See, Read, Compare: Candidate-Aware Verification for Agent Test-Time Scaling

Xinyu Ye[1], **Yongliang Wu**[3], **Xingyu Zhu**[4], **Peng Xia**[1], **Huaijin Wu**[2], **Yehui Tang**[5], **Rui Ye**[2], **Hao Xiong**[6], **Junchi Yan**[2], **Huaxiu Yao**[1*]

[1]University of North Carolina at Chapel Hill [2]Shanghai Jiao Tong University
[3]University of Illinois Urbana-Champaign [4]National University of Singapore
[5]Renmin University of China [6]Fudan University

Abstract

Test-time scaling (TTS) improves LLM agent performance by sampling K independent rollouts and selecting one with a verifier. In open-ended software engineering (SWE) agent settings, verification typically relies on code execution to obtain a direct correctness signal, but incurs substantial setup and testing cost. By contrast, execution-free verifiers offer a more scalable alternative that chooses from the submission alone. However, existing execution-free verifiers are largely submission-isolated: they fix rubrics before seeing the candidate batch, underuse the trajectory evidence already produced during agent rollouts, and rely on poorly calibrated pointwise scores for the final decision. As a result, they often miss the right candidate even when it lies within the pool. To address this gap, we propose a training-free and execution-free verifier for agent test-time scaling based on a *See*, *Read*, *Compare* (SRC) protocol, which treats the K candidates as mutual context rather than independent inputs. The **See** stage combines task understanding with cross-candidate contrast to induce an instance-specific rubric. The **Read** stage extracts structured behavioral features, and compresses each trajectory into an evidence-grounded summary along several cognitive dimensions: localization, hypotheses, interventions, and validation. The **Compare** stage performs rubric-guided pairwise selection through a two-phase hybrid tournament, preserving relative judgment with only linear complexity calls. Experiments show that SRC outperforms all compared execution-free baselines across SWE-bench Verified generator pools and generalizes across model backbones, to Terminal-Bench 2, and to feature implementation tasks on DeepSWE. The source code will be available at https://github.com/XinyuYee/see-read-compare.

1 Introduction

Large language model (LLM) agents that interact with code repositories, terminal environments, and web interfaces have become increasingly effective for software-engineering and systems tasks [1; 2; 3]. Recent work has highlighted test-time scaling (TTS) as a promising approach for improving the performance and reliability of such agents [4; 5]: sample K independent rollouts, then select the best one with a verifier [6; 7; 8; 9]. With a strong verifier, compute-optimal TTS can allow smaller models to match or even surpass larger models [6; 8; 10], making verification a central bottleneck for scaling agent performance at test time [11; 12].

In the SWE setting, execution-based verification is often impractical: hidden evaluators cannot be queried, re-execution is expensive at scale, and many deployment pipelines expose only the trajectory

*Corresponding author.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

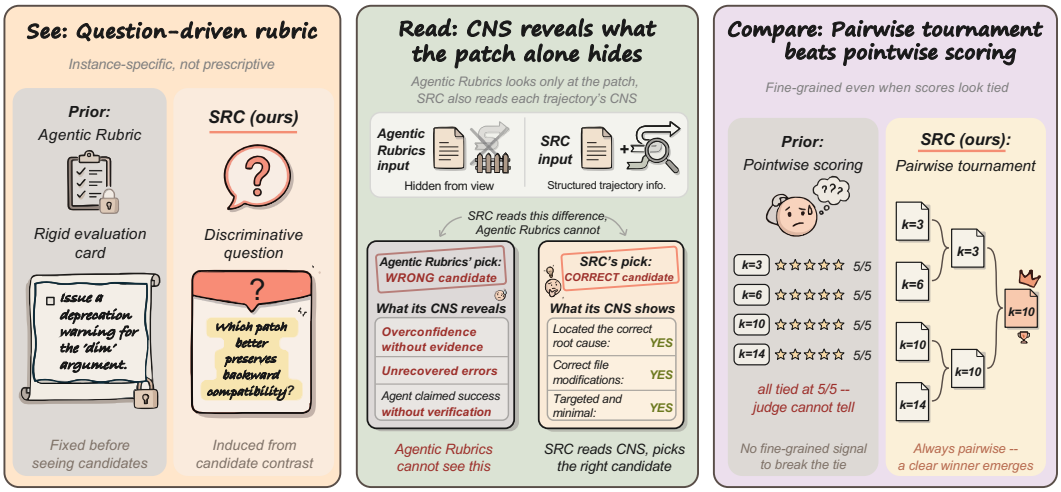


Figure 1: Three illustrative cases where the SRC verifier outperforms other rubric-based and execution-free verifiers. See Appendices C for case studies of these three patterns.

and final patch. We therefore focus on execution-free verifiers that select directly from submitted candidates. Among execution-free strategies, consistency-based selectors [13] pick the patch closest to the candidate majority but lack semantic grounding. Trained verifiers learn correctness signals from labeled trajectories or rollouts but require expensive supervision and often do not transfer across repositories or generator models [14; 15; 16]. Direct LLM judges score each candidate using the model's own implicit evaluation criteria, leaving the decision rationale opaque and vulnerable [17; 18; 19]. By contrast, rubric-based verification is attractive because it makes the LLM judge's evaluation criteria explicit: the LLM first generates a structured rubric for the task and then assesses each candidate against the rubric's dimensions [20]. Recently, Raghavendra et al. [21] proposed *Agentic Rubrics* for SWE tasks, where a rubric agent explores the repository through tool calls and synthesizes a per-instance rubric grounded in the codebase, and a separate LLM judge then uses that rubric to pointwise score each candidate patch. This decomposes patch quality into inspectable, interpretable axes, while still requiring no training and no code execution at scoring time.

However, existing rubric-based and execution-free verifiers fall short in systematically exploiting the candidate batch, leaving three sources of signal underused, as shown in Figure 1. **(1) Rigid, candidate-blind rubrics.** Rubrics are usually constructed before any submitted candidate is observed, so the criteria can over-specify one imagined solution path while missing the discriminative dimensions on which the actual K candidates disagree. **(2) Underused trajectory evidence.** Candidates are judged mainly from their final outputs. However, the agent trajectory can reveal localization attempts, hypotheses, interventions, error recovery, and validation status that a patch-only verifier cannot observe. **(3) Absolutely pointwise scoring.** Pointwise scores are often poorly calibrated across candidates and do not reliably surface the best one [18]. Relative comparison is a more natural judgment for an LLM judge, but full pairwise comparison requires high complexity.

To address this gap, we propose SRC (SEE, READ, COMPARE), a training-free and execution-free verifier built on a core design principle: the candidate batch is shared context that should inform rubric induction, evidence representation, and ranking. SEE synthesizes an instance-specific rubric from a diverse subset of submitted patches, targeting the dimensions on which the candidates actually differ. READ compresses each rollout into a Cognitive Narrative Summary (CNS) that records what the agent actually did, edited, tested, and verified. As Figure 1 (middle) shows, the CNS catches the patch picked by Agentic Rubrics as making confident success claims the agent never actually verified, while flagging the patch picked by SRC as one that found the true root cause and applied a minimal, targeted fix. COMPARE replaces absolute scoring with rubric-guided pairwise comparison aggregated through a two-phase hybrid Swiss tournament with linear complexity, which elects a single winner even on instances where pointwise scoring assigns the same score to four candidates at once. SRC selects the best candidate by leveraging both what each candidate submits and the process by which it produces that output.

In summary, our contributions are threefold:

- We frame execution-free agent TTS verification as *candidate-aware review*, where the candidate batch serves as shared context for rubric induction, trajectory-evidence representation, and relative ranking, rather than a set of independent objects to be scored in isolation.
- We introduce SRC, a three-stage training-free verifier: SEE induces instance-specific rubrics from cross-candidate disagreement; READ distills each agent trajectory into an evidence-grounded Cognitive Narrative Summary; and COMPARE performs rubric-guided pairwise selection via a hybrid Swiss tournament with linear complexity.
- We validate SRC on SWE-bench Verified across multiple generator pools. With Qwen3-Coder rollouts, SRC reaches 57.8% Best@16; with Kimi K2.5 rollouts, it reaches 73.2% Best@16. Component ablations show that the candidate-aware rubric, trajectory-evidence channel, and pairwise aggregation each contribute independently. Additional studies demonstrate SRC generalization across model backbones, feature implementation on DeepSWE, and the cross-domain dataset Terminal-Bench 2.

2 Related Work

Test-time Scaling for LLM Agents. Recent advances in LLMs have explored multiple directions for improving model capability, computational efficiency, and reliability [22; 23; 24; 25; 26]. Test-time scaling (TTS) improves task performance by spending additional inference compute on sampling, search, or selection guided by a verifier [6; 7; 9; 27; 11]. The paradigm is especially relevant for LLM agents, whose long-horizon decision making benefits from longer trajectories, parallel rollouts, and structured search over candidate solutions [4; 5]. A canonical scaling strategy is Best-of-K selection, where K independent rollouts are sampled and a verifier picks one. Self-Consistency [13] is the simplest such selector, choosing by majority agreement among candidate outputs. The verifier is the operator that turns more rollouts into more correct answers: a strong verifier converts additional samples into higher task success.

Verifiers for Software Engineering Agents. Software engineering grounds correctness in passing test suites, making verification well-defined yet operationally costly. SWE benchmarks [28; 29] and developer-style scaffolds [1; 2; 3] have made SWE a central testbed. Execution-based verifiers obtain strong signals by re-running each candidate in a sandboxed environment, but they incur substantial setup, compute, and time cost [3; 14; 30]. Trained execution-free verifiers reduce this cost by learning outcome or process signals from labeled trajectories or patches [15; 14; 16; 31]. Process reward models give finer-grained, step-level signals but typically require step-level supervision [32; 33; 34; 35]. A complementary line studies trajectory selection and multi-verifier aggregation for Best-of-N [36; 37], while structured semantic analysis of patches offers verification signals without running code [38].

Rubric-Based Verification. Rubrics decompose quality into inspectable dimensions and have served as a training-free, execution-free interface for evaluating open-ended outputs in reward modeling, reinforcement learning, and synthetic supervision [20; 39; 40; 41]. In SWE agent verification, the closest method is Agentic Rubrics [21], which constructs a per-instance rubric through agentic repository exploration and then scores each candidate against it. The resulting rubric is contextual to the task and the codebase, but the criteria are still committed before any submitted candidate is observed and can over-specify one imagined solution path that the actual submissions diverge from. SRC relocates the source of the rubric to the submitted candidate batch itself: SEE performs contrastive analysis on a diverse subset of submitted patches and induces an instance-specific rubric whose dimensions are exactly those on which the candidates disagree, while READ grounds each candidate in trajectory evidence and COMPARE resolves the ranking through pairwise judgment.

Comparison with Existing Verifiers. Table 1 compares SRC with five families of verifiers. The first three columns show what a verifier needs beyond the candidates, "Batch rubric" shows whether the criteria are built from the submitted candidates, "Trajectory" shows what the judge receives from the trajectory, and "Selection" shows how the final candidate is chosen. The test-based component of R2E-Gym [14] runs code in the repository. R2E-Gym trains the agent that writes these tests, and its full method adds a trained trajectory verifier. Trained verifiers such as SWE-RM [16] need no execution at scoring time, but they need labeled data and score each candidate on its own. Process

Table 1: Comparison of SRC with representative verifier families.

Family	Example	No training	No execution	No repo access	Batch rubric	Trajectory	Selection
Execution-based	R2E-Gym (EB) [14]	×	×	×	×	None	Test scores
Trained verifier	SWE-RM [16]	×	✓	✓	×	Raw	Pointwise
Process reward	PRM [34]	×	✓	✓	×	Raw steps	Pointwise
Rubric	Agentic Rubrics [21]	✓	✓	×	×	None	Pointwise
Pairwise	PairJudge [42]	×	✓	✓	×	Raw solution	Knockout
Ours	SRC	✓	✓	✓	✓	Evidence summary	Hybrid tournament

reward models [34] need step-level labels and multiply the step scores into one score per candidate. Agentic Rubrics [21] needs no training and runs no code at scoring time, but an agent with shell access explores the repository and writes one rubric per task without seeing any candidate. PairJudge [42] compares pairs of full math solutions in a knockout tournament, but it needs training. SRC is the only row that needs no training, no execution, and no repository access. It also builds the rubric from the candidate batch, gives the judge an evidence summary of each trajectory, and selects with a hybrid tournament that uses a linear number of judge calls.

3 Method

3.1 Problem formulation

Let t denote a software engineering task with problem statement $\mathbf{I}(t)$, a GitHub issue paired with the relevant repository checkout. An agent runs on t and produces K independently sampled candidates $\mathcal{T} = \{\tau^{(1)}, \ldots, \tau^{(K)}\}$. Each candidate τ records the agent's full rollout and exposes two pieces of evidence to the verifier: **1)** Interaction *trajectory* inside τ: the sequence of reasoning steps, tool calls, and environment observations the agent produced while solving t. **2)** The final *patch* $\rho(\tau)$: the agent's final answer, extracted from the end of the rollout. An execution-free verifier selects the best candidate $\tau^\star$ from the pool using only the candidates, without executing the patch.

Figure 2 shows the architecture of SRC. The candidate pool on the left holds K rollouts, each carrying a code submission (patch) and an agent trajectory. The patches and trajectories feed SEE and READ stage. SEE treats disagreement among the candidate patches as the discriminative signal it needs, and induces an instance-specific rubric. READ thoroughly mines each trajectory and distills it into an evidence-grounded summary the verifier can read alongside the patch. COMPARE selects the final candidate through rubric-guided pairwise review and hybrid tournament aggregation.

3.2 SEE: Candidate-Aware Rubric Induction

SEE addresses a central limitation of prior rubric-based verification [21]: a rubric written before seeing the submitted candidate pool can miss the distinctions that actually matter for selection, and a rubric built around one explored solution path can reward implementation-specific details that are not required for correctness. SRC instead treats disagreement among candidates as a signal. By deriving criteria from observed candidate diversity, SEE induces an instance-specific rubric that is less tied to one explored solution path.

Diverse Subset Selection. SEE first selects a small subset $\mathcal{S} \subset \{1, \ldots, K\}$ with $|\mathcal{S}| = n$ that exposes the main disagreements among candidates. We use greedy farthest-point sampling over final-output distance. The seed is the candidate with the longest patch, which typically carries the most content and serves as a stable anchor. We then add one candidate at a time, picking the one whose minimum distance to the current subset is largest:

$$k^\star = \arg\max_{k \notin \mathcal{S}} \min_{j \in \mathcal{S}} d\big(\rho(\tau^{(k)}), \rho(\tau^{(j)})\big). \tag{1}$$

d is the normalized edit distance between the two final-output strings. We repeat (1) until $|\mathcal{S}| = n$. Diversity sampling biases the subset toward candidates that expose the largest discriminative differences. The resulting subset gives the model a compact view of the most visibly different submissions, which is what the contrastive rubric induction in the next step needs.

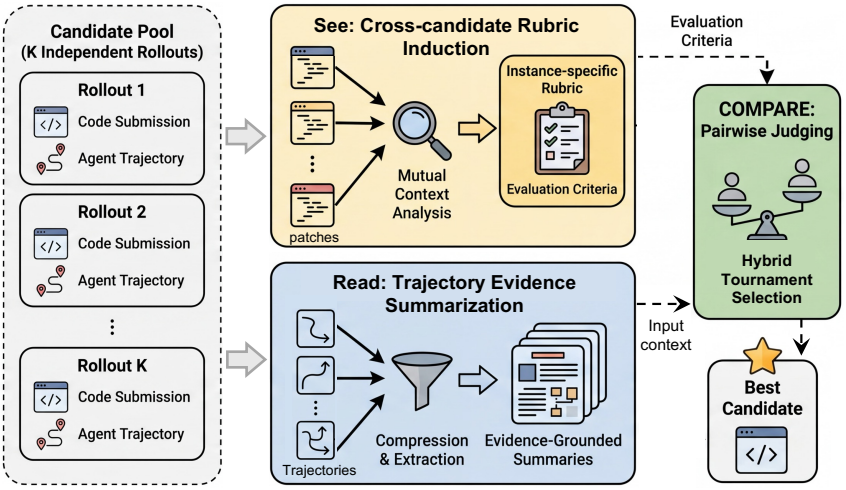


Figure 2: Overall Architecture of SRC. SEE (top) induces an instance-specific rubric from a diverse subset of patches, and READ (bottom) compresses each rollout into a summary grounded in trajectory evidence. COMPARE then selects the best candidate through a two-phase hybrid Swiss tournament.

Rubric Generation via Contrastive Analysis. The rubric generated model receives the task instruction $\mathbf{I}(t)$ and selected final outputs $\{\rho(\tau^{(k)})\}_{k\in\mathcal{S}}$, and then it is prompted to perform structured contrastive reasoning in five steps: **1) Identify the root cause**: extract the underlying bug or unexpected behavior from the problem statement. **2) Survey the patch landscape**: note each selected candidate's approach, the files it touches, and its likely flaws. **3) Extract divergence points**: pinpoint where the candidates disagree on what to do. **4) Mark the common ground**: list what all candidates share. This common ground is explicitly excluded from the rubric because it cannot discriminate between candidates. **5) Enumerate failure modes**: predict the likely ways a patch can be wrong, implied by the divergence pattern. The output is a JSON-formatted rubric with m dimensions:

$$\mathcal{R} \;=\; \{r_1, \ldots, r_m\}, \qquad r_i = (q_i,\, \mathbf{e}_i), \tag{2}$$

where each q_i is a discriminative question phrased for pairwise comparison, and $\mathbf{e}_i$ is a short list of 2 to 4 concrete indicators grounded in entities observed in the selected candidates. For SWE-bench we use $m = 4$ dimensions, named *root cause*, *fix correctness*, *change scope*, and *edge cases*. The full prompt template appears in Appendix B.1.

3.3 READ: Trajectory Evidence Summarization

Execution-free verifiers often judge a candidate from its final output alone, leaving the trajectory evidence underused. This is risky in agent TTS: an agent may submit a passing-looking patch while its trajectory shows it never verified the relevant test. The trajectory itself carries signals that the patch hides, including which files were localized, which hypotheses the agent considered and abandoned, how it recovered from errors, and whether it ran any verification. The challenge is that raw trajectories are long, noisy, and mixed with self-reports that often overstate progress [14]. READ therefore distills each trajectory into two complementary evidence channels: a deterministic *behavioral profile* and a compact *Cognitive Narrative Summary* (CNS) whose organization is inspired by cognitive accounts of reasoning [43] but grounded in observed agent behavior. The final trajectory summarization is passed alongside the patch to COMPARE stage.

Extracting Behavioral Features. We parse the trajectory into N turns $\tau = (u_1, \ldots, u_N)$, where each turn $u_j = (\mathit{thought}_j, \mathit{action}_j, \mathit{observation}_j)$ records the agent's reasoning, command, and environment response. Deterministic regex-based classifiers annotate each turn along four axes: action type, action outcome, cognitive phase, and hypothesis evolution. The per-turn annotations are aggregated into a single trajectory-level vector $b(\tau) \in \mathbb{R}^d$, covering total turns, phase completeness, explore-to-exploit ratio, error and recovery rates, and counts of edits, tests, and hypothesis changes. Every component is a deterministic function of τ and $\rho(\tau)$, so $b(\tau)$ adds no LLM cost.

Cognitive Narrative Summary Construction. Given the task instruction, candidate, and behavioral structural features, an LLM summarizer produces:

$$\mathrm{CNS}(\tau) = f^{\mathrm{sum}}_{\textsc{Read}}\big(\mathbf{I}(t), \tau, \rho(\tau), b(\tau)\big), \tag{3}$$

a structured text with five dimensions: (1) a *fault hypothesis chain* listing each hypothesis with its supporting evidence and final status, (2) a *causal model* linking root cause to mechanism to symptom, (3) an *intervention analysis* recording the files changed and whether the change is targeted and minimal, (4) a *verification status* flag noting whether tests and edge cases were actually run, and (5) *cognitive red flags* such as overconfidence, unrecovered errors, phase skipping, or unverified success claims. The prompt instructs the summarizer to ground every statement in observable facts from τ rather than the self-reported claims of the agent, because agents routinely report success when the patch is wrong. The structural-features block from $b(\tau)$ acts as a fact sheet that the summarizer cross-checks against the narrative produced by the agent. The full template and prompt appear in Appendix B.2 (Table 7).

3.4 COMPARE: Rubric-Guided Pairwise Selection

After SEE defines what to compare and READ prepares trajectory summary information, the verifier's remaining job is to pick one candidate from the K available. Pointwise scores are known to be arbitrary and inconsistent across candidates on hard problems [42; 12], while round-robin pairwise comparison would require $\binom{K}{2}$ judge calls and is wasteful when only the top candidate is needed. COMPARE runs a rubric-guided pairwise comparator inside a two-phase *hybrid Swiss* tournament that keeps the reliability of pairwise judgments while reducing the comparison budget to $O(K)$.

Pairwise judge call. For any candidate pair (i, j), the judge J receives the task instruction $\mathbf{I}(t)$, the rubric $\mathcal{R}$ from SEE, and each candidate's final patch $\rho(\tau^{(i)})$, $\rho(\tau^{(j)})$ paired with its $\mathrm{CNS}(\tau^{(i)})$, $\mathrm{CNS}(\tau^{(j)})$ from READ. A single LLM call returns a structured verdict

$$v_{ij} \;=\; \big(o_{ij},\, c_{ij}\big), \tag{4}$$

where $o_{ij} \in \{A, B, \mathrm{TIE}\}$ is the overall winner and $c_{ij} \in \{\text{high}, \text{medium}, \text{low}\}$ is the judge's confidence. The confidence is mapped to a continuous weight $w(\text{high}) = 1.0$, $w(\text{medium}) = 0.75$, $w(\text{low}) = 0.5$, which scales the score update in the tournament below.

Hybrid Tournament Aggregation. Pure knockout uses only $K-1$ calls, but a single early judge error can eliminate a strong candidate. A Swiss-system tournament restores robustness by giving every candidate multiple matches, but at the cost of $O(K \log K)$ pairwise calls, most of which are spent comparing low-ranked candidates that cannot be the top-1 pick. We propose a two-phase aggregator (Algorithm 1) that retains knockout's linear budget while restoring Swiss-style robustness on the small set of candidates that can plausibly be the argmax. Phase 1 runs knockout to surface a champion W and records $\mathcal{P}_W$, the set of candidates defeated along W's bracket path. Phase 2 runs a small Swiss tournament on $\mathcal{F} = \{W\} \cup \mathcal{P}_W$, the candidates most exposed to possible mis-elimination by the knockout phase. Let $\mathcal{M}_{\mathcal{F}}$ be the Phase 2 matches. The final score of finalist $k \in \mathcal{F}$ is

$$s_k \;=\; \frac{1}{R'} \sum_{r=1}^{R'} \delta_k^{(r)}, \qquad \delta_k^{(r)} \in \{0,\, 0.5\, w_k^{(r)},\, w_k^{(r)}\}, \tag{5}$$

where $\delta_k^{(r)}$ is the weight k collected in round r (loss / tie / win, scaled by that round's confidence weight $w_k^{(r)}$). Non-finalists receive $s_k = 0$. The verifier selects $\tau^\star = \tau^{(k^\star)}$ with $k^\star = \arg\max_k s_k$.

Complexity. Phase 1 narrows the field from K candidates to a shortlist $\mathcal{F}$ at $K - 1$ pairwise calls. Phase 2 then re-examines $\mathcal{F}$ in a Swiss schedule, which gives any candidate W defeated a second chance to outperform W against a different opponent. Because $|\mathcal{F}| \le \lceil \log_2 K \rceil + 1$ is tiny relative to K, Phase 2 adds at most $R' \lfloor (\lceil \log_2 K \rceil + 1)/2 \rfloor$ pairwise calls. The total pairwise-judge budget is therefore $O(K)$, with detailed call counts in Appendix D.

4 Experiments

Benchmark. We evaluate primarily on *SWE-bench Verified* [28], a suite of real-world Python software-engineering tasks. Each instance pairs a GitHub issue with a specific repository checkout

and a hidden test suite. A candidate `git` patch is judged correct only if it passes all designated fail-to-pass tests together with the pre-existing regression tests.

Baselines. Consistent with [21], we evaluate SRC against three families of execution-free verifiers on a matched candidate pool. **(1) Consistency-based selection.** *Self-Consistency* [13] selects the candidate patch whose diff has the highest average similarity to the remaining $K-1$ candidates. **(2) Direct patch-level scoring.** *Patch Classifier* [21] is an LLM judge that predicts patch correctness and outputs a continuous score in $[0, 1]$. *R2E-Gym Verifier* [14] is a trained patch-level classifier from the R2E-Gym pipeline that conditions on both the patch and the trajectory. **(3) Agent and rubric-based methods.** *Agentic Rubrics* [21], the current state of the art and our closest competitor, lets a rubric agent explore the repository to construct a binary rubric, and a separate LLM judge scores each candidate against it. We additionally report *Oracle@*K (resolved if any of the K rollouts is correct) and *Random / Pass@1* (verifier disabled, one rollout sampled uniformly) as upper and lower bounds.

Metrics. We report two metrics throughout: **Best@**K, the top-1 resolution rate over the full pool, and S**-Acc**, the selection accuracy on the instances whose pool contains both correct and incorrect candidates. Best@K is the headline number, while S-Acc isolates the verifier's discriminative ability on the regime where its choice can actually change the outcome.

4.1 Main Results

Rollout Setup. For each SWE-bench Verified instance we sample $K=16$ independent rollouts using the SWE-agent scaffold [1], at temperature 1.0 with a maximum of 50 turns per rollout. We use two generator models: **Qwen3-Coder-30B-A3B** and **Kimi K2.5**.

Verifier configuration. The *Agentic Rubrics* baseline uses Gemini 3 Pro to drive the agentic rubric generation (max 30 tool calls) and Gemini 2.5 Pro as the downstream judge, and the *Patch Classifier* also uses Gemini 2.5 Pro as its single-call pointwise scorer. Our SRC configuration uses Gemini 2.5 Flash for all three stages. Appendix E reports per-instance call counts and measured API spend. The API budget of SRC is slightly lower than Patch Classifier and roughly one-eighth of Agentic Rubrics.

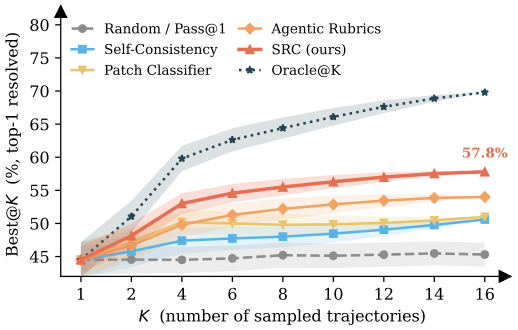


Figure 3: Best@K as a function of K on SWE-bench Verified with Qwen3-Coder-30B-A3B.

Discussion. Table 2 shows SRC as the strongest verifier in every cell across both generator pools, with consistent gains relative to each baseline family. We first examine the Qwen3-Coder pool at $K=16$. **(1)** Self-Consistency picks by surface diff similarity rather than semantic correctness, and SRC beats it by the largest margin we observe, $+7.2$pp Best@16 and $+13.5$pp S-Acc. **(2)** Patch Classifier and R2E-Gym Verifier score each candidate independently rather than against its peers, and SRC adds another $+6$ to $+7$pp Best@16 over them. **(3)** Against Agentic Rubrics, our closest competitor, SRC is both more accurate ($+3.8$pp Best@16 and $+8.2$pp S-Acc) and cheaper, at roughly $1/8$ of AR's per-instance API budget (Appendix E). The Kimi K2.5 pool corroborates these conclusions at a higher absolute level. The stronger generator lifts the Oracle@16 ceiling to 80.8%, and SRC converts that headroom into a strong $\mathbf{73.2}\%$ Best@16, leading Agentic Rubrics by $+2.2$pp and Self-Consistency by $+7.2$pp on the same metric. Two patterns cut across the per-family comparison. SRC's S-Acc lead exceeds its Best@K lead against every paired baseline on both pools, locating the win on the swing instances where the verifier's choice can change the outcome.

4.2 Generalization

To assess whether SRC depends on a particular backbone model and whether it transfers beyond a single task domain, we run three experiments: a backbone study on SWE-bench Verified that runs the entire pipeline under three different model backbones, a cross-domain study on Terminal-Bench 2, and a feature-implementation study on DeepSWE.

Table 2: Best-of-K scaling on SWE-bench Verified. Each cell reports *Best@K* (top-1 resolution rate over the full pool) and *S-Acc* (selection accuracy on the swing instances that contain at least one correct and at least one incorrect candidate). Horizontal rules separate verifier families: bounds (Oracle@K, Random / Pass@1), pointwise execution-free verifiers (Self-Consistency, Patch Classifier, R2E-Gym Verifier), and the rubric-based agentic verifier (Agentic Rubrics).

Generator	Method	K=4		K=8		K=16	
		Best@K	S-Acc	Best@K	S-Acc	Best@K	S-Acc
Qwen3-Coder 30B-A3B	Oracle@K	59.8	100	64.4	100	69.8	100
	Random / Pass@1	44.5	51.0	45.2	52.5	45.3	54.0
	Self-Consistency	47.4	58.1	48.0	61.5	50.6	63.9
	Patch Classifier	50.2	69.2	49.8	65.7	51.0	64.7
	R2E-Gym Verifier	48.6	64.1	49.6	65.3	51.4	65.8
	Agentic Rubrics	49.8	66.0	52.2	70.0	54.0	69.2
	SRC (ours)	**53.0**	**78.2**	**55.5**	**79.2**	**57.8**	**77.4**
Kimi K2.5	Oracle@K	77.2	100	79.2	100	80.8	100
	Random / Pass@1	63.6	52.1	63.2	57.1	63.6	60.2
	Self-Consistency	64.8	56.3	65.6	63.4	66.0	65.7
	Patch Classifier	68.8	70.4	68.8	72.0	68.8	72.2
	R2E-Gym Verifier	69.2	71.8	71.2	78.5	69.2	73.1
	Agentic Rubrics	68.0	67.6	69.2	73.1	71.0	77.3
	SRC (ours)	**70.4**	**76.1**	**71.6**	**79.6**	**73.2**	**82.4**

4.2.1 Generalization on model backbones: open-weight and proprietary models

This subsection tests whether SRC depends on the specific model that drives its three stages. We run the entire pipeline on three backbones from three distinct model families: an open-weight Qwen3.6-27B (Alibaba, served locally on two RTX 6000 Ada GPUs at bf16), the Gemini 2.5 Flash default (Google), and a stronger GPT-5.1 (OpenAI). Figure 4 shows that selection quality scales with backbone capability, with GPT-5.1 reaching the best result at 59.0% Best@16. The fully open-weight Qwen3.6-27B pipeline reaches a competitive 55.0% Best@16, within 2.8 points of the Gemini 2.5 Flash default while running entirely on local hardware. Spanning three independent model families shows that the robustness reflects the design of SRC rather than a single training lineage. Running every stage locally also removes SRC's dependence on a proprietary API, keeping candidate trajectories and induced rubrics on-premise.

4.2.2 Generalization on terminal tasks: Terminal-Bench 2

Terminal-Bench 2 [29] is a benchmark of 89 end-to-end command-line tasks (file editing, shell scripting, environment configuration, and software installation) executed inside a sandboxed Linux environment, with each task graded by a held-out test script run after the agent terminates. We evaluate the *same* SRC architecture on Terminal-Bench 2, with only the output representation $\rho(\tau)$ and the CNS template adapted to terminal sessions (Appendix B.5), using K=5 trajectories across the 89 tasks. Figure 5 reports results on two leaderboard agent submissions, TongAgents + Gemini 3.1 Pro and ForgeCode + GPT-5.4. We compare SRC with Self-Consistency and two training-free selectors from Li et al. [44]. Pointwise judges each trajectory on its own and selects the one judged successful with the highest confidence. Pairwise compares two trajectories at a time in a single bubble-sort pass and keeps the winner of each comparison. All LLM-based methods use GPT-6 Luna as the judge. With this judge, SRC outperforms all baselines on both submissions, with a lift over Pass@1 of +9.6pp on TongAgents and +7.7pp on ForgeCode.

4.2.3 Generalization on feature implementation: DeepSWE

We also test SRC on feature implementation, a different task type from the bug fixes in SWE-bench Verified. Here the agent has to build new behavior from a written request. DeepSWE [45] contains 113 tasks in five programming languages, of which 106 are feature requests, graded by hand-written behavioral tests.

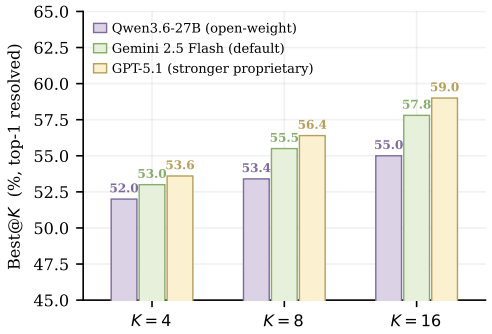


Figure 4: SRC Best@K on the Qwen3-Coder pool under three model backbones, each running the full pipeline end-to-end.

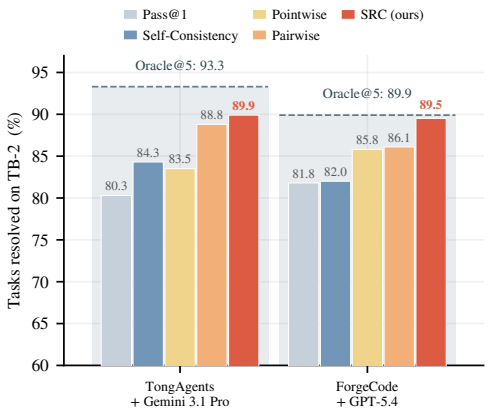


Figure 5: Terminal-Bench 2 resolution rate on two agent submissions. Pointwise and Pairwise follow Li et al. [44]. The shaded area marks Oracle@5.

We use GPT-5.6 Terra trajectories from mini-SWE-agent and take the first K candidates of each task. All methods select from the same candidates. SRC and Patch Classifier use GPT-6 Luna as the judge, and SRC uses prompts adapted to feature requests (Appendix B.4). Table 3 shows that SRC has the highest Best@K at every K. R2E-Gym Verifier is trained on bug-fixing tasks and does not transfer directly to DeepSWE, so it stays close to Random. Patch Classifier saturates from K=4 to K=6. Self-Consistency rises from K=2 to K=4 but drops at K=6. In contrast, SRC improves steadily as K grows, which shows better test-time scaling.

Table 3: Best@K (%) on DeepSWE.

Method	K=2	K=4	K=6
Random	33.6	35.2	34.8
Self-Consistency	36.3	41.6	40.7
R2E-Gym	31.9	35.4	36.3
Patch Classifier	30.1	38.1	38.1
SRC (ours)	**39.8**	**42.5**	**44.2**
Oracle@K	45.1	56.6	61.1

4.3 Ablation Studies

This subsection studies which candidate-aware ingredient inside SRC is responsible for the result. We answer by the following three questions: **1)** Are final-output evidence and trajectory-process evidence complementary? **2)** Should the rubric be conditioned on the submitted candidate batch? **3)** Does comparative ranking outperform isolated pointwise scoring under the *same* rubric?

Each component contributes independently. We measure what each of the three components adds by removing one at a time and comparing the result with the full SRC in Table 4. The full method has the highest S-Acc, so no component can be dropped without a loss. **(1)** SRC w/o COMPARE falls back to pointwise scoring and loses 4.8pp S-Acc. Appendix C.3 shows a case where seven candidates tie under the pointwise scores of Agentic Rubrics, while pairwise comparison separates the correct one. **(2)** SRC w/o READ judges from the final patch without the CNS channel and loses 4.1pp S-Acc. Appendix C.1 shows a case where CNS exposes a band-aid fix that the final patch alone does not reveal. **(3)** SRC w/o SEE replaces the candidate-aware rubric with fixed dimensions and loses 2.6pp S-Acc. Appendix C.2 shows a case where the rubric from SEE finds the axis that separates the candidates, backward compatibility, while the rubric of Agentic Rubrics dilutes it among items that do not discriminate.

Table 4: Component-wise ablation on SWE-bench Verified. Each row enables or disables one of the three SRC stages.

Configuration	Stages			S-Acc
	SEE	READ	COMPARE	
SRC w/o COMPARE	✓	✓	×	72.6%
SRC w/o READ	✓	×	✓	73.3%
SRC w/o SEE	×	✓	✓	74.8%
SRC (**ours**)	✓	✓	✓	**77.4%**

Table 5: Rubric-source comparison on the Qwen3-Coder pool with the Gemini 2.5 Flash judge and no CNS. "Batch" = the rubric sees the submitted candidate batch. "Repo" = rubric induction interacts with the repository. "Calls" = LLM calls to build the rubric.

Rubric source	Batch	Repo	Calls	S-Acc (%)	Best@16 (%)
Generic / fixed	×	×	0	60.2	48.6
Problem-only	×	×	1	62.8	50.0
Agentic Rubrics	×	✓	~ 23	69.2	54.0
SEE (ours)	✓	×	1	**73.3**	**55.6**

Table 6: Aggregation-strategy comparison on the Qwen3-Coder pool with the Gemini 2.5 Flash judge, holding the rubric and CNS channels fixed at SRC default. Per-instance judge calls are the cost axis.

Aggregation	Calls	S-Acc (%)	Best@16 (%)
Pointwise	16	72.6	55.2
Knockout	15	76.3	57.2
Swiss	32	76.7	57.4
Round-robin	120	77.1	57.6
Hybrid Swiss (ours)	21	**77.4**	**57.8**

Rubric source. We compare rubric sources with each one used in its native form. The Generic rubric is a fixed template shared by all instances. The problem-only rubric is generated by an LLM from the problem statement alone. Table 5 shows that SEE with COMPARE reaches 73.3% S-Acc, which is 13.1pp above the Generic rubric, 10.5pp above problem-only induction and 4.1pp above Agentic Rubrics. SEE needs one LLM call and no repository access, while Agentic Rubrics needs about 23 calls and access to the repository.

Comparative ranking beats isolated pointwise scoring. We hold the rubric and CNS channels fixed and vary only the aggregation strategy used by the COMPARE stage, on the Qwen3-Coder pool with the Gemini 2.5 Flash judge. Table 6 reports each strategy's S-Acc and Best@16 alongside its judge calls per instance, which serves as the inference-cost axis. Two patterns stand out. First, pairwise comparison clearly beats pointwise scoring: pointwise trails every pairwise aggregator by 3.7 to 4.8pp S-Acc and 2.0 to 2.6pp Best@16. Second, Hybrid Swiss achieves the best balance of accuracy and cost among the four pairwise aggregators. Round-robin spends 120 calls per instance, almost six times the 21 of Hybrid Swiss, with no accuracy gain. Hybrid Swiss stays in the same accuracy band as vanilla Swiss at two-thirds of its calls, and its two-phase design keeps the linear $O(K)$ budget of Knockout while avoiding the early-elimination risk of a single-elimination bracket (Algorithm 1). Hybrid Swiss thus occupies the top of the cost-accuracy Pareto frontier on this pool, reaching the pairwise accuracy band at an $O(K)$ call budget.

5 Conclusion

We present SRC, a training-free and execution-free verifier for agent test-time scaling. SRC uses the candidate batch as shared context across three stages: SEE induces an instance-specific rubric from task context and candidate disagreement, targeting the dimensions on which candidates actually differ. READ compresses each trajectory into a grounded Cognitive Narrative Summary, surfacing behavioral evidence that final-patch-only judges miss. COMPARE performs rubric-guided pairwise selection with a two-phase hybrid tournament, providing the reliability of pairwise judgment at $O(K)$ comparison cost. On SWE-bench Verified SRC reaches 57.8% Best@16 on the Qwen3-Coder pool and 73.2% on the Kimi K2.5 pool, without verifier training or candidate re-execution. Component ablations further show that SEE, READ, and COMPARE each contribute to the final selection accuracy. The same architecture also carries over to Terminal-Bench 2 and to feature implementation on DeepSWE. More broadly, these findings suggest a direction for execution-free agent verification: derive the criteria from where the candidates differ, judge each candidate by its trajectory evidence as well as its final output, and choose by comparing candidates with each other rather than scoring them alone.

References

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

[2] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software

developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

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

[4] Yingwei Ma, Yongbin Li, Yihong Dong, Xue Jiang, Yanhao Li, Yue Liu, Rongyu Cao, Jue Chen, Fei Huang, and Binhua Li. Thinking longer, not larger: Enhancing software engineering agents via scaling test-time compute. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 3730–3741. IEEE, 2025.

[5] King Zhu, Hanhao Li, Siwei Wu, Tianshun Xing, Dehua Ma, Xiangru Tang, Minghao Liu, Jian Yang, Jiaheng Liu, Yuchen Eleanor Jiang, et al. Scaling test-time compute for llm agents. *arXiv preprint arXiv:2506.12928*, 2025.

[6] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314*, 2024.

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

[8] Runze Liu, Junqi Gao, Jian Zhao, Kaiyan Zhang, Xiu Li, Biqing Qi, Wanli Ouyang, and Bowen Zhou. Can 1b llm surpass 405b llm? rethinking compute-optimal test-time scaling. *arXiv preprint arXiv:2502.06703*, 2025.

[9] Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Wenyue Hua, Haolun Wu, Zhihan Guo, Yufei Wang, Niklas Muennighoff, et al. A survey on test-time scaling in large language models: What, how, where, and how well? *arXiv preprint arXiv:2503.24235*, 2025.

[10] Yebin Yang, Huaijin Wu, Jingtao Han, Yu Wang, Xiaohan Qin, Jingzhi Wang, Debing Zhang, and Junchi Yan. Inner-layer token self-modulation as another scaling axis for llms. In *Forty-third International Conference on Machine Learning*.

[11] V Venktesh, Mandeep Rathee, and Avishek Anand. Trust but verify! a survey on verification design for test-time scaling. *arXiv preprint arXiv:2508.16665*, 2025.

[12] Harman Singh, Xiuyu Li, Kusha Sareen, Monishwaran Maheswaran, Sijun Tan, Xiaoxia Wu, Junxiong Wang, Alpay Ariyak, Qingyang Wu, Samir Khaki, et al. v_1: Unifying generation and self-verification for parallel reasoners. *arXiv preprint arXiv:2603.04304*, 2026.

[13] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[14] Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025.

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

[16] KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, X. W., Jiaxi Yang, Yuzhen Huang, Junyang Lin, and Junxian He. SWE-RM: Execution-free feedback for software engineering agents. In *International Conference on Learning Representations*, 2026.

[17] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in neural information processing systems*, 36:46595–46623, 2023.

[18] Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Lingpeng Kong, Qi Liu, Tianyu Liu, et al. Large language models are not fair evaluators. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9440–9450, 2024.

[19] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, et al. A survey on llm-as-a-judge. *The Innovation*, 2024.

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

[21] Mohit Raghavendra, Anisha Gunjal, Bing Liu, and Yunzhong He. Agentic rubrics as contextual verifiers for SWE agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15265–15290, 2026.

[22] Xiangdong Zhang, Xiaohan Qin, Sunan Zou, Tuo Dai, Xiaoming Shi, Huaijin Wu, Yebin Yang, Zhuo Xia, Shaofeng Zhang, Lin Yao, et al. xhc: Expanded hyper-connections. *arXiv preprint arXiv:2607.14530*, 2026.

[23] Huaijin Wu, Lianqiang Li, Hantao Huang, Yi Tu, Jihang Zhang, Minghui Yu, and Junchi Yan. Hshare: Fast llm decoding by hierarchical key-value sharing. In *International Conference on Learning Representations*, volume 2025, pages 89412–89428, 2025.

[24] Huaijin Wu, Bingrui Li, Yebin Yang, Yi Tu, Zhanpeng Zhou, Jianfei Chen, and Junchi Yan. Achieving low-bit muon through subspace preservation and grid quantization. In *The Fourteenth International Conference on Learning Representations*, 2026.

[25] Xingyu Zhu, Beier Zhu, Junfeng Fang, Shuo Wang, Yin Zhang, Xiang Wang, and Xiangnan He. GuardAlign: Test-time safety alignment in multimodal large language models. In *International Conference on Learning Representations*, 2026.

[26] Xingyu Zhu, Kesen Zhao, Liang Yi, Shuo Wang, Zhicai Wang, Beier Zhu, Hanwang Zhang, and Xiangnan He. Look carefully: Adaptive visual reinforcements in multimodal large language models for hallucination mitigation. In *International Conference on Learning Representations*, 2026.

[27] Mohammad Ali Alomrani, Yingxue Zhang, Derek Li, Qianyi Sun, Soumyasundar Pal, Zhanguang Zhang, Yaochen Hu, Rohan Deepak Ajwani, Antonios Valkanas, Raika Karimi, et al. Reasoning on a budget: A survey of adaptive and controllable test-time compute in llms. *arXiv preprint arXiv:2507.02076*, 2025.

[28] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The twelfth international conference on learning representations*, 2023.

[29] Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

[30] Xinhao Zheng, Yang Li, Cunxin Fan, Huaijin Wu, Xinhao Song, and Junchi Yan. Learning plaintext-ciphertext cryptographic problems via anf-based sat instance representation. *Advances in Neural Information Processing Systems*, 37:41649–41669, 2024.

[31] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

[32] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[33] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. *arXiv preprint arXiv:2211.14275*, 2022.

[34] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *The twelfth international conference on learning representations*, 2024.

[35] Congming Zheng, Jiachen Zhu, Zhuoying Ou, Yuxiang Chen, Kangning Zhang, Rong Shan, Zeyu Zheng, Mengyue Yang, Jianghao Lin, Yong Yu, et al. A survey of process reward models: From outcome signals to process supervisions for large language models. *arXiv preprint arXiv:2510.08049*, 2025.

[36] Bin Yu, Xinming Wang, Shijie Lian, Haotian Li, Changti Wu, Ruina Hu, Bailing Wang, Yuliang Wei, and Kai Chen. Trajselector: Harnessing latent representations for efficient and effective best-of-n in large reasoning model. *arXiv preprint arXiv:2510.16449*, 2025.

[37] Shalev Lifshitz, Sheila A McIlraith, and Yilun Du. Multi-agent verification: Scaling test-time compute with multiple verifiers. *arXiv preprint arXiv:2502.20379*, 2025.

[38] Shubham Ugare and Satish Chandra. Agentic code reasoning. *arXiv preprint arXiv:2603.01896*, 2026.

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

[40] Tianci Liu, Ran Xu, Tony Yu, Ilgee Hong, Carl Yang, Tuo Zhao, and Haoyu Wang. Openrubrics: Towards scalable synthetic rubric generation for reward modeling and llm alignment. *arXiv preprint arXiv:2510.07743*, 2025.

[41] Akira Kawabata and Saku Sugawara. C2: Scalable rubric-augmented reward modeling from binary preferences. *arXiv preprint arXiv:2604.13618*, 2026.

[42] Yantao Liu, Zijun Yao, Rui Min, Yixin Cao, Lei Hou, and Juanzi Li. PairJudge RM: Perform best-of-n sampling with knockout tournament. *arXiv preprint arXiv:2501.13007*, 2025.

[43] Daniel Kahneman. Thinking, fast and slow. *Farrar, Straus and Giroux*, 2011.

[44] Xiaochuan Li, Ryan Ming, Pranav Setlur, Abhijay Paladugu, Andy Tang, Hao Kang, Shuai Shao, Rong Jin, and Chenyan Xiong. Evaluating test-time scaling of general LLM agents. *arXiv preprint arXiv:2602.18998*, 2026.

[45] Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026.

A Limitations

SRC is designed for execution-free selection, and does not replace executable tests, formal checks, or human review when those signals are available. Like other LLM-as-judge methods, it can inherit judge-model biases or context-length limits. In addition, our empirical coverage focuses on software-engineering and terminal-agent benchmarks, extending the protocol to other agent domains and safety-critical settings remains future work.

B Stage prompts: SEE, READ, COMPARE

We list the system and user prompts that drive each of SRC's three stages, exactly as used in our experiments. Placeholders such as `{problem_statement}` are Python f-string substitutions filled at call time. Em-dashes and arrow glyphs in the original prompts are rendered as ASCII (- and ->) here for typesetting; the runtime strings use Unicode.

B.1 SEE stage (contrastive rubric generation)

SEE: System Prompt

```
You are an expert software engineer. You will receive a problem description
    and several diverse patches that attempt to fix it.
Your job: analyse the patches contrastively and produce evaluation
    dimensions that a pairwise judge can use to decide which patch is better.

Think step by step:
1. ROOT CAUSE -- What is the underlying bug? Extract the specific error,
    traceback, or unexpected behavior from the problem statement.
2. PATCH LANDSCAPE -- For each sample patch, note:
   - Which file(s) / class(es) / function(s) it modifies (use exact paths
    from the diff)
   - What fix strategy it uses
   - Potential flaws you can spot
3. KEY DIVERGENCE POINTS -- Where do the patches DISAGREE? These are the
    most discriminative axes.
4. COMMON GROUND -- What do most patches agree on? These are NOT useful as
    evaluation criteria (no discriminative power).
5. FAILURE MODES TO WATCH FOR -- Based on the divergence, what are the most
    likely ways a patch can be wrong?
   - Band-Aid Fix: suppresses the symptom without fixing the root cause
   - Wrong Layer: changes the wrong module / class / function
   - Root Cause Missed: partially addresses the issue, leaves core bug
    present
   - Missing Edges: does not handle boundary / corner cases from the issue
   - Scope Creep: introduces unrelated refactors or changes

Output a JSON object with two fields:
1. "analysis": your step-by-step contrastive analysis (string)
2. "dimensions": an object with exactly 4 keys: root_cause, fix_correctness,
    change_scope, edge_cases
   Each dimension has:
     - "question": a discriminative question for a pairwise judge ("Which
    patch better ...?")
     - "indicators": array of 2-4 specific, concrete indicators the judge
    should look for -- grounded in file paths, class names, error messages,
    or code patterns observed in the patches. Each indicator starts with a
    verb (Modifies, Fixes, Avoids, Handles, Preserves, Checks).
```

SEE: User Prompt Template

```
## Problem Statement (GitHub Issue)
{problem_statement}

## Sample Patches ({n} diverse attempts)
### Sample Patch #1
```diff
{patch_1}
```
Sample Patch #2
```diff
{patch_2}
```
Sample Patch #3
```diff
{patch_3}
```

Analyse the patches contrastively and generate the evaluation dimensions
    JSON.
```

B.2 READ stage (Cognitive Narrative Summary)

Table 7: The five dimensions of CNS(τ) produced by READ.

Dimension	Structured content
Fault hypothesis chain	Each hypothesis → evidence → outcome
Causal model	Root cause → mechanism → symptom
Intervention analysis	Files changed → effect → targeted
Verification status	Tests run and edges checked
Cognitive red flags	Overconfidence, unrecovered errors, phase skipping, excessive exploration, unverified success claims

READ: User Prompt Template

```
You are an expert software engineering analyst. Given an SWE-agent
    trajectory (a sequence of the agent's thoughts, actions, and
    observations while attempting to fix a repo issue), produce a structured
    Cognitive Narrative Summary (CNS).

IMPORTANT: Focus on the OBSERVABLE FACTS and EPISTEMIC QUALITY of the
    agent's reasoning process, NOT on whether the agent claimed success.
    Agents frequently claim success when their fix is wrong. If the agent
    CLAIMS something in its thought but the observation does not confirm it,
    flag as DISCREPANCY.

## Problem Statement
{problem_statement}

## Agent Trajectory
{trajectory}

## Final Patch (git diff)
{patch}

## Behavioural Features (deterministic, computed from the trajectory)
{features}    # total_turns, phase_completeness, explore_exploit_ratio,
```

```
            # error_rate, recovery_rate, has_verification_phase,
            # files_explored, patch_locality, total_edits, total_tests,
            # hypothesis_changes, self_corrections, patch_lines_changed,
            # metacognitive_score

## Instructions
Produce a CNS with EXACTLY these 5 sections (be concise, ~50-100 words per
    section):

### 1. FAULT HYPOTHESIS CHAIN
List each fault hypothesis the agent formed, with status:
- H1: [hypothesis] -> [evidence for/against] -> [Confirmed/Refuted/Abandoned]
- H2: ...

### 2. CAUSAL MODEL
What is the agent's implicit causal model?
[Root Cause] -> [Mechanism] -> [Symptom]
Cite the key observation that supports this model.

### 3. INTERVENTION ANALYSIS
- Files changed: [list]
- What each change does: [brief]
- Is the intervention targeted and minimal? [Yes/Partially/No]

### 4. VERIFICATION STATUS
- Did the agent run tests? [Yes/No]
- Did the agent check edge cases? [Yes/No]

### 5. COGNITIVE RED FLAGS
List any red flags from the trajectory:
- Overconfidence without evidence?
- Unrecovered errors?
- Phase skipping (implement without localize)?
- Excessive exploration without convergence?
- Agent claimed success without verification?
```

B.3 COMPARE stage (rubric-guided pairwise judging)

COMPARE: User Prompt Template

```
You are an expert judge comparing two candidate patches for the same
    software problem.
Your goal: determine which patch is MORE LIKELY to be a CORRECT fix.

## Problem Statement
{problem_statement}

## Patch A
```diff
{patch_a}
```

Patch B
```diff
{patch_b}
```

Cognitive Narrative Summary
CNS for Patch A
{cns_a}
CNS for Patch B
```

```
{cns_b}

## Evaluation Rubric
For each dimension below, determine which patch better satisfies the
    criteria.

### root_cause
**{question}**
  - {indicator_1}
  - {indicator_2}
  ...
### fix_correctness
**{question}**
  - {indicator_1}
  - {indicator_2}
  ...
### change_scope
**{question}**
  - {indicator_1}
  - {indicator_2}
  ...
### edge_cases
**{question}**
  - {indicator_1}
  - {indicator_2}
  ...

## Instructions
1. For each dimension, compare the two patches against the indicators. State
    which is better (A/B/TIE).
2. Weigh the dimensions by importance -- root cause and correctness matter
    more than style or scope.
3. Give an overall verdict based on the dimension judgments.

Confidence guide:
- "high": patches differ meaningfully and one is clearly better
- "medium": patches differ but the better choice requires inference
- "low": patches are very similar or evidence is ambiguous

## Output (strict JSON, no markdown):
{
    "root_cause": "A" or "B" or "TIE",
    "fix_correctness": "A" or "B" or "TIE",
    "change_scope": "A" or "B" or "TIE",
    "edge_cases": "A" or "B" or "TIE",
    "overall": "A" or "B" or "TIE",
    "confidence": "high" or "medium" or "low",
    "reasoning": "2-3 sentences citing specific evidence"
}
```

B.4 Prompt changes for feature tasks

DeepSWE tasks request new behavior rather than a bug fix, so we adapt the three prompts and keep the pipeline. SEE first extracts the required behaviors from the request, and its rubric dimensions become requirement coverage, behavior correctness, contract match, and edge cases. READ keeps five sections, which become design decision chain, behavior model, intervention analysis, verification status per requirement, and cognitive red flags. COMPARE asks which patch is more likely to pass hidden behavioral tests, gives most weight to behavior correctness and contract match, and ignores style and diff size. A tied knockout match is replayed with the two candidates swapped, so bracket position does not decide a tie.

B.5 Prompt changes for terminal tasks

Terminal-Bench 2 candidates are terminal sessions rather than patches, so we adapt the three prompts and keep the pipeline. SEE reads sample trajectories and compares their approaches, commands and outcomes, and its rubric dimensions become task completion, output correctness, error resolution, and approach soundness. READ keeps five sections, which become approach, key actions, error handling, verification status, and red flags. It also receives structural features such as step count, error rate, and whether a verification step exists, and it is told to rely on terminal output rather than the agent's claims. COMPARE asks which trajectory is more likely to have completed the task, treats terminal output as ground truth, and gives most weight to task completion and output correctness.

C Case studies

This appendix walks through three SWE-bench Verified instances side-by-side to make the per-instance mechanism behind SRC's gain concrete. *Case 1* (§C.1) shows how the CNS channel exposes a band-aid pattern in the agent's reasoning that a raw-trajectory or final-patch-only view misses. *Case 2* (§C.2) shows how the contrastive rubric induced by SEE picks up a discriminative axis (backward compatibility) that AR's fixed-axis rubric dilutes among ∼10 non-discriminative "chaff" items. *Case 3* (§C.3) shows how SRC's pairwise COMPARE stage resolves a 7-way tie that AR's pointwise weighted sum cannot break, with AR falling back to a lexicographic argmax that picks the wrong candidate.

C.1 Case 1: CNS exposes band-aid intent (`django-12125`)

We illustrate SRC's contrastive-rubric-plus-pairwise-judging architecture against the per-patch agentic-rubric scorer of Raghavendra et al. [21] on a single SWE-bench Verified instance, by walking through the artefacts each verifier produces.

Bug summary

Django `makemigrations` mis-serializes inner classes: a model field defined as `Outer.Inner` is written into the migration file as `test1.models.Inner` instead of `test1.models.Outer.Inner`. The canonical fix replaces `obj.__class__.__name__` with `obj.__class__.__qualname__` in either `django/utils/deconstruct.py` or `TypeSerializer.serialize` of `django/db/migrations/serializer.py`; the symptom is a one-line root-cause issue and any patch that re-parses the already-incorrect path string downstream is a band-aid.

Pool composition. Of the $K{=}16$ trajectories sampled for this instance, only two produced correct fixes ($k{=}10$ and one other); the remaining 14 produced patches that look superficially plausible but apply band-aid string-parsing in the wrong layer (typically `rsplit('.', 1)` inside `_serialize_path`).

Outcome. SRC picks $k{=}10$ (correct); Agentic Rubrics picks $k{=}0$ (incorrect band-aid).

CNS for $k{=}10$ (SRC's pick; correct).

```
### 1. FAULT HYPOTHESIS CHAIN
- H1: The `makemigrations` command generates incorrect paths for inner
    classes
      because the serialization process for types does not correctly capture
      the fully qualified name.
  Evidence: The problem description explicitly states `test1.models.Inner`
            instead of `test1.models.Outer.Inner`. The agent's initial search
            for "deconstruct" and "serialize" in migration-related files
            (writer.py, serializer.py) directly targets this mechanism.
  Status:   Confirmed.
- H2: The `TypeSerializer` in `django/db/migrations/serializer.py` is
```

```
      responsible for generating the string representation of types, and it
      incorrectly uses `__name__` instead of `__qualname__` for nested
    classes.
  Evidence: The agent repeatedly views `serializer.py` and ultimately
    modifies
            the `TypeSerializer` to incorporate `__qualname__`.
  Status:   Confirmed.

### 2. CAUSAL MODEL
Root Cause:  Django's 'TypeSerializer' uses '__name__' for type
    serialization.
Mechanism:  `__name__` provides only the immediate name of a class, omitting
    its
            enclosing class context, which is crucial for nested classes.
Symptom:    `makemigrations` produces migration files with unqualified paths
    for
            inner classes (e.g. `test1.models.Inner` instead of
            `test1.models.Outer.Inner`).
Key Observation: The agent's final patch directly modifies `TypeSerializer`
    to
                prioritize `__qualname__` over `__name__` for non-builtin
                types, directly addressing the identified mechanism.

### 3. INTERVENTION ANALYSIS
- Files changed: `django/db/migrations/serializer.py`
- The change modifies 'TypeSerializer.serialize'.  It introduces logic to use
    'self.value.__qualname__' for constructing the type's string
    representation, with a fallback to 'self.value.__name__' if
    '__qualname__' is unavailable or contains '<' (local / anonymous class),
    ensuring robustness.
- Targeted and minimal:  YES. The change is localized to a single method in
    one file, directly addressing the identified serialization logic.

### 4. VERIFICATION STATUS
- Did the agent run tests? Yes. The agent created `simple_final_test.py`
  (Turn 40) and executed it (Turn 41).
- Did the agent check edge cases? Partially. Agent's thought in Turn 42
  explicitly mentions checking "Complex nesting" and "Edge cases handled
  properly," and the patch itself includes logic for `__qualname__`
    fallbacks.

### 5. COGNITIVE RED FLAGS
- Overconfidence without evidence: No.
- Unrecovered errors: No.
- Phase skipping (implement without localize): No.
- Excessive exploration without convergence: Partially; the agent explored
  many `django/db/models/fields/*.py` files after initially identifying
  `serializer.py`, but eventually converged on the correct file.
- Agent claimed success without verification: No. Verified with a custom test
  script before submission.
```

CNS for k=0 (Agentic Rubrics' pick; band-aid)

```
### 1. FAULT HYPOTHESIS CHAIN
- H1: The `makemigrations` command incorrectly serializes paths for inner
      classes.
  Evidence: Problem description explicitly states this. -> Confirmed.
- H2: The issue resides in the migration serialization logic, specifically
      within `django/db/migrations/serializer.py`.
  Evidence: Agent's `find` and `grep` actions for "deconstruct" and
            "serialize" led to this file. -> Confirmed.
```

```
- H3: The `_serialize_path` method in `DeconstructableSerializer` incorrectly
      parses fully qualified names of nested classes, failing to preserve the
      full parent path.
  Evidence: Agent's patch directly modifies this method to handle multi-part
            paths for nested classes. -> Confirmed.

### 2. CAUSAL MODEL
Root Cause: The `_serialize_path` method in
    `django/db/migrations/serializer.py`
            uses a simplistic `rsplit(".", 1)` approach, which fails to
            correctly identify the full module path for deeply nested
    classes.
Mechanism:  This incorrect parsing leads to the generated migration file
            referencing the inner class as if it were a top-level class
    within
            its module, rather than a nested class within its parent.
Symptom:    `manage.py makemigrations` produces migration files with
    incorrect
            field references like `test1.models.Inner` instead of
            `test1.models.Outer.Inner`.
Key Observation: The agent's modification to `_serialize_path` to explicitly
                 handle `parts = path.split(".")` and reconstruct the
                 `module_parts` and `name` supports this causal model.

### 3. INTERVENTION ANALYSIS
- Files changed: `django/db/migrations/serializer.py`
- The `_serialize_path` static method is modified to parse the input `path`
    by
  splitting it into all its components. It then reconstructs the module and
  name, ensuring that for nested classes, the full qualified path is
    preserved
  in the `name` variable, while still handling the `django.db.models`
  shorthand.
- Targeted and minimal:  PARTIALLY. It is targeted at the identified
    problematic method.  However, the new logic introduces additional
    complexity and conditional branches, and duplicates some existing logic,
    making it less minimal than it could be.

### 4. VERIFICATION STATUS
- Did the agent run tests? Yes. The agent attempted to run a specific
  `unittest` (`tests.field_deconstruction.tests.FieldDeconstructionTests.
  test_char_field`) in Turn 46.
- Did the agent check edge cases? Yes. The agent explicitly states checking
  "Regular Django fields like `CharField` still work" in Turn 46.

### 5. COGNITIVE RED FLAGS
- Overconfidence without evidence:  YES. In Turn 47, after a test resulted in
    an ‘ERROR‘, the agent stated, "No matter, I have confidence in my fix
    based on manual verification," dismissing an observable failure without
    investigation.
- Unrecovered errors:  YES. The ‘unittest‘ command in Turn 46 produced an
    ‘ERROR‘ that the agent did not investigate or resolve before proceeding
    to submission.
- Agent claimed success without verification:  YES. The agent claimed success
    and submitted the solution despite an observed test ‘ERROR‘ and without
    providing any observable evidence for the "manual verification" it
    claimed.
```

Why SRC succeeds. Two factors combine on this instance. *First,* CNS *surfaces the band-aid pattern.* The CNS for k=10 records the agent modifying `TypeSerializer.serialize` to use `__qualname__` with a clean fallback to `__name__`, and is flagged TARGETED AND MINIMAL: YES. The CNS for k=0 records the agent reaching for `rsplit('.', 1)` to manually re-parse an already-incorrect path string, and is flagged PARTIALLY on minimality; its COGNITIVE RED FLAGS field

surfaces three items in succession – overconfidence, an unrecovered unit-test ERROR, and a claim of success without verification. The pairwise judge sees both CNS blocks side by side and reads the band-aid pattern directly off the structured fields rather than having to infer it from raw trajectory text. *Second, pairwise comparison resolves what per-patch scoring blurs.* Under independent per-patch scoring, k=0 collects credit on the runtime items (R1 no `ValueError`, R2 correct dotted path, R3 no regressions) and on the integrity items (I1–I3 preserve existing `deconstruct.py` behavior, which k=0 trivially does by not touching the file at all). The two patches' weighted sums land close enough that AR's argmax falls on k=0; under pairwise comparison, the rubric's "which addresses the source" framing forces a head-to-head and the CNS-augmented judge resolves it with `high` confidence in favor of k=10.

C.2 Case 2: contrastive rubric beats fixed-axis rubric (`pydata__xarray-3993`)

This case isolates the *rubric* channel. We hold the judge model fixed and walk through how SRC's contrastive rubric (output of SEE) and Agentic Rubrics' fixed-axis rubric score the same two candidate patches.

Bug summary

`xarray`'s `DataArray.integrate` uses a `dim` parameter, while its sibling `Dataset.integrate` (and both `differentiate` methods) use `coord` – the API is inconsistent. The PR description explicitly asks whether a deprecation cycle is needed.

- *Gold fix* (k=5): rename `dim`→`coord` *and* introduce a deprecation cycle (`FutureWarning`, backward-compatible mapping).
- *Wrong fix* (k=10): rename only – a breaking change that drops backward compatibility.

Pool composition and outcome. K=16 trajectories; only one (k=5) implements the deprecation cycle. SRC picks k=5 (correct); Agentic Rubrics picks k=10 (the breaking rename) because its weighted sum dilutes the discriminative axis (*backward compatibility*) among ten non-discriminative items.

SRC rubric (output of SEE for xarray-3993; `change_scope` is the discriminative dimension)

```
{
  "id": "root_cause",
  "description": "Which patch most directly and accurately addresses the API
    inconsistency between `DataArray.integrate` and `Dataset.integrate`
    regarding the integration axis argument?",
  "indicators": [
    "Modifies the `DataArray.integrate` method signature to use `coord` as
    the primary parameter.",
    "Updates the docstring of `DataArray.integrate` in
    `xarray/core/dataarray.py` to refer to `coord`.",
    "Changes the internal call `self._to_temp_dataset().integrate` to pass
    `coord` instead of `dim`."
  ]  # both candidates satisfy this dimension
}
{
  "id": "fix_correctness",
  "description": "Which patch provides the most appropriate and robust
    solution for the `DataArray.integrate` method, considering both the
    desired API and potential user impact?",
  "indicators": [
    "Correctly renames the `dim` parameter to `coord` in
    `xarray/core/dataarray.py`.",
    "Ensures the `integrate` method functions correctly with the new `coord`
    parameter.",
```

```
    "Avoids introducing new bugs or unexpected behavior in argument parsing
    (e.g., with positional arguments)."
  ]  # both satisfy
}
{
  "id": "change_scope",
  "description": "Which patch implements the change with the most
    appropriate scope, considering the need for API consistency versus
    backward compatibility?",
  "indicators": [
    "Introduces a deprecation warning for the old ‘dim‘ argument in
    ‘xarray/core/dataarray.py‘.",
    "Provides backward compatibility for existing code that uses ‘dim‘ as an
    argument.",
    "Avoids unnecessary changes or refactoring outside the immediate problem
    of parameter renaming."
  ]    # <- the decisive dimension; only k=5 satisfies the first two
    indicators
}
{
  "id": "edge_cases",
  "description": "Which patch best handles the transition for existing users
    and potential
                  edge cases related to argument passing?",
  "indicators": [
    "Handles cases where both `dim` and `coord` might be passed to
    `DataArray.integrate`.",
    "Manages positional arguments gracefully during the transition period.",
    "Clearly communicates the deprecation to users through warnings or
    documentation (if applicable)."
  ]
}
```

Why SRC succeeds. SEE's rubric is induced *after* sampling three diverse candidate patches from the batch. The contrastive prompt explicitly asks the LLM to "identify common ground" (which is then *excluded* from the rubric since it has no discriminative power) and to "extract divergence points". On this instance the divergence is exactly the deprecation cycle, and the LLM lifts it into a top-level rubric dimension named `change_scope` whose two key indicators – *introduces a deprecation warning* and *provides backward compatibility* – decide the verdict in one head-to-head pairwise call. SRC's contrastive sampling pre-filters the rubric to the dimensions on which the candidates actually disagree. In contrast, AR's exhaustiveness is a liability when most rubric items are non-discriminative.

C.3 Case 3: pairwise COMPARE resolves a 7-way AR tie (`pydata__xarray-7393`)

This case isolates the *aggregation* channel. AR's rubric is reasonable on its surface. It identifies the right files and the right behavioral test. The discriminative items, however, prescribe a single implementation path. As a result, the LLM grader collapses correct and almost-correct candidates onto the same binary score, and AR's pointwise weighted sum has no information left to separate them.

Bug summary

`xarray`'s `PandasMultiIndexingAdapter.__array__` extracts level values from the underlying `pd.MultiIndex` without casting them back to the original `dtype`. As a result, an `int32` coordinate is silently upcast to `int64` after a `Dataset.stack`. The fix must preserve the original integer precision through `stack`/`unstack`.

- *Gold fix* (k=10): inside `PandasMultiIndexingAdapter.__array__`, casts the extracted level values to `self.dtype` on the `self.level is not None` branch.

- *Plausible alternatives* (the other six tied candidates): patch a different code path such as `PandasMultiIndex.stack`, the `PandasIndexingAdapter` sibling, or the downstream symptom. Each either changes behavior the rubric does not check, or leaves the upstream cause untouched.

Pool composition and outcome. K=16 trajectories. SRC picks k=10 (correct), while seven tie at AR's maximum weighted score $18/35 = 0.514$, and it breaks the tie lexicographically and picks k=3 (wrong).

Agentic Rubrics for xarray-7393

```
file_change_rubrics: FC1 [w=3] Modifies
    ‘PandasMultiIndexingAdapter.__array__‘ in ‘xarray/core/indexing.py‘. #
    0/7 tied pass
  FC2 [w=3] Updates the ‘self.level is not None‘ branch to cast level values
    to ‘self.dtype‘. # 0/7 tied pass
  FC3 [w=2]  Avoids modifying `PandasMultiIndex.stack` in
    `xarray/core/indexes.py`.# 7/7

  FC4 [w=2]  Avoids modifying `PandasIndexingAdapter.__array__`.
                           # 7/7

spec_alignment_rubrics:
  SA1 [w=3] Ensures ‘stack‘ operations preserve the original integer
    precision (e.g. int32).  # 0/7 tied pass
  SA2 [w=3] Implements dtype preservation by explicitly using
    ‘self.dtype‘.   # 0/7 tied pass
  SA3 [w=2] Supports passing an explicit ‘dtype‘ and prioritizes it over
    ‘self.dtype‘.         # 0/7 tied pass

integrity_rubrics:
  I1  [w=3]  Preserves the fallback to `super().__array__(dtype)` when
    `self.level is None`.        # 7/7

  I2  [w=2]  Maintains correct handling of `pd.PeriodIndex`.
                           # 7/7

  I3  [w=3]  Avoids changing the signature of
    `PandasMultiIndexingAdapter.__array__`.               # 7/7

runtime_rubrics:
  R1 [w=3] Evaluates ‘ds['a'].values.dtype ==
    ds.stack(b=('a',))['a'].values.dtype‘ to True.   # 0/7 tied pass
  R2  [w=3]  Passes existing `stack`/`unstack` test suites.  # 7/7 pass
  R3  [w=3]  Avoids raising `TypeError`/`ValueError` when accessing level
    values of a stacked MultiIndex. # 7/7
```

Table 8: `pydata__xarray-7393`: the seven candidates tied at AR's maximum weighted score (top row, all $0.514 = 18/35$) and SRC's COMPARE tournament score on the same candidates (bottom row, normalized win rate). $\star$ marks the gold candidate, † marks AR's lexicographic pick. The tournament cleanly separates k=10 from the rest.

Candidate k	$3^{\dagger}$	6	8	$10^{\star}$	11	13	14
AR weighted (/35)	0.514	0.514	0.514	**0.514**	0.514	0.514	0.514
SRC COMPARE	0.438	0.500	0.625	**1.000**	0.375	0.125	0.500

Why SRC succeeds. SRC's COMPARE stage replaces "does this candidate satisfy item i?" with "does candidate A or candidate B better address the bug?". The judge reads each pair's final patch and

Algorithm 1 COMPARE *(default in* SRC*)*: knockout + Swiss on the champion's path-losers.

Require: trajectories $\mathcal{T} = \{\tau^{(1)}, \ldots, \tau^{(K)}\}$, rubric $\mathcal{R}$, judge J, task t
Ensure: selected trajectory $\tau^\star$

$\mathcal{S} \leftarrow [1, 2, \ldots, K]$; $\mathcal{P}_k \leftarrow \emptyset$ for all k ▷ Phase 1: knockout
while $|\mathcal{S}| > 1$ **do**
  $\mathcal{S}' \leftarrow \emptyset$
  for $i = 0, 2, \ldots, 2\lfloor(|\mathcal{S}| - 1)/2\rfloor$ **do**
    $a \leftarrow \mathcal{S}[i]$, $b \leftarrow \mathcal{S}[i{+}1]$
    $(v, c, \cdot) \leftarrow J(\tau^{(a)}, \tau^{(b)} \mid t, \mathcal{R})$
    $w \leftarrow a$ if $v{=}A$ else b; $\ell \leftarrow b$ if $v{=}A$ else a
    $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{w\}$; $\mathcal{P}_w \leftarrow \mathcal{P}_w \cup \{\ell\}$
  end for
  if $|\mathcal{S}|$ is odd **then** $\mathcal{S}' \leftarrow \mathcal{S}' \cup \{\mathcal{S}[\text{END}]\}$ ▷ bye
  end if
  $\mathcal{S} \leftarrow \mathcal{S}'$
end while
$W \leftarrow \mathcal{S}[0]$ ▷ Phase-1 champion
$\mathcal{F} \leftarrow \{W\} \cup \mathcal{P}_W$; $|\mathcal{F}| = \lceil \log_2 K \rceil + 1$ ▷ Phase 2: Swiss on path-losers
$\mathbf{s} \leftarrow \textsc{Swiss}(\mathcal{F}, \mathcal{R}, J, t)$ ▷ Swiss subroutine; cost in Appendix D
return $\tau^{(\arg\max_{k \in \mathcal{F}} s_k)}$ ▷ ties broken by Phase-1 wins

CNS together with SEE's contrastive rubric, then issues a relative verdict. Even when no single rubric item fires for either candidate, which is exactly the situation here, the judge can still rank them on the underlying causal question of whether the patch fixes the dtype upcast at the root or only masks the symptom downstream. The two-phase hybrid tournament (Algorithm 1) propagates these per-pair verdicts into a continuous tournament score. On this instance the gold candidate $k{=}10$ wins all of its matches and reaches a normalized score of 1.000, separated from the second-best survivor $k{=}8$ at 0.625 by a wide margin. The lesson is complementary to Case 2: even with a reasonable rubric, pointwise scoring can be too coarse to break ties produced by binary grading. Pairwise comparison reintroduces the continuous semantic signal that the binary rubric throws away.

D Per-instance call counts of COMPARE aggregators

Table 9 tabulates the number of pairwise judge calls $C_{\text{COMPARE}}(K)$ used by each aggregation strategy as a function of pool size K. Pointwise scoring spends one call per candidate. Knockout uses $K{-}1$ calls (each match eliminates one candidate). Vanilla Swiss uses $\lceil \log_2 K \rceil \cdot \lfloor K/2 \rfloor$ calls. Round-robin pairs every candidate with every other for $\binom{K}{2}$ calls. Our hybrid COMPARE (Algorithm 1) is $K{-}1 + R' \cdot \lfloor (R{+}1)/2 \rfloor$ where $R = \lceil \log_2 K \rceil$ and $R' = \max(\lceil \log_2(R{+}1) \rceil, 2)$, scaling as $O(K)$.

Table 9: Pairwise judge calls $C_{\text{COMPARE}}(K)$ for each aggregation strategy as a function of pool size K. Lower is cheaper; the asymptotic order is given on the right. SRC's default COMPARE (Algorithm 1) matches knockout asymptotically while retaining Swiss-style robustness.

Aggregator	$K{=}4$	$K{=}8$	$K{=}16$	$K{=}32$	$K{=}64$	Asymptotic
Pointwise (per-candidate)	4	8	16	32	64	$O(K)$
Knockout	3	7	15	31	63	$O(K)$
Vanilla Swiss	4	12	32	80	192	$O(K \log K)$
SRC COMPARE *(ours)*	5	11	**21**	40	72	$O(K)$
Round-robin	6	28	120	496	2016	$O(K^2)$

E Comparison on cost of different LLM-based verifiers

Table 10 reports the per-instance LLM call counts and measured API spend for SRC and the two rubric-based baselines on SWE-bench Verified at $K{=}16$. "Rubric" is the LLM (and call count) used for rubric induction; "Judge" is the LLM (and total call count per instance) used for downstream

scoring or pairwise comparison. "Avg. cost / inst" is the average API spend per instance. SRC's per-instance cost is slightly lower than the naive Patch Classifier and roughly one-eighth of Agentic Rubrics; the bulk of SRC's budget is the $1+K+C_{\text{COMPARE}}(K) = 1+16+21 = 38$ Gemini 2.5 Flash calls per instance enumerated in Appendix D.

Table 10: Per-instance verifier call counts and average API cost per instance on SWE-bench Verified (K=16). Each verifier is configured as in §4.1. "Calls" is per-instance; "Avg. cost / inst" is the measured API spend per instance.

Method	Rubric model		Judge model		Avg. cost / inst
	LLM	calls	LLM	calls	
Patch Classifier	/	/	Gemini 2.5 Pro	16	$0.084
Agentic Rubrics	Gemini 3 Pro	∼23	Gemini 2.5 Pro	16	$0.540
SRC (ours)	Gemini 2.5 Flash	1	Gemini 2.5 Flash	37	$0.067

F All-Wrong Pools and Abstention

SRC always returns one candidate. This matches the Best-of-K setting used in this paper, where a verifier must pick one of the K candidates. Deployment is different. All K candidates can be wrong, and a forced choice then returns a wrong patch with no warning.

This case is common. On the Qwen3-Coder pool at K=16, 151 of 500 instances have no correct candidate. No selector can resolve these instances, and S-Acc excludes them by construction. Our results therefore do not measure how SRC behaves on them.

For production deployment, SRC can be extended to abstain. COMPARE can return a `both_bad` field alongside its verdict, which marks comparisons where neither candidate appears to meet the rubric. If most comparisons involving the final winner return `both_bad`, SRC can abstain or trigger additional sampling.

G Failure Analysis

We reviewed the 60 of 266 swing instances that SRC gets wrong on the Qwen3-Coder pool at K=16. In 83% of them the selected patch edits the same file as a correct candidate, so SRC usually finds the right location and then picks the wrong edit. The failures fall into three causes. **Thin coverage** (28 cases, 47%): only one or two of the 16 candidates are correct, and they face many plausible wrong patches. **Minimal-diff misfire** (9 cases, 15%): the correct fix is the larger one, sometimes across several files, and SRC picks a compact but incomplete edit. **Hairline call** (23 cases, 38%): the wrong pick is nearly identical in size and location to the many correct candidates, and only 3 of these remain in pools where at least 13 of 16 candidates are correct. In `django-13112`, the only correct candidate fixes the root cause in `ForeignObject.deconstruct()` and includes a large test, while SRC picks a compact edit to the downstream helper `resolve_relation` that fixes a symptom. Each cause suggests a different direction. Thin coverage calls for comparison that protects a lone outlier candidate, and it is also where a targeted execution check is worth its cost. Minimal-diff misfires call for checking whether a patch fully covers the root cause rather than preferring a smaller diff. Hairline calls are best resolved by comparing only the lines where two candidates differ.

H Cross-model selection on SWE-bench Verified

Here we test whether SRC remains effective when the candidate batch mixes attempts from multiple generator models, a setting that mirrors production deployments where several frontier models are queried in parallel and a verifier picks the best output. We construct selection pools of size $K \in \{3, 4, 5\}$ from up to five contemporary frontier models (Claude 4.5 Opus, Gemini 3 Flash, Claude Opus 4.6, MiniMax 2.5, GPT 5.2) on SWE-bench Verified, drawing all rollouts from each model's official public submission to avoid confounding the comparison with our own sampling. Figure 6 shows that across $K \in \{3, 4, 5\}$, SRC sits above every baseline (Random, Self-Consistency,

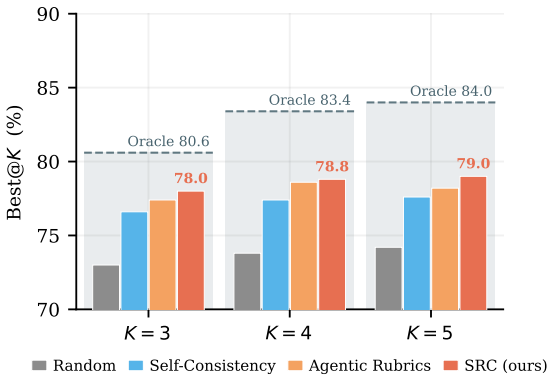


Figure 6: Cross-model Best-of-K selection on SWE-bench Verified.

Agentic Rubrics) under matched candidate pools, and the remaining headroom to Oracle@K ($\sim$80–84%) is the execution-free ceiling rather than a verifier limitation. SRC's sustained margin under that headwind extends the headline claim from same-generator pools to heterogeneous frontier-model pools and positions SRC as a multi-model selector for production routing setups.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes].

   Justification: The abstract and introduction state the paper's scope as training-free, execution-free Best-of-K verification for LLM agents, and the main empirical claims are supported by the method and experiments.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes].

   Justification: The appendix discusses limitations of execution-free selection, LLM-as-judge dependence, and benchmark coverage.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A].

Justification: The paper does not present theoretical theorems or proofs; the complexity statements are algorithmic call-count accounting for the verifier.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes].

Justification: The paper specifies the method, benchmarks, metrics, candidate-pool construction, generator models, judge models, and ablations; the appendix provides the prompt templates.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes].

   Justification: The code will be available at https://github.com/XinyuYee/see-read-compare.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes].

   Justification: The experiments specify the benchmarks, baselines, metrics, candidate-pool sizes, generator settings, judge backbones, and open-weight deployment configuration.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes].

   Justification: Figure 3 reports $\pm 2\sigma$ bands derived from K-subset draws over the fixed candidate pool.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes].

   Justification: The appendix reports per-instance LLM call counts and measured API spend, while the experiments specify the open-weight verifier setting on two RTX 6000 Ada GPUs; the code repository provides runnable configurations.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes].

   Justification: The work uses public benchmarks and agent submissions, releases no private or human-subject data, and is intended for verifier evaluation and reproducibility.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A].

    Justification: There is no societal impact of the work performed.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A].

    Justification: The paper does not release a high-risk dataset or model; the released assets are verifier code, prompts, configurations, and evaluation scripts.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes].

    Justification: The paper cites the benchmark and framework sources, and the code repository documents asset provenance, versions, licenses or terms of use, and redistribution restrictions.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.

- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes].

    Justification: The new released asset is the SRC code and reproducibility package, documented in the code repository; no new model weights or human-subject dataset are released.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A].

    Justification: The paper does not involve crowdsourcing, participant studies, human annotation, or other research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A].

Justification: IRB approval is not applicable because the work does not involve crowdsourcing or human-subject research.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes].

Justification: LLMs are core components of SRC's rubric induction, trajectory summarization, and pairwise judging; the method, experiments, and prompt appendix describe their use.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.