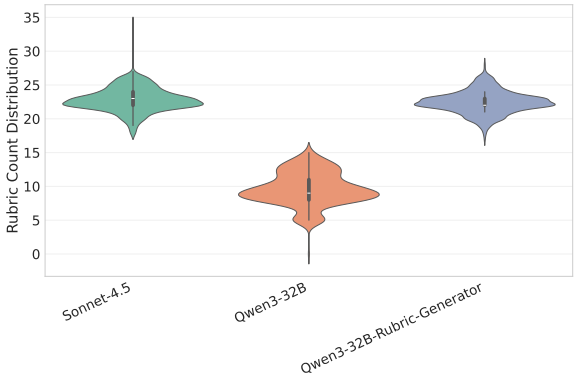
High consistency rates are important, since they lead to reproducible assessments and lower gaming opportunities. Future work can study this further, to establish best practices for writing strong, deterministic yet non-prescriptive rubrics.

### A.5 Hybrid verifiers using rubrics v/s classifier

We also study how agentic rubrics compare against a classifier based approach when combined with generated tests in a hybrid approach similar to R2E-Gym (Jain et al., 2025). In figure 11, we see that a hybrid verifier setup where we combine agentic tests and rubrics in a simple aggregation setup can work better than both methods in isolation. This opens up future work in combining these verifiers in more complex pipelines to extract maximum utility from different verfication methods.




(a) Agentic harness use improvement through finetuning




(b) Rubric structure improvement through finetuning

Figure 10: Finetuning Qwen3-32B on Sonnet-4.5 rubric agent trajectories leads to (a) better use of the Agentic harness as demonstrated by reduced errors and (b) improved rubric distribution, matching the original model.

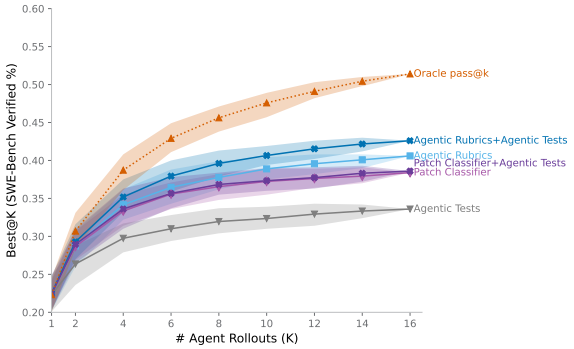



Figure 11: Combining Agentic Rubrics with Agentic Tests, to build a Hybrid Verifier.

### A.6 Categories of rubric utility classification

In table 5, we describe the taxonomy used to label rubric utility. This was crafted manually by inspecting the rollouts and failure modes with different rubrics.

### A.7 SWE Agent Setup

We use the SWE-Agent scaffold (Yang et al., 2024) for all agentic setups. We add a turn reminder every 5 turns about the number of turns left for the agent before autosubmission. In addition, we wanted to study and distill rubric generation capability but many state-of-the-art models completely skip any thinking or assistant tokens and just returns the tool call with the default SWE-Agent setup. So we change the parsing function to explicitly mandate that in each turn, it return a short summary of its thinking. Prompts for all methods can be found in A.10.

### A.8 Rubric and their grading - Illustrative Examples

In A.9 we show three representative examples in which the patches pass all the Ground-Truth tests for the problem but still score low on rubrics. In particular, in the first example of `matplotlib__matplotlib-26291`, users report that creating an inset axis and saving a figure with `bbox_inches="tight"` crashes with an `AttributeError`. The root cause is that Matplotlib calls a locator object with `renderer = None`, while downstream code assumes a valid renderer.

The *candidate patch* from Qwen3-32B guards against this by returning a dummy zero-size bounding box whenever the renderer is missing or an error occurs. This is enough to satisfy the ground-truth tests.

However, the *rubrics* encode a stronger notion of correctness. They require that the fix (1) preserves the positioning and sizing of inset axes (R4), (2) works for both absolute and relative size specifications(SA4), and (3) still works in the original failure mode with `bbox_inches='tight'` (SA5), without breaking existing user code (R3). These checks treat the inset axis as a real visual element whose layout must remain meaningful. The candidate's dummy box violates this implicit contract: it causes tight layout to ignore the inset entirely. The user no longer sees a crash, but also no longer gets a correctly placed inset in the "tight" saved figure.

## A.9 Rubric Examples and grading

### matplotlib__matplotlib-26291

**Problem.** Creating inset axes and saving with `bbox_inches='tight'` can raise `AttributeError: 'NoneType' object has no attribute '_get_renderer'` because the inset locator is called with `renderer=None` and attempts to access the figure renderer through a None figure.

**Candidate patch (core diff)**

```
--- a/lib/mpl_toolkits/axes_grid1/inset_locator.py
+++ b/lib/mpl_toolkits/axes_grid1/inset_locator.py
@@ -70,7 +70,18 @@ class AnchoredLocatorBase(AnchoredOffsetbox):

     def __call__(self, ax, renderer):
         self.axes = ax
-        bbox = self.get_window_extent(renderer)
+        if renderer is None:
+            # No renderer available, return a dummy bounding box of zero size
+            # to avoid errors during tight layout.
+            dummy_bbox = Bbox.from_bounds(0, 0, 0, 0)
+            tr = ax.figure.transSubfigure.inverted()
+            return TransformedBbox(dummy_bbox, tr)
+        try:
+            bbox = self.get_window_extent(renderer)
+        except (AttributeError, RuntimeError) as e:
+            # Handle cases where the renderer is not available, returning
+            # a dummy bbox again.
+            dummy_bbox = Bbox.from_bounds(0, 0, 0, 0)
+            tr = ax.figure.transSubfigure.inverted()
+            return TransformedBbox(dummy_bbox, tr)
         px, py = self.get_offset(bbox.width, bbox.height, 0, 0, renderer)
         bbox_canvas = Bbox.from_bounds(px, py, bbox.width, bbox.height)
         tr = ax.figure.transSubfigure.inverted()
         return TransformedBbox(bbox_canvas, tr)
```

**Rubrics (FAILED / PASSED for candidate patch)**

- FC1: Modifies the `AnchoredLocatorBase.__call__` method in `lib/mpl_toolkits/axes_grid1/inset_locator.py` to set `self.figure` before calling `get_window_extent`.
- FC2: Ensures the fix is localized to the `__call__` method in `AnchoredLocatorBase` without modifying unrelated methods or classes.
- FC3: Sets `self.figure` from `ax.figure` in `AnchoredLocatorBase.__call__` before the `get_window_extent` call.
- FC4: Avoids modifying the parent class `AnchoredOffsetbox` in `lib/matplotlib/offsetbox.py` or its `get_window_extent` method.
- FC5: Does not introduce changes to other locator classes like `AnchoredSizeLocator` or `AnchoredZoomLocator` beyond what is inherited from `AnchoredLocatorBase`.
- FC6: Avoids adding unnecessary imports or dependencies to `inset_locator.py`.

- I1: Does not weaken or skip existing tests in `lib/mpl_toolkits/axes_grid1/tests/test_axes_grid1.py` that verify `inset_axes` functionality.
- I2: Maintains the public API signature of `inset_axes` without adding or removing parameters.
- I3: Preserves the existing behavior of `AnchoredLocatorBase.__call__` returning a `TransformedBbox` as before.
- I4: Does not modify the constructor `__init__` of `AnchoredLocatorBase` or its parent class initialization.
- I5: Avoids introducing deprecation warnings or changing the inheritance hierarchy of `AnchoredLocatorBase`.

- R1: Ensures `self.figure` is set to a non-None `Figure` instance before `get_window_extent` accesses `self.figure._get_renderer()`.
- R2: Allows `inset_axes` creation to proceed through the full render path, including `tight_bbox.adjust_bbox`, without `AttributeError`.
- R3: Maintains backward compatibility so existing code using `inset_axes` continues to work without modification.
- R4: Preserves correct positioning and sizing of inset axes relative to the parent axes after the fix.
- R5: Ensures the fix handles both cases where `renderer` is None and where `renderer` is explicitly passed to `get_window_extent`.
- R6: Allows the locator to be called multiple times on the same axes without error or state corruption.

- SA1: Resolves the `AttributeError: 'NoneType' object has no attribute '_get_renderer'` that occurs when calling `inset_axes` with the code example from the bug report.
- SA2: Allows `get_window_extent` to successfully access `self.figure._get_renderer()` when `renderer` is None in the `offsetbox.py` implementation.
- SA3: Preserves the existing behavior where `self.axes` is set to the `ax` parameter in `AnchoredLocatorBase.__call__`.
- SA4: Maintains compatibility with both absolute size specifications like `width=1.3` and relative specifications like `width='30%'` as shown in the `inset_locator_demo.py` examples.
- SA5: Ensures the fix works when `bbox_inches='tight'` is used during figure saving, which triggers the `adjust_bbox` path that calls the locator.

### matplotlib__matplotlib-25332

**Problem.** After calling `fig.align_labels()` or `fig.align_ylabels()`, `pickle.dumps(fig)` fails with `TypeError: cannot pickle 'weakref.ReferenceType' object` because `Grouper` stores weak references that are not directly pickle-safe.

**Candidate patch (full diff)**

```
diff --git a/lib/matplotlib/cbook.py b/lib/matplotlib/cbook.py
index c9699b2e21..1adf054843 100644
--- a/lib/matplotlib/cbook.py
+++ b/lib/matplotlib/cbook.py
@@ -752,6 +752,24 @@ def print_cycles(objects, outstream=sys.stdout, show_progress=False):


 class Grouper:
+    def __getstate__(self):
```

```
+        # Convert weakrefs to strong references for pickling
+        clean_mapping = {}
+        for key, value in self._mapping.items():
+            # key is a weakref, value is a list of weakrefs
+            strong_key = key()
+            strong_values = [ref() for ref in value]
+            clean_mapping[strong_key] = strong_values
+        return clean_mapping
+
+    def __setstate__(self, clean_mapping):
+        # Reconstruct the _mapping with weakrefs from strong references
+        self._mapping = {}
+        for key, value in clean_mapping.items():
+            # key is a strong reference, value is a list of strong references
+            weak_key = weakref.ref(key)
+            weak_values = [weakref.ref(val) for val in value]
+            self._mapping[weak_key] = weak_values
     """"""
     A disjoint-set data structure.
```

**Rubrics (FAILED / PASSED for candidate patch)**

- FC1: Modifies `lib/matplotlib/figure.py` to handle `_align_label_groups` during pickling in `__getstate__` or `__setstate__`.
- FC2: Avoids modifying unrelated files beyond `lib/matplotlib/figure.py`, `lib/matplotlib/cbook.py`, and `lib/matplotlib/tests/test_pickle.py`.
- FC3: Adds or modifies pickle support methods (`__getstate__`, `__setstate__`, `__reduce__`, or `__reduce_ex__`) specifically to handle `Grouper` weak references.
- FC4: Preserves existing pickling logic in `Figure.__getstate__` for canvas, `_dpi`, `__mpl_version__`, and `_restore_to_pylab`.
- FC5: Implements the solution either by modifying `cbook.Grouper` class to add pickle support or by handling `_align_label_groups` specially in `Figure`'s pickle methods.
- FC6: Avoids adding unnecessary complexity such as deep copying the entire figure state or serializing canvas data that should remain transient.

- I1: Adds a test case in `lib/matplotlib/tests/test_pickle.py` that reproduces the exact scenario from the bug description with `align_labels()` call.
- I2: Avoids weakening or skipping existing pickle tests in `test_pickle.py`.
- I3: Maintains backward compatibility by not changing the public API of `Figure.align_labels`, `Figure.align_xlabels`, or `Figure.align_ylabels`.
- I4: Does not break existing functionality where figures without `align_labels()` can still be pickled successfully.
- I5: Preserves the `Grouper` API if modifying `cbook.Grouper` class, keeping `join()`, `joined()`, `get_siblings()`, `clean()`, and `__iter__()` methods functional.
- I6: Avoids removing or commenting out the `_align_label_groups` attribute initialization in `Figure.__init__`.

- R1: Maintains correct label alignment behavior where labels in the same row or column remain visually aligned after pickle round-trip.
- R2: Ensures the unpickled figure's `_align_label_groups` contains valid `Grouper` objects or equivalent structures that preserve axis grouping relationships.
- R3: Prevents `AttributeError` or `KeyError` when accessing `_align_label_groups` after unpickling the figure.
- R4: Ensures that calling `align_labels()` again on an unpickled figure does not raise exceptions and properly updates groupings.
- R5: Preserves memory efficiency by not creating strong references to axes objects where weak references were intentionally used.
- R6: Handles edge cases where axes are deleted or garbage collected between pickling and unpickling without causing crashes.

- SA1: Ensures `pickle.dumps(fig)` succeeds after calling `fig.align_labels()` without raising `TypeError` about `weakref.ReferenceType`.
- SA2: Ensures `pickle.dumps(fig)` succeeds after calling `fig.align_xlabels()` independently.
- SA3: Ensures `pickle.dumps(fig)` succeeds after calling `fig.align_ylabels()` independently.
- SA4: Preserves label alignment behavior after unpickling so that aligned labels remain properly grouped.
- SA5: Allows round-trip pickling where `pickle.loads(pickle.dumps(fig))` produces a functional figure with preserved alignment state.

## django__django-13417

**Problem.** For models with `Meta.ordering`, annotated querysets that introduce `GROUP BY` clauses (e.g., via `annotate(Count('pk'))`) still report `qs.ordered` as `True` even though the generated SQL no longer includes an `ORDER BY`. This makes the `ordered` property inconsistent with the actual ordering behavior.

**Candidate patch (core diff)**

```
diff --git a/django/db/models/sql/query.py b/django/db/models/sql/query.py
index 4648daf395..e1672b0c1c 100644
--- a/django/db/models/sql/query.py
+++ b/django/db/models/sql/query.py
@@ -1030,6 +1030,7 @@ class Query(BaseExpression):
         else:
             self.set_annotation_mask(set(self.annotation_select).difference({alias}))
         self.annotations[alias] = annotation
+        self.clear_ordering(True)

     def resolve_expression(self, query, *args, **kwargs):
         clone = self.clone()
@@ -1038,10 +1039,13 @@ class Query(BaseExpression):
         clone.subquery = True
         # It's safe to drop ordering if the queryset isn't using slicing,
         # distinct(*fields) or select_for_update().
+        # Clear ordering if annotation requires grouping
+        self.clear_ordering(True)
         if (self.low_mark == 0 and self.high_mark is None and
                 not self.distinct_fields and
                 not self.select_for_update):
```

```
            clone.clear_ordering(True)
+       self.clear_ordering(True)
        clone.where.resolve_expression(query, *args, **kwargs)
        for key, value in clone.annotations.items():
            resolved = value.resolve_expression(query, *args, **kwargs)
```

**Rubrics (FAILED / PASSED for candidate patch)**

- FC1: Modifies the `ordered` property in `django/db/models/query.py` to check if `query.group_by` is set and if ordering comes only from `Meta.ordering`.
- FC2: Avoids modifying `django/db/models/sql/compiler.py` lines 590–591 where `order_by` is set to `None` when `_meta_ordering` is `True` and grouping exists.
- FC3: Preserves the existing logic in `QuerySet.ordered` for checking `self.query.extra_order_by`, `self.query.order_by`, and `self.query.default_ordering`.
- FC4: Adds a check in `QuerySet.ordered` that returns `False` when `self.query.group_by` is not `None` and ordering only comes from `self.query.get_meta().ordering`.
- FC5: Does not modify the `Query.default_ordering` attribute or `Query.set_group_by` method in `django/db/models/sql/query.py`.
- FC6: Limits changes to the `ordered` property method between lines 1218–1230 in `django/db/models/query.py` without altering other `QuerySet` methods.

- I1: Avoids weakening existing tests by adding `pytest.mark.skip` or removing assertions from test files.
- I2: Does not modify the public API of `QuerySet` class beyond fixing the `ordered` property behavior.
- I3: Maintains the `EmptyQuerySet` check in `ordered` that returns `True` for empty querysets.
- I4: Preserves `query.default_ordering` attribute semantics without changing its value based on `GROUP BY` presence.
- I5: Does not introduce new `QuerySet` methods or properties beyond fixing the existing `ordered` property.

- R1: Ensures `ordered` returns `False` when query execution would produce SQL without `ORDER BY` clause due to `GROUP BY` with `Meta.ordering`.
- R2: Maintains `ordered` as deterministic based on query state without database round-trips or SQL compilation.
- R3: Preserves the `ordered` contract for `first` and `last` methods in `django/db/models/query.py`.
- R4: Ensures `ordered` checks `query.group_by` state consistently whether it is `None`, `True`, or a tuple of expressions.
- R5: Avoids race conditions by checking immutable query attributes like `group_by` and `default_ordering` without modifying them.
- R6: Returns correct `ordered` value for chained queryset operations like `annotate` followed by `filter` without requiring SQL generation.

- SA1: Ensures `QuerySet.ordered` returns `False` when `annotate` with `Count` or similar aggregation creates a `GROUP BY` clause on models with `Meta.ordering`.
- SA2: Maintains `QuerySet.ordered` returning `True` when no `GROUP BY` is present even if the model has `Meta.ordering`.
- SA3: Ensures `QuerySet.ordered` still returns `True` when explicit `order_by` is called even with `GROUP BY` present.
- SA4: Aligns `ordered` behavior with actual SQL generation where `compiler.py` lines 590–591 clear `order_by` when grouping and `self._meta_ordering` are both present.
- SA5: Preserves backward compatibility for `QuerySet.ordered` when `query.group_by` is `None`.

## A.10 Prompts - Baselines and Agentic Rubrics

### Agentic Patch Similarity (Rollout Generation)

```
agent:
  templates:
    system_template: |-
      You are a helpful assistant that can interact with a computer to solve tasks.
    instance_template: |-
      <uploaded_files>
      {{working_dir}}
      </uploaded_files>
      I've uploaded a python code repository in the directory {{working_dir}}. Consider the following PR description:

      <pr_description>
      {{problem_statement}}
      </pr_description>

      Can you help me implement the necessary changes to the repository so that the requirements specified in the <
      pr_description> are met?
      I've already taken care of all changes to any of the test files described in the <pr_description>. This means you
      DON'T have to modify the testing logic or any of the tests in any way!
      Your task is to make the minimal changes to non-tests files in the {{working_dir}} directory to ensure the <
      pr_description> is satisfied.
      Follow these steps to resolve the issue:
      1. As a first step, it might be a good idea to find and read code relevant to the <pr_description>
      2. Create a script to reproduce the error and execute it with `python <filename.py>` using the bash tool, to
      confirm the error
      3. Edit the sourcecode of the repo to resolve the issue
      4. Rerun your reproduce script and confirm that the error is fixed!
      5. Think about edgecases and make sure your fix handles them as well
      Your thinking should be thorough and so it's fine if it's very long.
    next_step_template: |-
      OBSERVATION:
      {{observation}}
    next_step_no_output_template: |-
      Your command ran successfully and did not produce any output.
  tools:
    env_variables:
      PAGER: cat
      MANPAGER: cat
```

```
      LESS: -R
      PIP_PROGRESS_BAR: 'off'
      TQDM_DISABLE: '1'
      GIT_PAGER: cat
    bundles:
      - path: tools/registry
      - path: tools/edit_anthropic
      - path: tools/review_on_submit_m
    registry_variables:
      USE_FILEMAP: 'true'
      SUBMIT_REVIEW_MESSAGES:
        - |
          Thank you for your work on this issue. Please carefully follow the steps below to help review your changes.

          1. If you made any changes to your code after running the reproduction script, please run the reproduction
      script again.
             If the reproduction script is failing, please revisit your changes and make sure they are correct.
             If you have already removed your reproduction script, please ignore this step.
          2. Remove your reproduction script (if you haven't done so already).
          3. If you have modified any TEST files, please revert them to the state they had before you started fixing the
      issue.
             You can do this with `git checkout -- /path/to/test/file.py`. Use below <diff> to find the files you need to
      revert.
          4. Run the submit command again to confirm.

          Here is a list of all of your changes:

          <diff>
          {{diff}}
          </diff>
    enable_bash_tool: true
    disable_image_processing: true
    parse_function:
      type: function_calling
  history_processors:
    - type: cache_control
      last_n_messages: 2
  model:
    temperature: 1.
    retry:
      retries: 3
```

## Agentic Patch Similarity (Judge)

```
REFERENCE_BASELINE_SYSTEM_PROMPT = """You are an expert code reviewer evaluating AI-generated patches for software
      engineering tasks. Your task is to compare a CANDIDATE PATCH against a REFERENCE PATCH (golden answer) and rate the
       candidate on a scale of 1-5.

Scoring Criteria:
- 5 (Excellent): The candidate patch is functionally equivalent to the reference. It correctly addresses the problem,
      handles edge cases, and follows best practices.
- 4 (Good): The candidate patch addresses the core problem correctly but may have minor differences in implementation
      approach or style compared to the reference.
- 3 (Acceptable): The candidate patch partially addresses the problem. It may miss some edge cases or have incomplete
      fixes, but demonstrates understanding of the issue.
- 2 (Poor): The candidate patch shows an attempt to fix the problem but has significant issues. It may introduce bugs,
      miss the root cause, or have incorrect logic.
- 1 (Incorrect): The candidate patch fails to address the problem, is completely wrong, or makes changes unrelated to the
       issue.

Return your evaluation as JSON with a score and brief reasoning.

JSON format:
{
  "score": <integer 1-5>,
  "reasoning": "<brief explanation>"
}
"""


REFERENCE_BASELINE_USER_PROMPT = """Please evaluate the CANDIDATE PATCH by comparing it to the REFERENCE PATCH (golden
      answer).

[PROBLEM DESCRIPTION]
{problem_statement}

[REFERENCE PATCH (Golden Answer)]
{reference_patch}

[CANDIDATE PATCH (To Evaluate)]
{candidate_patch}
```

```
Please evaluate the CANDIDATE PATCH and return the score in JSON format only.

EVALUATION:"""
```

## Agentic Rubrics

```
agent:
  templates:
    system_template: |-
      You are an expert code reviewer that can understand issues and are well versed in the codebase. Your job is to
      write high-quality rubrics to grade the solution to a given issue.

      IMPORTANT: In EVERY turn, you MUST ALWAYS include:
      1. A summary of your thinking - explain what you're planning to do and why, and what tool you're going to use in
      (4-5 sentences max).
      2. A tool call. You can only make one tool call per turn.
    instance_template: |-
      <uploaded_files>
      {{working_dir}}
      </uploaded_files>
     I've uploaded a python code repository in the directory {{working_dir}}. Consider the following PR description:

      <pr_description>
      {{problem_statement}}
      </pr_description>

      Can you help me write high quality rubrics to grade the solution to the task described in the <pr_description>?
      This means you SHOULDN'T attempt to solve the task yourself, but rather understand the task, go through the
      codebase, and write rubrics only.
      Follow these steps to write the rubrics:
      1. As a first step, it might be a good idea to find and read code relevant to the <pr_description> by searching the
       codebase using search tools.
      2. Then think of the approach to solve the task (functional requirements, non-functional requirements, etc.)
      3. Also understand how the codebase is structured and how the code is organized, as well as coding style, etc.
      4. Write a list of rubrics that can be used to grade the solution to the task described in the <pr_description>.
      Your rubrics should be along the axes described below.
      5. Then finally, make a new file in the {{working_dir}} directory called `rubrics.yaml` with the rubrics you wrote.
       It should be a valid YAML file with the structure described later below
      6. Also take a turn to ensure the yaml file is parseable by the yaml.safe_load function on the file itself. If it
      throws an error, fix the yaml file and take another turn to ensure it is parseable.
      7. This should be the only file you create in the {{working_dir}} directory. DO NOT create/modify/delete any other
      files or directories.
      8. Finally, submit the task.

      Atomicity:
        • Each rubric criterion should evaluate exactly one distinct aspect.
        • Avoid bundling multiple criteria into a single rubric. Most stacked criteria with the word "and" can be broken
      up into multiple pieces.

      Self-containment & specificity (strict):
        • Do NOT write generic items; bind each item to exact paths/symbols/tokens seen in this instance.
        • Never rely on cross-item references; each item stands alone with its own identifiers and patterns.
        • The judge will only have access to the problem and patch and the current rubric under evaluation, so make sure
      the rubric can be evaluated without any other information.

      Mutually Exclusive, Collectively Exhaustive (MECE):
        • The rubric set should be mutually exclusive and collectively exhaustive.

      Style constraints (strict):
        • YAML only-no prose outside YAML.
        • Avoid backslash-heavy patterns; if you absolutely must include one, double any backslashes so the YAML stays
      valid.
        • Each rubric description starts with a third-person singular verb (e.g., Identifies, Implements, Validates,
      Confirms, Avoids, Cleans up, Plans).
        • Make descriptions concrete using tokens from PATCH/PR_DESCRIPTION.
        • Each rubric item includes: id (short), description (verb-first, instance-grounded), weight (int; 1=nice, 2=
      valuable, 3=must).
        • Avoid double-counting: do not re-score the same behavior under multiple items.

      Pattern-writing guidelines (keep literal YAML-friendly text; no regex required):
        • Use plain path mentions like "diff --git a/path/to/file.py" or "+++ b/path/to/file.py".
        • Refer to symbols with straightforward phrases such as "def my_function(" instead of regex classes.
        • Describe value patterns in words (e.g., "string containing total") instead of complex expressions.
        • If you need to forbid something, just mention the exact string '@pytest.mark.skip', etc

      Axes (execution-free):
        • file_change_rubrics (4 - 8): Scope, locality, and sufficiency of edits in PATCH (files/symbols/guards/regexes/
      flags). Penalize unrelated file churn; reward minimal, reversible diffs tied to the stated bug.
        • spec_alignment_rubrics (3 - 6): Alignment of code to PR_DESCRIPTION. Use textual acceptance criteria (required
      types/conditions/error handling/API contracts) and ensure the patch reflects them.
        • integrity_rubrics (3 - 6): Hygiene and "no-cheating" safeguards-avoid test weakening (if tests appear in PATCH)
      , mass renames, or dependency churn; preserve public API/semantics unless PR_DESCRIPTION requires otherwise.
```

```
      • runtime_rubrics (3 – 6): Natural-language criteria describing **intended runtime behavior** (NOT concrete tests
    ), supported by execution-free textual evidence.
          - **Distinguishability:** Ensures the patch introduces or preserves signals that differentiate correct vs.
    incorrect behavior under plausible inputs (e.g., specific exception class, sentinel return, boundary guard).
          - **Regression safety:** Confirms backward-compatibility constraints (e.g., original API signatures/flags
    remain valid, deprecations gated via warnings).
          - **Determinism / flake resistance:** Avoids nondeterministic sources at runtime (unseeded randomness, wall-
    clock sleeps, network I/O) that would make tests flaky.
          - **Resource & timeout bounds:** Prevents pathological loops or heavy calls; respects existing timeouts/
    limits.
          - **Error-surface clarity:** Produces stable, specific messages/exception types that a test could assert
    against (not vague strings).
          - **Harness integrity:** Does not bypass or disable the project's runner/verifier hooks (e.g., keeps
    regression filters, CLI exit codes).

    Return exactly this YAML structure:
    metadata:
      task_summary: "<one-sentence summary grounded in PR_DESCRIPTION>"
      underlying_bug: "<precise failure trigger grounded in PR_DESCRIPTION>"
    axes:
      file_change_rubrics:
        - id: "FC1"
          description: "Identifies ..."
          weight: 3
        - id: "FC2"
          description: "Identifies ..."
          weight: 2
      spec_alignment_rubrics:
        - id: "SA1"
          description: "Recognizes ..."
          weight: 2
      integrity_rubrics:
        - id: "I1"
          description: "Confirms ..."
          weight: 2
      runtime_rubrics:
        - id: "R1"
          description: "Maintains ..."
          weight: 2

  next_step_template: |-
    OBSERVATION:
    {{observation}}
  next_step_no_output_template: |-
    Your command ran successfully and did not produce any output.
tools:
  env_variables:
    PAGER: cat
    MANPAGER: cat
    LESS: -R
    PIP_PROGRESS_BAR: 'off'
    TQDM_DISABLE: '1'
    GIT_PAGER: cat
  bundles:
    - path: tools/registry
    - path: tools/edit_anthropic
    - path: tools/review_on_submit_m
  registry_variables:
    USE_FILEMAP: 'true'
    SUBMIT_REVIEW_MESSAGES:
      - |
        Thank you for your work on writing the rubrics. Please carefully follow the steps below to help review your
    changes.

        1. If you made any changes to your code other than the `rubrics.yaml` file in the testbed directory, please
    revert them to the state they had before you started writing the rubrics.
        2. You can do this with `git checkout -- /path/to/file.py`. Use below <diff> to find the files you need to
    revert. This has to be done, otherwise we can't extract the rubrics.yaml file.          3. Run the submit command
    again to confirm.

        Here is a list of all of your changes:

        <diff>
        {{diff}}
        </diff>
  enable_bash_tool: true
  disable_image_processing: true
  parse_function:
    type: function_calling
history_processors:
  - type: cache_control
    last_n_messages: 2
model:
  temperature: 1.0
  retry:
```

```
      retries: 3
```

## Agentic Tests

```
agent:
  templates:
    system_template: |-
      You are a programming agent who is provided a github issue and repository bash environment and is tasked to
      generate a standalone test script that can reproduce and verify the issue without relying on any testing frameworks
      .
    instance_template: |-
      <uploaded_files>
      {{working_dir}}
      </uploaded_files>
      I've uploaded a python code repository in the directory {{working_dir}}. Consider the following PR description:

      <pr_description>
      {{problem_statement}}
      </pr_description>

      Can you help me write a standalone test_issue.py file that tests and reproduces the issue described in the <
      pr_description>?
      This test file should be completely self-contained and executable directly with Python, without requiring any
      testing frameworks like pytest or unittest.

      IMPORTANT GUIDELINES:
      1. First, explore the repository to understand what the issue is about and how to test and reproduce it. Focus on
      understanding the core functionality rather than the testing structure.

      2. Create a standalone Python script (test_issue.py) that:
        - Imports only the necessary modules from the repository
        - Sets up the minimum environment needed to reproduce the issue
        - Contains all logic within the script itself (no external test dependencies)
        - Runs quickly and terminates itself (no background servers or long-running processes)
        - Write at least ten test cases to test the issue.

      3. CRITICAL: For each of the test cases: your test script MUST use these EXACT print statements to indicate test
      results for each test case:
        - Print "FAILED" when the code confirms the bug exists, and so the test case fails.
        - Print "PASSED" when the code runs without the issue and so the test case passes.
        - Print "Other issues" when unexpected problems occur
        IMPORTANT: Again include the above print statements for each of the test cases in /testbed/test_issue.py.

      4. Include error handling to prevent the script from crashing:
        - Catch exceptions appropriately
        - Always output one of the three exact phrases above
        - DO NOT use assertions that might terminate the program (without error handling)

      5. The test should fail (print "FAILED") when run against the current repo state.

      6. Your test script should also check for the correct behaviour when the issue is fixed (i.e. print "PASSED"). If
      the issue is not fixed and the code exhibits incorrect behavior after applying a fix, it should print "Other issues
      " or "FAILED" as appropriate.

      7. Write the final test script to /testbed/test_issue.py. Ensure that the script is runnable via `python test_issue
      .py`.

      8. The final line in the test script should be a comment about how many test cases were written. Example: `# Total
      tests: 6`.

      9. This is important each test case number (Out of the total you wrote), we will parse the output of test script
      one-by-one and check if the solution passes test case i. For example, """test_case_{i} PASSED""" or """test_case_{i
      } FAILED""".

      Example format for a single test case in the test script:
      ```python
      import sys
      from some_package import relevant_module

      def test1():
          try:
              # Setup minimal test environment
              test_input = "example input that triggers the issue"

              # Attempt the operation that should reproduce the issue
              result = relevant_module.function_with_issue(test_input)

              # check if the issue is reproduced
              if result == "expected output that indicates the issue":
                  return "FAILED"
              else:
                  # check if result matches the expected output when the issue is resolved
```
```

```
                # ensure to perform all necessary checks
                assert result == "expected output when resolved"
                return "PASSED"

            except Exception as e:
                return "Other issues"  # Optional: can include error details for debugging

    ...

    if __name__ == "__main__":
        print(f"test_case_1 {test1()}")
        ...

    # Total tests: 5
    ```

    FINAL CHECKS:
    - Does each one of your test run standalone (without pytest/unittest)?
    - Does each one of your test contain EXACTLY ONE of the three required print statements?
    - Does each one of your test terminate automatically after printing the result?
    - Does each one of your test properly reproduce the issue described in the problem statement?
    - Is it simple, focused, and free of unnecessary complexity?
    - Does the final line in the test script contain the correct number of test cases and with the exact format `#
    Total tests: <number of test cases>` (no commas, no spaces, no other text)?

    GENERAL INSTRUCTIONS:
    - Each response must include both
      - natural language reasoning about your approach
      - a function call to solve the task
    - You can take multiple turns to solve the task, but only finish once you're confident in your solution
    - If a file_editor edit fails, view the file before retrying with adjusted content

    General Steps:
    1. Understand the issue, corresponding code and how to reproduce the issue.
    2. Write a standalone test script that reproduces the issue. Make sure that the output is "FAILED" for each of the
    single test.
    3. Add further test cases including more thorough testing, inputs, edge cases to ensure the issue is correctly
    identified.
    4. Run the test script to ensure output is as expected (see example output format below).

    The final output of the test script should resemble the following format (just an example):
    <EXAMPLE OUTPUT FORMAT>
    test_case_1 FAILED
    test_case_2 PASSED
    test_case_3 FAILED
    test_case_4 PASSED
    test_case_5 FAILED
    test_case_6 PASSED
    test_case_7 FAILED
    test_case_8 PASSED
    test_case_9 FAILED
    </EXAMPLE OUTPUT FORMAT>
    You must follow the above format for the output of the test script. Other issues should be max 1-2 test cases (in
    worst case).

    Finally, use submit tool to submit.

    CRITICAL: Do not submit until you have added diverse test cases and thoroughly verified the output of the test
    script.
    NOTE: for django environments: you should use test_sqlite settings file during testing.

  next_step_template: |-
    OBSERVATION:
    {{observation}}
  next_step_no_output_template: |-
    Your command ran successfully and did not produce any output.
tools:
  env_variables:
    PAGER: cat
    MANPAGER: cat
    LESS: -R
    PIP_PROGRESS_BAR: 'off'
    TQDM_DISABLE: '1'
    GIT_PAGER: cat
  bundles:
    - path: tools/registry
    - path: tools/edit_anthropic
    - path: tools/review_on_submit_m
  registry_variables:
    USE_FILEMAP: 'true'
    SUBMIT_REVIEW_MESSAGES:
      - |
        Thank you for your work on writing the tests. Please carefully follow the steps below to help review your
    changes.
```
```

```
        1. If you made any changes to your code other than the `test_issue.py` file, please revert them to the state
    they had before you started writing the tests.
        2. You can do this with `git checkout -- /path/to/file.py`. Use below <diff> to find the files you need to
    revert.
        3. Run the submit command again to confirm.

        Here is a list of all of your changes:

        <diff>
        {{diff}}
        </diff>
  enable_bash_tool: true
  disable_image_processing: true
  parse_function:
    type: function_calling
history_processors:
  - type: cache_control
    last_n_messages: 2
model:
  temperature: 1.
  retry:
    retries: 3
```

## Patch Classifier

```
SYSTEM_PROMPT = """You are an expert judge evaluating AI assistant interactions. Your task is to determine if the
    assistant successfully resolved the user's request.

Key evaluation criteria:
1. Did the assistant complete the main task requested by the user?
2. Did the assistant handle all edge cases and requirements specified?
3. Were there any errors or issues in the final solution?

Respond only with "<judgement>YES</judgement>" or "<judgement>NO</judgement>" based on
if the assistant successfully resolved the user's request."""

USER_PROMPT_TEMPLATE = """Please evaluate the following request to solve a coding issue and the proposed solution:

[PROMPT]

{problem_statement}

[SOLUTION]

{model_patch}"""
```

## Non-Agentic Rubrics

```
SYSTEM_PROMPT = """You are an expert code reviewer that can understand issues and are well versed in codebases. Your job
    is to write high-quality rubrics to grade the solution to a given issue.

Based on the problem description provided, write rubrics that can be used to evaluate a patch that attempts to solve the
    issue.

Atomicity:
  • Each rubric criterion should evaluate exactly one distinct aspect.
  • Avoid bundling multiple criteria into a single rubric. Most stacked criteria with the word "and" can be broken up
    into multiple pieces.

Self-containment & specificity (strict):
  • Do NOT write generic items; bind each item to exact paths/symbols/tokens mentioned in the problem description.
  • Never rely on cross-item references; each item stands alone with its own identifiers and patterns.
  • The judge will only have access to the problem and patch and the current rubric under evaluation, so make sure the
    rubric can be evaluated without any other information.

Mutually Exclusive, Collectively Exhaustive (MECE):
  • The rubric set should be mutually exclusive and collectively exhaustive.

Style constraints (strict):
  • YAML only-no prose outside YAML.
  • Avoid backslash-heavy patterns; if you absolutely must include one, double any backslashes so the YAML stays valid.
  • Each rubric description starts with a third-person singular verb (e.g., Identifies, Implements, Validates, Confirms,
    Avoids, Cleans up, Plans).
  • Make descriptions concrete using tokens from the problem description.
  • Each rubric item includes: id (short), description (verb-first, instance-grounded), weight (int; 1=nice, 2=valuable,
    3=must).
  • Avoid double-counting: do not re-score the same behavior under multiple items.

Pattern-writing guidelines (keep literal YAML-friendly text; no regex required):
```

```
  • Use plain path mentions like "diff --git a/path/to/file.py" or "+++ b/path/to/file.py".
  • Refer to symbols with straightforward phrases such as "def my_function(" instead of regex classes.
  • Describe value patterns in words (e.g., "string containing total") instead of complex expressions.
  • If you need to forbid something, just mention the exact string '@pytest.mark.skip', etc

Axes (execution-free):
  • file_change_rubrics (4-8): Scope, locality, and sufficiency of edits in PATCH (files/symbols/guards/regexes/flags).
      Penalize unrelated file churn; reward minimal, reversible diffs tied to the stated bug.
  • spec_alignment_rubrics (3-6): Alignment of code to the problem description. Use textual acceptance criteria (required
       types/conditions/error handling/API contracts) and ensure the patch reflects them.
  • integrity_rubrics (3-6): Hygiene and "no-cheating" safeguards-avoid test weakening (if tests appear in PATCH), mass
      renames, or dependency churn; preserve public API/semantics unless the problem description requires otherwise.
  • runtime_rubrics (3-6): Natural-language criteria describing **intended runtime behavior** (NOT concrete tests),
      supported by execution-free textual evidence.
      - **Distinguishability:** Ensures the patch introduces or preserves signals that differentiate correct vs.
      incorrect behavior under plausible inputs (e.g., specific exception class, sentinel return, boundary guard).
      - **Regression safety:** Confirms backward-compatibility constraints (e.g., original API signatures/flags remain
      valid, deprecations gated via warnings).
      - **Determinism / flake resistance:** Avoids nondeterministic sources at runtime (unseeded randomness, wall-clock
      sleeps, network I/O) that would make tests flaky.
      - **Resource & timeout bounds:** Prevents pathological loops or heavy calls; respects existing timeouts/limits.
      - **Error-surface clarity:** Produces stable, specific messages/exception types that a test could assert against (
      not vague strings).
      - **Harness integrity:** Does not bypass or disable the project's runner/verifier hooks (e.g., keeps regression
      filters, CLI exit codes).

Return exactly this YAML structure (and nothing else):
metadata:
  task_summary: "<one-sentence summary grounded in the problem description>"
  underlying_bug: "<precise failure trigger grounded in the problem description>"
axes:
  file_change_rubrics:
    - id: "FC1"
      description: "Identifies ..."
      weight: 3
    - id: "FC2"
      description: "Identifies ..."
      weight: 2
  spec_alignment_rubrics:
    - id: "SA1"
      description: "Recognizes ..."
      weight: 2
  integrity_rubrics:
    - id: "I1"
      description: "Confirms ..."
      weight: 2
  runtime_rubrics:
    - id: "R1"
      description: "Maintains ..."
      weight: 2
"""

USER_PROMPT_TEMPLATE = """Consider the following problem description:

<problem_description>
{problem_statement}
</problem_description>

Write high quality rubrics to grade a patch that attempts to solve the task described in the <problem_description>.
Output ONLY valid YAML with the structure specified in the system prompt. Do not include any other text or explanation.
"""
```

## A.11 Rubric Judge Prompt

**Rubric Judge Prompt**

```
SYSTEM_PROMPT = """
You are a rubric based evaluator for software-engineering agent's generated patch. Use the provided rubric to evaluate
      the generated patch.

Inputs (provided later):
        - PR_DESCRIPTION: problem + expected behavior.
        - RUBRIC: dictionary of n rubric items for an ideal patch with their ids as keys and descriptions as values.
  - PATCH: the model's predicted code patch

Your job:
        1.      Analyze the rubric and the patch to evaluate the SWE-agent's generated patch.
        2.      Emit a score for each rubric item. The score should be a binary score of 1 if the patch satisfies the
      rubric item and 0 otherwise.

Return the scores in a JSON format.
```

```
JSON format:
{
  "<rubric_id_1>": <score_1>,
  "<rubric_id_2>": <score_2>,
  "<rubric_id_3>": <score_3>,
  "<rubric_id_4>": <score_4>,
  ...
  "<rubric_id_n>": <score_n>
}"""


USER_PROMPT = """
PR_DESCRIPTION:
{pr_description}

PATCH:
{patch}

RUBRIC:
{rubric}

Please evaluate the PATCH using the rubric and return the scores in JSON format.
SCORES:
"""
```

## A.12 Rubric Utility Analysis Prompt

**Rubric Utility Analysis Prompt**

```
SYSTEM_PROMPT = """
You are an expert software engineer. Your job is to analyze how a candidate patch
is graded by tests and rubrics.

Inputs (provided later):
  - problem_statement
  - golden_patch (the ground-truth code patch)
  - candidate_patch (the model's predicted code patch)
  - golden_test_cases
  - test_case_reward (0 or 1)
  - rubric_descriptions (map: rubric_id -> text)
  - rubrics_to_analyze (list of rubric ids to analyze)

Treat problem_statement + golden_patch + golden_test_cases as the ground-truth
specification for correct behavior.

We focus on HIGH-ALIGNMENT cases between tests and rubrics:
  - If test_case_reward = 0, rubrics_to_analyze is the set of FAILING rubrics.
    Here we ask: how well do these failing rubrics align with the reasons the
    golden tests reject the candidate_patch?
  - If test_case_reward = 1, rubrics_to_analyze is the set of ACCEPTED rubrics.
    Here we ask: how well do these accepted rubrics align with the reasons the
    golden tests accept the candidate_patch?

Your task:
  1. For each rubric in rubrics_to_analyze, decide whether it is:
     a) Valid   - its judgment is consistent with the ground-truth spec and it
                  provides a correct, meaningful reason that agrees with the
                  test outcome.
     b) Spurious - its judgment is not well supported by the ground-truth spec
                  or adds noise (e.g., unnecessary constraints, conflicts with
                  the golden behavior, or misinterprets the situation), even
                  though the sign of the score aligns with the tests.
  2. Assign each rubric to exactly ONE sub-category under its main category
     (Valid or Spurious). If no sub-category fits, create a new one with a short
     title and a brief description.

VALID sub-categories (use when the rubric adds real value and is consistent
with the spec):
  - Core Semantics
      Checks whether the patch actually fixes or preserves the functional
      behavior described in the problem (root cause, outputs, semantics).
  - Edge Coverage
      Enforces handling of important edge cases or reproducer scenarios that
      are implied by the spec but not fully covered by tests.
  - API / Compat
      Ensures public API shape, types, and behavior stay compatible with
      existing callers/versions, or change exactly as required by the spec.
  - Structure / Scope
      Ensures the change is in the correct module/layer and reasonably
      localized (no wrong-layer fix, no unrelated edits, no scope creep).
  - Performance Risk
      Flags likely performance or resource regressions that tests do not
```

```
      directly measure.
  - Security / Safety
      Enforces validation, security, or safety properties that should not be
      weakened by the patch.
  - [NEW VALID]
      If none fit, create a new Valid sub-category with a short title and a
      1-2 sentence description.

SPURIOUS sub-categories (use when the patch is acceptable given the spec, but
the rubric's failure is not well justified):
  - Low-Signal
      Encodes irrelevant or redundant constraints (style-only, hygiene-only,
      or no new information beyond other checks).
  - Over-Specified
      Demands a specific implementation even though alternative correct fixes
      are allowed.
  - Spec Conflict
      Contradicts the problem statement, golden_patch, or golden_test_cases
      (forbids behavior the spec allows or disagrees with the reference).
  - Test Mismatch
      Assumes a different test setup or evaluation protocol than the one
      actually used (e.g., expects test edits that are not in scope).
  - Eval Error
      Failure is caused by a scoring / matching / parsing bug rather than a
      real property of the candidate_patch.
  - [NEW SPURIOUS]
      If none fit, create a new Spurious sub-category with a short title and a
      1-2 sentence description.

Reasoning rules:
  - Always treat golden_patch + golden_test_cases as the authoritative spec.
  - If a rubric's explanation clearly conflicts with this spec, it is almost
    certainly Spurious.
  - Mark a rubric as Valid only if its judgment and rationale are consistent
    with the spec AND meaningfully explain why the test outcome (0 or 1) is
    correct.

Output format:
Return a JSON array, one entry per rubric in rubrics_to_analyze:

[
  {
    "rubric_id": "FC5",
    "rubric_description": "<from rubric_descriptions>",
    "tier_category": "Valid" or "Spurious",
    "subcategory_title": "<one of the titles above or a new one>",
    "subcategory_description": "<1-2 sentence definition>",
    "justification": "<2-4 sentences citing candidate_patch, golden_patch,
                      golden_test_cases, and referring to test_case_reward>"
  }
]
"""

USER_PROMPT = """
problem_statement:
{problem_statement}

golden_patch:
{golden_patch}

candidate_patch:
{candidate_patch}

golden_test_cases:
{golden_test_cases}

test_case_reward:
{test_case_reward}

rubric_descriptions:
{rubric_descriptions}

rubrics_to_analyze:
{rubrics_to_analyze}

Please classify each rubric in rubrics_to_analyze as Valid or Spurious,
assign a sub-category, and return the JSON array as specified above.

RESULT:
"""
```

| Tier | Sub-category | Description |
|---|---|---|
| High-Utility | Rule Break | Violation of an internal contract or assumption that should hold even if tests do not exercise it directly. |
| High-Utility | Band-Aid Fix | Patch makes tests pass but does not actually fix the underlying semantics or root cause of the bug. |
| High-Utility | Wrong Layer | Fix is implemented in the wrong module, class, or layer; the behavior change should live elsewhere in the codebase. |
| High-Utility | Root Cause Missed | Core bug remains present or is only partially addressed; the patch does not truly resolve the reported issue. |
| High-Utility | Missing Edges | Important edge cases or reproducer scenarios implied by the spec are not handled, even though the main path may pass tests. |
| High-Utility | Scope Creep | Patch introduces unrelated or off-scope changes (e.g., extra files, debug code, refactors, binary blobs). |
| High-Utility | Perf Risk | Patch is likely to introduce a non-trivial performance or resource-usage regression that tests do not directly measure. |
| High-Utility | API Break | Patch breaks a public API or its backward-compatible behavior as relied on by existing callers or downstream code. |
| High-Utility | Security Risk | Patch weakens validation, safety, or security properties (e.g., injection, data exposure) beyond what tests explicitly guard. |
| Low-Utility | Test Rules Mismatch | Rubric assumes a different test setup or evaluation protocol than the one actually used (e.g., expects modifying tests or harness behavior). |
| Low-Utility | Over-Specified Fix | Rubric demands a particular implementation strategy or pattern even though multiple correct fixes are allowed by the spec. |
| Low-Utility | API Over-Strict | Rubric penalizes benign API or structural changes even when observable behavior matches the ground-truth spec. |
| Low-Utility | Style Nit | Rubric focuses on purely stylistic or cosmetic issues with no semantic impact on correctness or behavior. |
| Low-Utility | Spec Clash | Rubric contradicts the problem statement, golden patch, or golden test cases (e.g., forbids behavior that the reference explicitly permits). |
| Low-Utility | Ref Patch Conflict | Golden patch itself violates the rubric's stated constraint, indicating that the rubric is misaligned with the reference solution. |
| Low-Utility | Redundant Signal | Rubric adds no new information beyond other rubrics, effectively double-counting the same issue without additional insight. |
| Low-Utility | Eval Bug | Rubric failure arises from a scoring, matching, or parsing bug rather than an actual violation of the rubric text by the candidate patch. |
| Low-Utility | Irrelevant Rule | Rubric encodes a constraint that is not actually needed or justified for this particular bug or problem context. |

Table 5: Rubric category taxonomy used in utility analysis.