Beyond Binary Verifiable Rewards: Behavioral Self-Rewarding for Software Engineering Agents

Anonymous authors
Paper under double-blind review

Abstract

Reinforcement learning with verifiable rewards (RLVR) provides reliable supervision for software engineering agents through executable tests, but binary rewards fail to distinguish trajectories with substantially different debugging quality. We investigate whether the current policy can judge its own trajectories and use these judgments to improve task performance without a separately trained reward model. To this end, we introduce Behavioral Self-Rewarding (BSR), a reward-refinement method that reuses the policy to compare observable debugging behavior among trajectories from the same task with the same verifier outcome. BSR aggregates pairwise preferences into task-local scores and converts them into bounded auxiliary rewards, introducing reward contrast while preserving the verifier-defined pass/fail ordering. This provides additional supervision from already-collected trajectories without requiring new environment interactions. Experiments on SWE-bench Verified and SWE-bench-Live show consistent improvements across the evaluated Qwen3-14B and Qwen3-32B configurations, supporting using the policy's own comparative judgments as a complementary learning signal for training software engineering agents.

1 Introduction

Large language models are increasingly being deployed as software engineering (SWE) agents: systems that operate inside real repositories and produce patches for concrete issues through interaction with developer tools. Such repository-level software engineering requires an agent to reason under partial information while interacting with a changing repository and deciding how to proceed from the observations it gathers (Yang et al., 2024). This makes software engineering both practically important and technically demanding for language agents, while producing rich multi-step trajectories that capture how an agent approaches a problem. Recent benchmarks and systems have made this setting concrete by evaluating agents on real issue-resolution tasks with executable feedback (Jimenez et al., 2024; Xia et al., 2024; Wang et al., 2025a).

This setting is also a natural fit for reinforcement learning with verifiable rewards (RLVR), where rewards come from automatic checkers rather than learned preference models (Shao et al., 2024; Guo et al., 2025). Recent work extends RLVR from single-turn reasoning to multi-step agentic RL, where models learn from trajectories of actions and observations rather than isolated final answers (Chen et al., 2025a; Wang et al., 2025b; Luo et al., 2025). Software engineering is a compelling instance because interactions are long-horizon and behaviorally complex, while final patches can often be validated through repository test suites (Pan et al., 2024; Wei et al., 2026; Jain et al., 2025; Golubev et al., 2025).

However, binary test outcomes compress rich interaction trajectories into a single success or failure signal. Failed trajectories can vary substantially in debugging quality: some locate relevant code, gather useful evidence, or make progress toward a fix, while others achieve little, yet both receive reward zero. Under group-relative objectives such as GRPO or leave-one-out PPO, rollout groups with identical verifier outcomes have zero reward variance and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

therefore zero outcome-derived relative advantages (Shao et al., 2024; Chen et al., 2025a). Useful distinctions among already-collected trajectories consequently remain unavailable to the reward signal. This is particularly costly for SWE agents, whose rollouts require repeated model inference and tool execution. Several recent approaches enrich verifier supervision with dense reward-model scores and learned trajectory evaluations (Tao et al., 2026; Han et al., 2026; Shum et al., 2026). These approaches demonstrate the value of supervision beyond binary outcomes, but obtain the additional signal from a separate reward model. Rather than developing another trajectory scorer, we investigate a complementary question: *can the current policy itself judge its own SWE trajectories, and how can those judgments be used to improve its task performance?*

Self-rewarding and LLM-as-a-judge methods motivate this question by showing that language models can provide evaluative signals for generated responses and individual reasoning steps (Yuan et al., 2024; Zheng et al., 2023; Liu et al., 2023; Zhang et al., 2025). For interactive SWE agents, the object of judgment is instead a long-horizon trajectory of search, inspection, editing, and verification. These trajectories record not only the final patch but also the evidence gathered, the actions taken in response, and the checks performed along the way. The challenge is to turn judgments about this interaction experience into useful training signals while retaining the distinction between passing and failing attempts. We therefore study a restricted judging role: the executable verifier determines the outcome of each attempt, while the current policy compares attempts at the same task that share that outcome. This asks the policy for local preferences, rather than an independent correctness decision or a globally calibrated trajectory-quality score.

To investigate this question, we introduce *Behavioral Self-Rewarding* (BSR), a reward-refinement method for multi-step agentic RLVR. For each software-engineering task, the policy samples multiple trajectories and receives the ordinary test-based reward. BSR partitions these trajectories into task-local verifier-outcome buckets and reuses the current policy to perform pairwise comparisons within each bucket. The comparisons follow a rubric focused on observable debugging actions and their outcomes rather than writing style or confidence. Each judgment identifies a preferred trajectory and a coarse preference-strength label, which are aggregated into task-local ranking scores. These scores are normalized and transformed into centered, bounded auxiliary rewards that supplement the verifier reward. The reward bounds ensure that every verifier-passing trajectory retains a higher reward than every verifier-failing trajectory. BSR thus combines current-policy trajectory comparison with verifier-conditioned reward refinement, using the policy's preferences directly in on-policy RL without fitting a separate trajectory reward model.

BSR integrates with group-relative RL by introducing reward contrast where binary verification provides none. In all-fail rollout groups, nonconstant auxiliary rewards can restore group-relative learning signal; in mixed-outcome groups, they provide finer distinctions among trajectories sharing an outcome. Reward refinement requires additional model inference but no new environment interactions. We evaluate BSR against verifier-only RL to test whether this self-judgment procedure improves the acting policy. Experiments cover Qwen3-14B and Qwen3-32B on SWE-bench Verified, SWE-bench-Live Lite, and Terminal-Bench 2.1. BSR improves average accuracy across the evaluated configurations on all three benchmarks, with consistent gains in Pass@3 on the two SWE benchmarks. Ablations examine the reward mapping and the choice of refined outcome buckets; failure-only refinement achieves the highest average accuracy and Pass@3 among the tested bucket configurations. In the random-reward control, using the same auxiliary reward amplitude range without trajectory rankings does not reproduce these gains. Together, these results support using the current policy's comparative judgments as an additional source of supervision for learning from repository-level interaction experience.

2 Preliminaries

We consider software-engineering tasks under RLVR: A task x consists of an initial repository state, an issue description, and an executable verifier. A policy π_θ interacts with the repository through developer tools to gather evidence, modify code, and produce a final

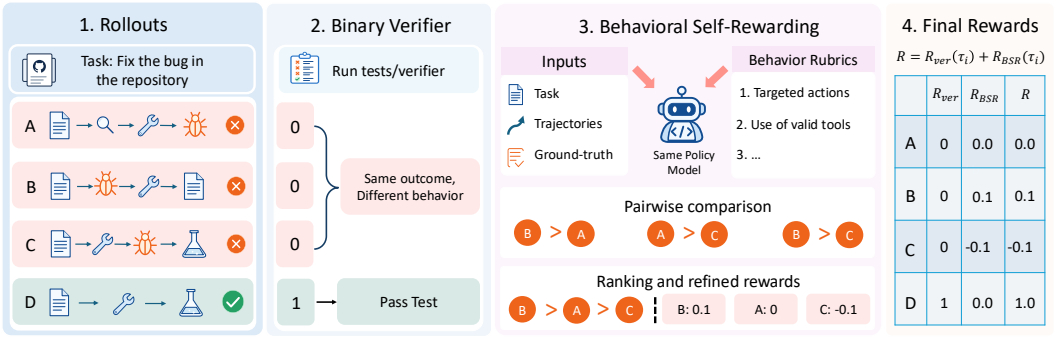


Figure 1: Overview of Behavioral Self-Rewarding (BSR). Binary rewards provide no contrast when trajectories share the same verifier outcome, while BSR ranks them within the outcome group to recover a bounded learning signal.

patch. For each task, we sample $K \geq 2$ trajectories,

$$\{\tau_i\}_{i=1}^{K} \sim \pi_\theta(\cdot \mid x), \tag{1}$$

where τ_i records the full multi-step interaction with the environment. The verifier assigns a binary reward,

$$R_{\text{ver}}(\tau_i) = \begin{cases} 1, & \text{if the final patch passes the verifier,} \\ 0, & \text{otherwise.} \end{cases} \tag{2}$$

In verifier-only RL, the training reward is $R_i = R_{\text{ver}}(\tau_i)$.

Group Relative Policy Optimization (GRPO) computes advantages by normalizing rewards within each sampled group (Shao et al., 2024):

$$A_i^{\text{GRPO}} = \frac{R_i - \mu_R}{\sigma_R + \epsilon}, \qquad \mu_R = \frac{1}{K}\sum_{j=1}^{K} R_j, \qquad \sigma_R = \sqrt{\frac{1}{K}\sum_{j=1}^{K}(R_j - \mu_R)^2}, \tag{3}$$

where $\epsilon > 0$ ensures numerical stability. LOOP adapts group-relative optimization to long-horizon interactive agents using a leave-one-out baseline (Chen et al., 2025a):

$$A_i^{\text{LOO}} = R_i - \frac{1}{K-1}\sum_{j \neq i} R_j. \tag{4}$$

Both methods use these advantages in clipped policy-gradient objectives without a learned value model. With binary verifier rewards, all-fail and all-pass rollout groups have identical rewards and therefore zero advantages under both formulations. Such groups provide no reward-derived policy-gradient signal, even when their trajectories differ substantially in debugging quality. This motivates refining rewards among trajectories with the same verifier outcome while retaining verifier-defined task success.

3 Method

We introduce *Behavioral Self-Rewarding* (BSR) to study whether the current policy's judgments can provide useful supervision for its own software-engineering training. For each task, we first collect policy trajectories and their executable verifier outcomes, then reuse the same model as a post-rollout judge to compare attempts with the same outcome. The judge receives the task, shared verifier outcome, compact trajectory representations, and the training task's reference patch $p^\star$. This deliberately asymmetric setup gives the post-rollout judge a solution reference for interpreting the relevance of observed actions and edits, while the acting policy generates trajectories without access to $p^\star$ during both training and evaluation. The reference is intended to reduce ambiguity in task-local comparisons; it is

Algorithm 1 Behavioral Self-Rewarding with Pairwise Ranking

Require: Task x, reference patch $p^\star$, policy π_θ, group size K, verifier V
Require: Amplitudes $a_y \geq 0$ satisfying $a_0 + a_1 < 1$, deadzone thresholds $\delta_y \geq 0$, softness parameters $\tau_y > 0$ for $y \in \{0, 1\}$

```
Sample trajectories {τ_i}_{i=1}^K ~ π_θ(· | x)
for i = 1, ..., K do
    y_i ← V(τ_i)                          ▷ y_i = 1 if the trajectory passes the verifier; otherwise y_i = 0
    h_i ← SummarizeTrajectory(τ_i)
end for
for y ∈ {0, 1} do
    B_y ← {i : y_i = y}                   ▷ Bucket trajectories by verifier outcome
    if a_y = 0 or |B_y| < 2 then
        r_i ← 0 for all i ∈ B_y
        continue
    end if
    C_y ← PairwiseCompare(x, p*, {h_i : i ∈ B_y}, B_y, y)          ▷ See Algorithm 3
    {r_i : i ∈ B_y} ← PairwiseRewardMap(C_y, B_y, a_y, δ_y, τ_y)   ▷ See Algorithm 2
end for
for i = 1, ..., K do
    R_i ← y_i + r_i                       ▷ Add bounded self-reward to verifier reward
end for
return refined rewards {R_i}_{i=1}^K
```

provided as context for the behavioral rubric rather than used to compute an explicit patch-similarity reward. The resulting preferences are converted into auxiliary rewards for policy optimization without training a separate reward model. The refined training reward is

$$R_i = R_{\text{ver}}(\tau_i) + r_i, \tag{5}$$

where r_i is a bounded auxiliary reward derived from the policy's within-outcome comparisons. We instantiate this judgment through pairwise ranking, asking the model to express local preferences rather than assign absolute trajectory-quality scores. Each comparison concerns attempts at the same task and can be inspected directly, without requiring a quality scale calibrated across tasks. Alternative instantiations based on joint group rankings and absolute rubric scores are discussed in Appendix A.4.

3.1 Pairwise Behavioral Ranking

BSR refines the rewards used by group-relative RL. For each task x, the current policy samples K trajectories, and the verifier assigns each trajectory a binary outcome:

$$y_i = R_{\text{ver}}(\tau_i) \in \{0, 1\}. \tag{6}$$

We partition the trajectories into task-local verifier-outcome buckets,

$$B_y = \{i \in \{1, \ldots, K\} : y_i = y\}, \qquad y \in \{0, 1\}. \tag{7}$$

Each outcome bucket has a reward amplitude $a_y \geq 0$, with $a_y = 0$ disabling refinement for that outcome. The judge compares only trajectories from the same task and verifier-outcome bucket. A failure bucket may contain all K trajectories from an all-fail rollout group or only the failed subset of a mixed-outcome group; success buckets are handled analogously. This restricts the policy's judging role to expressing preferences among attempts with a shared outcome. The distinction between passing and failing attempts is preserved by the reward-amplitude constraint in Equation 15.

Outcome-separated rewarding. The judging criterion depends on the shared verifier outcome. Among failed trajectories, the policy is asked to identify which attempt shows more useful progress toward eventual success. Relevant evidence includes whether the agent searched the appropriate part of the repository, gathered useful evidence, made directionally plausible edits, and failed in a recoverable way. Among successful trajectories, the policy instead compares how the task was solved: whether the agent acted efficiently, produced a targeted fix, avoided brittle or unnecessary changes, and verified the result. Separate rubrics

distinguish progress among unsuccessful attempts from the quality of successful execution, rather than applying the same criterion to both outcomes.

Trajectory summarization. Raw interaction trajectories can exceed the judge's context length. We construct compact representations by removing the thinking section from each step, retaining exact actions, and truncating observations beyond a token threshold. This presents the judge with the observable actions and their outcomes. We also include deterministic trajectory statistics, such as command executions before and after the final edit, as supporting evidence for the comparison. The full representation and summarization procedure are provided in Appendix A.3.

Weighted pairwise comparisons. For each scheduled pair (i, j) within a bucket, the current policy returns a preferred trajectory $w_{ij} \in \{i, j\}$ and a coarse preference-strength label $\alpha_{ij} \in \{1, 2, 3\}$, indicating slight, clear, or strong preference. The strength label serves as a weight when fitting the task-local scores, retaining coarse information about the relative preference without requiring fine-grained scalar quality judgments. To reduce positional artifacts, we randomize the order in which the two trajectories are presented.

3.2 From Pairwise Comparisons to Rewards

For each outcome bucket B_y, let $\mathcal{C}_y$ denote the scheduled pairwise comparisons, each represented by $(i, j, w_{ij}, \alpha_{ij})$. We aggregate these weighted preferences into task-local scores s_i using a regularized weighted Bradley–Terry model (Bradley and Terry, 1952), described in Appendix A.2. These scores summarize the policy's comparative judgments within the bucket. To use them as auxiliary rewards, we normalize the scores, apply a soft deadzone, and center and bound the resulting values.

Task-local score normalization. The fitted scores describe relative preferences within a task and outcome bucket rather than a shared quality scale across tasks. For buckets with at least two trajectories, we standardize the scores as

$$z_i = \frac{s_i - \mu_y}{\sigma_y + \epsilon}, \qquad \mu_y = \frac{1}{|B_y|} \sum_{j \in B_y} s_j, \qquad \sigma_y^2 = \frac{1}{|B_y| - 1} \sum_{j \in B_y} (s_j - \mu_y)^2, \tag{8}$$

where $\epsilon > 0$ is a small constant for numerical stability. Buckets with fewer than two trajectories receive zero auxiliary reward because no pairwise comparison is possible. This normalization makes the subsequent reward map depend on relative position within the current bucket rather than the raw scale of the fitted scores.

Soft deadzone. We apply a soft deadzone to suppress normalized scores near zero and smoothly transform larger deviations from the bucket mean:

$$u_i = \operatorname{sign}(z_i) \tanh\left(\frac{\max(|z_i| - \delta_y, 0)}{\tau_y} \right). \tag{9}$$

The threshold δ_y determines the region around zero in which scores are suppressed, following the use of thresholding to remove small deviations (Drucker et al., 1996; Donoho, 1995). Trajectories with $|z_i| \leq \delta_y$ have $u_i = 0$. Outside this region, the softness parameter τ_y controls how quickly $|u_i|$ approaches one.

Centering and bounding. Let

$$\mathcal{A}_y = \{i \in B_y : |z_i| > \delta_y\} \tag{10}$$

denote the active set of trajectories outside the deadzone. If $\mathcal{A}_y$ is empty, we set $q_i = 0$ for every trajectory in the bucket. Otherwise, we center the active scores:

$$\bar{u}_y = \frac{1}{|\mathcal{A}_y|} \sum_{j \in \mathcal{A}_y} u_j, \qquad \tilde{u}_i = u_i - \bar{u}_y, \quad i \in \mathcal{A}_y, \tag{11}$$

and rescale them by their maximum absolute value within the active set:

$$q_i = \frac{\tilde{u}_i}{\max_{j \in \mathcal{A}_y} |\tilde{u}_j| + \epsilon}, \quad i \in \mathcal{A}_y. \tag{12}$$

Algorithm 2 PairwiseRewardMap

Require: Comparisons $\mathcal{C}_y$, bucket B_y, amplitude a_y, deadzone δ_y, softness τ_y
$\{s_i : i \in B_y\} \leftarrow \text{WeightedBTLFit}(\mathcal{C}_y, B_y)$ ▷ Aggregate pairwise judgments
Compute $\{z_i : i \in B_y\}$ using Equation 8 ▷ Normalize within the bucket
for $i \in B_y$ **do**
  $u_i \leftarrow \text{sign}(z_i) \tanh(\max(|z_i| - \delta_y, 0)/\tau_y)$ ▷ Apply deadzone and soft mapping
end for
$\mathcal{A}_y \leftarrow \{i \in B_y : |z_i| > \delta_y\}$ ▷ Select trajectories outside the deadzone
if $\mathcal{A}_y = \emptyset$ **then**
  return $r_i = 0$ for all $i \in B_y$ ▷ No scores outside the deadzone
end if
Center active scores using Equation 11 ▷ Make the signal contrastive
Normalize active scores using Equation 12 ▷ Bound q_i to $[-1, 1]$
return $r_i = a_y q_i$ for all $i \in B_y$ ▷ Inactive trajectories have $q_i = 0$

For inactive trajectories $i \notin \mathcal{A}_y$, we set $q_i = 0$. The resulting scores satisfy $q_i \in [-1, 1]$. Centering makes the auxiliary signal contrastive within the active subset, while the rescaling bounds its magnitude before applying the outcome-specific reward amplitude.

Final reward. For trajectory $i \in B_y$, the auxiliary self-judgment reward is

$$r_i = a_y q_i, \tag{13}$$

where $a_y \geq 0$ controls the amplitude for outcome bucket y. Adding this refinement to the verifier reward gives

$$R_i = R_{\text{ver}}(\tau_i) + r_i = \begin{cases} 1 + a_1 q_i, & R_{\text{ver}}(\tau_i) = 1, \\ a_0 q_i, & R_{\text{ver}}(\tau_i) = 0. \end{cases} \tag{14}$$

Since $q_i \in [-1, 1]$, passing trajectories receive rewards of at least $1 - a_1$, whereas failing trajectories receive rewards of at most a_0. We choose nonnegative amplitudes satisfying

$$a_0 + a_1 < 1, \tag{15}$$

which guarantees that every verifier-passing trajectory receives a higher reward than every verifier-failing trajectory. The refined rewards are then used in group-relative policy optimization. In all-fail rollout groups, nonconstant auxiliary rewards can restore group-relative learning signal; in mixed-outcome groups, they provide additional distinctions within each refined outcome bucket.

4 Experiments

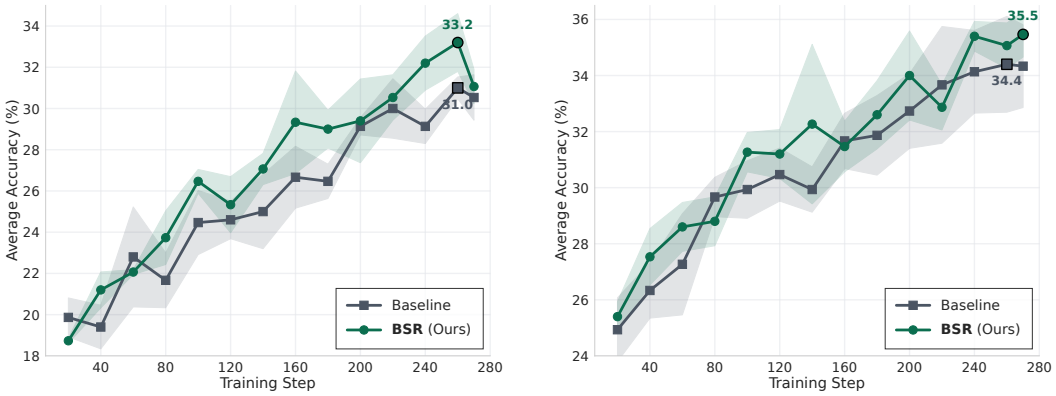
Setup. We use RLLM (Tan et al., 2025) to train on a filtered version of R2E-Gym (Jain et al., 2025) and evaluate on SWE-bench Verified (Jimenez et al., 2024) and SWE-bench-Live Lite (Zhang et al., 2026). Training rollouts use a maximum of 40 agent steps; evaluation uses 50 steps, temperature 1, and three evaluation runs per checkpoint. We report average accuracy across these runs and Pass@3, the fraction of tasks solved by at least one of the three sampled trajectories. In the main BSR setting, we refine only the failure bucket, with $a_0 = 0.15$, $a_1 = 0$, $\delta_0 = 0.5$, and $\tau_0 = 0.5$. This configuration performs similarly to or better than refining both outcome buckets while requiring fewer pairwise comparisons (Section 4.3). The subsequent ablations and trajectory-behavior analysis use SWE-bench Verified. Complete training settings are provided in Appendix B.1.

4.1 Main Results

Table 1 compares BSR with verifier-only RL across two model sizes, predefined training-step budgets, and two SWE benchmarks. BSR improves both average accuracy and Pass@3 in every evaluated setting on SWE-bench Verified and SWE-bench-Live. For example, on SWE-bench Verified, Qwen3-14B improves from 32.1% to 34.7% average accuracy under the 400-step budget. On SWE-bench-Live, the same setting improves average accuracy

Table 1: Performance (%) on SWE-bench Verified and SWE-bench-Live Lite. Standard deviations are across three evaluation runs.

			SWE-bench Verified		SWE-bench-Live Lite	
Size	Budget	Method	Avg. Acc. ± Std.	Pass@3	Avg. Acc. ± Std.	Pass@3
14B	270	Baseline	31.0 ± 0.5	45.0	7.11 ± 0.19	13.33
		BSR (Ours)	$\mathbf{33.2 \pm 1.4}$	**46.6**	$\mathbf{8.67 \pm 0.88}$	**15.33**
	400	Baseline	32.1 ± 1.1	46.6	6.33 ± 0.34	11.33
		BSR (Ours)	$\mathbf{34.7 \pm 1.0}$	**48.0**	$\mathbf{8.11 \pm 1.02}$	**13.67**
32B	270	Baseline	34.4 ± 1.7	48.8	7.67 ± 0.88	12.33
		BSR (Ours)	$\mathbf{35.5 \pm 0.8}$	**50.6**	$\mathbf{8.56 \pm 0.51}$	**13.67**



(a) Qwen3-14B (b) Qwen3-32B

Figure 2: SWE-bench Verified evaluation accuracy over training for Qwen3-14B and Qwen3-32B. Each point averages three evaluation runs.

from 6.33% to 8.11%. The gains hold across Qwen3-14B and Qwen3-32B and across both training budgets, indicating that the improvement is not specific to a single model scale or training duration. Figure 2 shows the corresponding SWE-bench Verified learning curves. We additionally evaluate on Terminal-Bench 2.1 (Merrill et al., 2026). BSR improves average accuracy across all three evaluated model and training-budget settings, full results are provided in Appendix B.2.

Statistic	Baseline	BSR (Ours)
Process metrics		
Related file viewed before first edit ↑	0.733	**0.748**
Search-to-edit conversion ↑	0.465	**0.494**
Post-edit test rate ↑	0.301	**0.311**
Duplicate command rate ↓	0.209	**0.208**
Selected workload statistics		
Average number of errors ↓	13.38	**12.76**
Average number of searches	4.74	4.06
Average number of tests	0.747	0.797
Average total time (s) ↓	1132.28	**1068.55**

Table 2: Trajectory behavior statistics over the full set of trajectories. Arrows indicate the preferred direction for metrics where applicable. Details are in Appendix B.4.

Behavioral changes. Table 2 compares investigation, editing, and verification behavior. BSR more often views a related file to the issue before the first edit (0.748 versus 0.733),

Refined bucket(s)	Avg.	Pass@3
Baseline	31.0	45.0
Pass bucket only	32.0	44.0
Pass + fail buckets	33.1	44.6
Fail bucket only **(BSR)**	**33.2**	**46.6**

Table 4: Ablation of the verifier-outcome buckets receiving BSR (%).

Method	Avg. acc.	Pass@3
Baseline	31.0	45.0
Random reward	30.9	42.8
BSR **(Ours)**	**33.2**	**46.6**

Table 5: Random-reward ablation on Qwen3-14B. Random rewards use the same amplitude range as BSR but ignore behavioral rankings.

achieves higher search-to-edit conversion (0.494 versus 0.465), and performs more post-edit testing (0.311 versus 0.301). These changes coincide with fewer searches and recorded errors, alongside a higher average number of tests. Overall, the aggregate pattern is consistent with stronger reference-oriented investigation, search follow-through, and verification. Full trajectory statistics and metric definitions are provided in Appendix B.4.

4.2 Ablation: Deadzone and Softness

We examine how the deadzone and softness affect the conversion of pairwise rankings into auxiliary rewards. The deadzone suppresses near-zero normalized scores before centering, while softness controls how deviations outside the deadzone are transformed. Table 3 compares the full mapping with variants that remove both components or retain the deadzone without softness. Removing both yields 32.0% average accuracy and 45.6% Pass@3. Adding the deadzone alone raises average accuracy to 32.6%, but lowers Pass@3 to 44.0%. The full BSR mapping achieves 33.2% average accuracy, improving both metrics over either simplified variant. Thus, the deadzone alone does not improve every metric, whereas combining it with softness gives the strongest results among the tested mappings. Full training curves are provided in Figure 7 in appendix.

Reward mapping	Avg.	Pass@3
No deadzone, no softness	32.0	45.6
Deadzone, no softness	32.6	44.0
Deadzone + softness **(BSR)**	**33.2**	**46.6**

Table 3: Reward-mapping ablation on 14B.

4.3 Ablation: Verifier-Outcome Buckets

We compare applying BSR to the failure bucket, the success bucket, or both. The failure bucket includes failed trajectories from both all-fail and mixed-outcome rollout groups. Table 4 shows that failure-only refinement achieves the highest average accuracy and Pass@3, at 33.2% and 46.6%, respectively. Refining both buckets achieves similar average accuracy of 33.1%, but lower Pass@3 of 44.6%, while requiring additional comparisons among successful trajectories. Success-only refinement reaches 32.0% average accuracy and 44.0% Pass@3: it improves average accuracy over the verifier-only baseline, but does not improve coverage across three sampled solutions. These results favor failure-only refinement in this setting. Extending refinement to successful trajectories increases judgment workload without improving either reported metric over failure-only refinement. Full training curves appear in Figure 6 in the appendix.

4.4 Additional Study: Random-reward control

We test whether arbitrary auxiliary rewards can reproduce BSR's gains without behavioral rankings. The control assigns random rewards within the same amplitude range as BSR, but does not associate them with pairwise trajectory preferences. Table 5 shows that the control achieves 30.9% average accuracy and 42.8% Pass@3, compared with 31.0% and 45.0% for verifier-only RL. BSR instead reaches 33.2% average accuracy and 46.6% Pass@3. Random rewards therefore fail to improve average accuracy and reduce Pass@3, whereas BSR improves both. This shows that matching the reward amplitude alone is insufficient to reproduce the gains from BSR's ranking-based refinement.

5 Related Work

Verifiable rewards for agentic RL. RLVR uses automatic correctness checks, with DeepSeek-R1 demonstrating its effectiveness at scale (Guo et al., 2025). Building on PPO (Schulman et al., 2017), GRPO removes the learned critic and normalizes within-group rewards, GSPO uses sequence-level optimization, and LOOP applies a leave-one-out baseline to long-horizon interactive agents (Shao et al., 2024; Zheng et al., 2025; Chen et al., 2025a). Hybrid-reward methods study how additional evaluative signals can complement binary verification. HERO combines verifier outcomes with reward-model scores through outcome-conditioned normalization (Tao et al., 2026), while PAPO integrates rubric-based process feedback through decoupled advantage normalization (Tan et al., 2026). BSR focuses on how the additional signal is obtained: we reuse the current policy to produce task-local trajectory preferences, then incorporate these preferences into its own training through bounded reward refinement.

Software engineering agents, environments, and training. SWE-bench and SWE-bench-Live benchmark real-world issue resolution (Jimenez et al., 2024; Zhang et al., 2026). SWE-agent, Agentless, and related work explore systems and workflows for repository-level issue resolution (Yang et al., 2024; Zhang et al., 2024; Xia et al., 2024; Wang et al., 2025a; Bairi et al., 2024; Chen et al., 2025b). SWE-Gym and R2E-Gym provide executable repository-level training environments (Pan et al., 2024; Jain et al., 2025). Training approaches include SWE-RL on software evolution data (Wei et al., 2026), multi-turn RL in stateful repositories (Golubev et al., 2025), self-play bug injection and repair (Wei et al., 2025), and CWM's combination of interaction traces and multi-task RL (Copet et al., 2025). Methods like SWE-RM (Shum et al., 2026) and SWE-TRACE (Han et al., 2026) develop dedicated reward models for SWE trajectories and use them during both training and test-time inference. BSR instead investigates whether the acting policy itself can supply useful feedback for its own on-policy learning. Its judgments are obtained directly from the current policy through within-task, within-outcome comparisons, without a separate reward-model training stage. Our experiments evaluate whether adding this self-comparison procedure improves verifier-only training in a fixed SWE-agent setup.

Process supervision and trajectory preferences. Process supervision evaluates intermediate reasoning (Lightman et al., 2024), while trajectory preferences support reward learning (Christiano et al., 2017). Bradley–Terry aggregation and RankNet provide pairwise-ranking foundations (Bradley and Terry, 1952; Burges et al., 2005), and DPO learns directly from preference pairs (Rafailov et al., 2023). LLM-as-a-judge methods provide model-generated evaluations (Zheng et al., 2023; Liu et al., 2023). Self-Rewarding Language Models use the model's own response evaluations to construct preferences for iterative DPO (Yuan et al., 2024). Process-based Self-Rewarding applies the model's pairwise judgments to individual mathematical reasoning steps and uses step-wise preference optimization (Zhang et al., 2025). BSR builds on self-generated feedback but studies a different judgment unit and learning setting: observable debugging behavior across complete tool-interaction trajectories, compared within the same task and verifier-outcome bucket. Its preferences are aggregated into bounded auxiliary rewards for on-policy RL, rather than used as response- or step-level preference-optimization targets.

6 Conclusion

We introduced Behavioral Self-Rewarding (BSR) to investigate whether the current policy can use its own comparative judgments to improve software-engineering performance. BSR compares trajectories from the same task and verifier-outcome bucket, converting task-local preferences into bounded auxiliary rewards while preserving the verifier-defined pass/fail ordering. By reusing the policy as a post-rollout judge, it provides additional supervision from collected trajectories without training a separate reward model. Experiments on SWE-bench Verified and SWE-bench-Live Lite show improvements over verifier-only RL across the evaluated Qwen3-14B and Qwen3-32B configurations. Together, these results support the policy's own comparative judgments as a complement to binary verification, allowing distinctions among otherwise reward-equivalent trajectories to contribute to learning from repository-level interaction experience.

LLM Usage Disclosure

We used large language models to assist with drafting initial versions of selected sections, improving language and readability, and retrieving and discovering potentially relevant related work. All identified literature was independently reviewed and verified by the authors. The authors take full responsibility for the final content of the paper.

References

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D C, Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, Balasubramanyan Ashok, and Shashank Shet. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE):675–698, 2024.

Ralph Allan Bradley and Milton E Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. *Biometrika*, 39(3/4):324–345, 1952.

Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In *Proceedings of the 22nd international conference on Machine learning*, pages 89–96, 2005.

Kevin Chen, Marco Cusumano-Towner, Brody Huval, Aleksei Petrenko, Jackson Hamburger, Vladlen Koltun, and Philipp Krähenbühl. Reinforcement learning for long-horizon interactive llm agents. *arXiv preprint arXiv:2502.01600*, 2025a.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. Locagent: Graph-guided llm agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727, 2025b.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

David L Donoho. De-noising by soft-thresholding. *IEEE transactions on information theory*, 41(3):613–627, 1995.

Harris Drucker, Christopher J Burges, Linda Kaufman, Alex Smola, and Vladimir Vapnik. Support vector regression machines. *Advances in neural information processing systems*, 9, 1996.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, et al. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Hao Han, Jin Xie, Xuehao Ma, Weiquan Zhu, Ziyao Zhang, ZhiLiang Long, Hongkai Chen, and Qingwen Ye. Swe-trace: Optimizing long-horizon swe agents through rubric process reward models and heuristic test-time scaling. *arXiv preprint arXiv:2604.14820*, 2026.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2egym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.

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 *International Conference on Learning Representations*, volume 2024, pages 39578–39601, 2024.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-eval: Nlg evaluation using gpt-4 with better human alignment. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 2511–2522, 2023.

Xufang Luo, Yuge Zhang, Zhiyuan He, Zilong Wang, Siyun Zhao, Dongsheng Li, Luna K Qiu, and Yuqing Yang. Agent lightning: Train any ai agents with reinforcement learning. *arXiv preprint arXiv:2508.03680*, 2025.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations*, volume 2026, pages 40903–40986, 2026.

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.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems*, 36:53728–53741, 2023.

Pascal Sauer, Ágnes Cseh, and Pascal Lenzner. Improving ranking quality and fairness in swiss-system chess tournaments. *arXiv preprint arXiv:2112.10522*, 2021.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, Jiaxi Yang, Yuzhen Huang, Junyang Lin, Junxian He, et al. Swe-rm: Execution-free feedback for software engineering agents. In *International Conference on Learning Representations*, volume 2026, pages 143426–143446, 2026.

Sijun Tan, Michael Luo, Colin Cai, Tarun Venkat, Kyle Montgomery, Aaron Hao, Tianhao Wu, Arnav Balyan, Manan Roongta, Chenguang Wang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. rllm: A framework for post-training language agents. `https://pretty-radio-b75.notion.site/rLLM-A-Framework-for-Post-Training-Language-Agents-21b81902c146819db63cd98a54ba5f31`, 2025. Notion Blog.

Zelin Tan, Zhouliang Yu, Bohan Lin, Zijie Geng, Hejia Geng, Yudong Zhang, Mulei Zhang, Yang Chen, Shuyue Hu, Zhenfei Yin, et al. Papo: Stabilizing rubric integration training via decoupled advantage normalization. *arXiv preprint arXiv:2603.26535*, 2026.

Leitian Tao, Ilia Kulikov, Swarnadeep Saha, Tianlu Wang, Jing Xu, Sharon Li, Jason E Weston, and Ping Yu. Hybrid reinforcement: when reward is sparse, better to be dense. In *International Conference on Learning Representations*, volume 2026, pages 136479–136500, 2026.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Ziwei Chai, Y Charles, HS Che, Cheng Chen, et al. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

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. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919, 2025a.

Zihan Wang, Kangrui Wang, Qineng Wang, Pingyue Zhang, Linjie Li, Zhengyuan Yang, Xing Jin, Kefan Yu, Minh Nhat Nguyen, Licheng Liu, et al. Ragen: Understanding self-evolution in llm agents via multi-turn reinforcement learning. *arXiv preprint arXiv:2504.20073*, 2025b.

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

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525, 2026.

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

John Yang, Carlos 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.

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Xian Li, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*, 2024.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! *Advances in Neural Information Processing Systems*, 38, 2026.

Shimao Zhang, Xiao Liu, Xin Zhang, Junxiao Liu, Zheheng Luo, Shujian Huang, and Yeyun Gong. Process-based self-rewarding language models. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 18097–18110, 2025.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, et al. Group sequence policy optimization. arxiv. *arXiv preprint arXiv:2507.18071*, 2025.

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.

A Additional Method Details

A.1 Pairwise Comparison Scheduling.

Algorithm 3 describes the comparison schedule for each verifier-outcome bucket B_y, where $y \in \{0, 1\}$ denotes the shared verifier outcome. For $2 \leq |B_y| < 5$, we use exhaustive pairwise comparison. For larger buckets, we use a Swiss-style tournament schedule (Sauer et al., 2021) to reduce the number of model calls while maintaining balanced comparison coverage. Table 6 shows the resulting reduction in model comparisons.

For each scheduled pair, we ask the model to compare the two trajectory summaries given the task x, reference patch $p^\star$, and bucket outcome y. The model selects the better trajectory under the shared outcome: better progress among failures or cleaner behavior among successes. We denote the collection of performed comparisons by $\mathcal{C}_y$, where each comparison record $(i, j, w_{ij}, \alpha_{ij})$ contains the compared pair (i, j), the preferred trajectory w_{ij}, and the preference-strength label α_{ij}. To reduce position bias during BSR training, we randomize the presentation order of each pairwise comparison.

We use separate prompts for the pass and fail buckets (Figures 9 and 10). They have the same general structure but direct the model to different behavioral criteria depending on the verifier outcome. We show the user prompt in Figure 11 and an example ranking response in Figure 12.

Algorithm 3 PairwiseCompare: Comparison Scheduling for One Outcome Bucket

Require: Task x, reference patch $p^\star$, model π_θ, trajectory summaries $\{h_i : i \in B_y\}$, bucket B_y, outcome $y \in \{0, 1\}$

$n \leftarrow |B_y|$
$\mathcal{C}_y \leftarrow \emptyset$
if $n < 2$ **then**
  return $\mathcal{C}_y$ ▷ No pairwise comparison is possible
else if $n < 5$ **then** ▷ **Case 1: Exhaustive comparison**
  $\mathcal{P} \leftarrow \{(i, j) : i, j \in B_y,\ i < j\}$
  for $(i, j) \in \mathcal{P}$ **do**
    $(w_{ij}, \alpha_{ij}) \leftarrow \text{ModelCompare}_{\pi_\theta}(x, p^\star, h_i, h_j, y)$
    Append $(i, j, w_{ij}, \alpha_{ij})$ to $\mathcal{C}_y$
  end for
else ▷ **Case 2: Swiss-style comparison**
  $T \leftarrow \min(\lceil \log_2 n \rceil + 2,\ n - 1)$ ▷ Number of rounds
  Initialize wins $v_i \leftarrow 0$ and match counts $c_i \leftarrow 0$ for all $i \in B_y$
  Initialize pair counts $c_{ij} \leftarrow 0$ for all $i, j \in B_y$
  for $t = 1, \ldots, T$ **do**
    Sort trajectories by decreasing v_i, then increasing c_i
    $\mathcal{P}_t \leftarrow \emptyset$
    Form round pairs greedily among unused trajectories ▷ Prefer unseen pairs, fewer rematches, and similar win counts
    Update c_{ij} for each scheduled pair $(i, j) \in \mathcal{P}_t$
    for $(i, j) \in \mathcal{P}_t$ **do**
      $(w_{ij}, \alpha_{ij}) \leftarrow \text{ModelCompare}_{\pi_\theta}(x, p^\star, h_i, h_j, y)$
      Append $(i, j, w_{ij}, \alpha_{ij})$ to $\mathcal{C}_y$
      $v_{w_{ij}} \leftarrow v_{w_{ij}} + 1$ ▷ Update Swiss score
      $c_i \leftarrow c_i + 1, \quad c_j \leftarrow c_j + 1$ ▷ Update match counts
    end for
  end for
end if
return $\mathcal{C}_y$

A.2 Weighted Bradley–Terry Score Fitting

Weighted Bradley–Terry aggregation. Given the performed comparisons $\mathcal{C}_y$ for an outcome bucket B_y, we convert pairwise preferences into task-local scalar scores using a weighted Bradley–Terry model (Bradley and Terry, 1952). Each trajectory $i \in B_y$ is assigned

Bucket size n	Full comparisons $\binom{n}{2}$	Scheduled comparisons	Reduction
1	0	0	–
2	1	1	0.0%
3	3	3	0.0%
4	6	6	0.0%
5	10	8	20.0%
6	15	15	0.0%
7	21	15	28.6%
8	28	20	28.6%

Table 6: Comparison reduction from the scheduling scheme. For $n < 5$, the scheduler uses exhaustive pairwise comparison. For $n \geq 5$, it uses $T = \min(\lceil \log_2 n \rceil + 2, n - 1)$ Swiss-style rounds, with at most $\lfloor n/2 \rfloor$ comparisons per round.

a latent score s_i. For a comparison between trajectories i and j, the model defines

$$P(i \succ j) = \sigma(s_i - s_j), \qquad \sigma(t) = \frac{1}{1 + \exp(-t)}. \tag{16}$$

Let $m_{ij} = 1$ if $w_{ij} = i$ and $m_{ij} = 0$ otherwise. We fit the scores by minimizing the weighted negative log likelihood with ℓ_2 regularization:

$$\mathcal{L}_{\mathrm{BTL}}(s) = - \sum_{(i,j,w_{ij},\alpha_{ij}) \in \mathcal{C}_y} \alpha_{ij} \left[m_{ij} \log \sigma(s_i - s_j) + (1 - m_{ij}) \log \sigma(s_j - s_i) \right] + \frac{\lambda}{2} \sum_{i \in B_y} s_i^2. \tag{17}$$

Algorithm 4 gives the optimization procedure used to obtain the scores. We use a diagonal second-order preconditioner and re-center the scores after each iteration, since the downstream reward map depends only on their relative values.

Algorithm 4 Weighted BTL Score Fitting

Require: Bucket B_y, comparisons $\mathcal{C}_y$, regularization λ, learning rate η
Initialize $s_i \leftarrow 0$ for all $i \in B_y$
repeat
  for $i \in B_y$ **do**
    $g_i \leftarrow \lambda s_i$
    $d_i \leftarrow \lambda$
  end for
  for $(i, j, w_{ij}, \alpha_{ij}) \in \mathcal{C}_y$ **do**
    $m_{ij} \leftarrow \mathbf{1}\{w_{ij} = i\}$
    $p_{ij} \leftarrow \sigma(s_i - s_j)$
    $g_i \leftarrow g_i + \alpha_{ij}(p_{ij} - m_{ij})$
    $g_j \leftarrow g_j - \alpha_{ij}(p_{ij} - m_{ij})$
    $d_i \leftarrow d_i + \alpha_{ij} p_{ij}(1 - p_{ij})$
    $d_j \leftarrow d_j + \alpha_{ij} p_{ij}(1 - p_{ij})$
  end for
  for $i \in B_y$ **do**
    $s_i \leftarrow s_i - \eta\, g_i/(d_i + \epsilon)$
  end for
  Center scores: $s_i \leftarrow s_i - \frac{1}{|B_y|} \sum_{j \in B_y} s_j$
until convergence
return scores $\{s_i : i \in B_y\}$

A.3 Trajectory Summarization

Before pairwise judgment, we convert each raw agent trajectory into a compact step-wise representation h_i. We parse the interaction transcript into an ordered sequence of tool actions and resulting observations. Because the judge is intended to compare externally visible debugging behavior, we remove the model's internal thinking and retain the action-observation trace. We use deterministic reductions whenever possible and only fall back to LLM-based summarization when an observation remains too long to preserve directly.

Action reparsing. Tool actions are originally emitted in a verbose function-call format. We parse each assistant message to extract the tool name and its parameters, and render the action as a compact function-style call, such as `tool_name(arg=value)`. This preserves the invoked tool and argument values while removing formatting overhead such as XML-style wrappers. Multiline values are kept as multiline strings, structured values such as lists or dictionaries are preserved, numbers and booleans are left unquoted, and ordinary strings are JSON-escaped. If no valid tool call is found, we retain the original action text; if a tool is found but has no parameters, we render it as a zero-argument call.

Observation reduction. Observations are reduced using tool-specific deterministic rules before any LLM-based summarization. Short observations are kept unchanged. For bash outputs, observations containing errors, tracebacks, or exceptions are preserved so that failure information remains visible to the judge. For file-editor observations, we distinguish file views from other editor operations. When a long file view is produced by a numbered `cat`-style output, we retain the first and last portions and insert a marker indicating that the middle was omitted. Other compact file-editor outputs are kept unchanged. If an observation remains long and does not match one of these safe reduction cases, we summarize it with an observation-specific prompt that preserves the key command outcome, relevant files or paths, and any errors.

A.4 Other Potential Instantiations of Behavioral Reward

Beyond pairwise ranking, we explored two alternative instantiations of behavioral reward: group ranking and rubric grading. In preliminary experiments, neither produced a clear improvement, so we did not tune or investigate them further. Nevertheless, they illustrate other ways in which behavioral judgment could be used to refine verifier rewards.

Group ranking. Group ranking evaluates all trajectories in the same verifier-outcome bucket jointly. Each trajectory is first summarized through a structured process evaluation covering aspects such as localization, root-cause analysis, fix quality, verification, and overall behavior. The judge then considers the full group and assigns each trajectory a coarse score from 1 to 5, with ties allowed. These scores are mapped to bounded auxiliary rewards within the corresponding verifier-outcome bucket. Compared with pairwise ranking, this approach gives the judge a global view of the group and can reduce the number of direct comparison calls.

Rubric grading. Rubric grading evaluates each trajectory independently against a predefined set of behavioral criteria. The judge determines whether each progress criterion is satisfied and whether any undesirable behavior, represented as a red flag, is present. Unsatisfied progress criteria and triggered red flags produce weighted penalties that refine the verifier reward. This approach is simple, parallelizable, and provides an absolute behavioral assessment without requiring comparisons among trajectories.

Discussion. The three approaches make different trade-offs. Pairwise ranking provides direct relative supervision and requires only local comparisons, but requires additional judgment compute to evaluate multiple pairs. Group ranking compares the entire bucket more efficiently and captures global relative structure, but its scores remain relative to the particular group and may be sensitive to group composition or listwise presentation. Rubric grading is less expensive and produces an absolute assessment for each trajectory, but it does not directly compare trajectories and depends on the completeness and calibration of the predefined rubric. Conversely, both pairwise and group ranking lack a task-independent absolute quality scale. Combining absolute rubric-based assessment with comparative ranking may therefore be a useful direction for future work.

Parameter	Value
Model Configuration	
Max Prompt Length	4096
Max Response Length	32768
Training Settings	
Train Dataset	R2E-Gym Subset
Train Batch Size	64
Learning Rate	1×10^{-6}
RL Settings	
Advantage Estimator	LOOP
Rollout Samples	8
Rollout Temperature	1.0
PPO Mini-batch Size	64
KL Loss	False
Entropy Coefficient	0.0
Clip Ratio High	0.28
Training Max Agent Steps	40
BSR Reward Settings	
Rewarded Outcome Bucket	Fail ($y = 0$)
Reward Amplitude a_0	0.15
Deadzone Threshold δ_0	0.5
Softness τ_0	0.5

Table 7: Important hyperparameters used in the main experiments.

B Additional Experiment Details, Results, and Analysis

B.1 Detailed Setup

We follow RLLM for training and use the filtered R2E-Gym dataset (4,578 tasks) as our training set. We show important parameters in Table 7. We evaluate on SWE-bench Verified (500 tasks) and SWE-bench-Live Lite (300 tasks). SWE-bench Verified is evaluated periodically throughout training. For both benchmarks, evaluation uses a maximum of 50 agent steps, temperature 1, and three runs. For each benchmark, we report the mean and standard deviation of accuracy across three evaluation runs. Pass@3 is the fraction of tasks solved in at least one of the three runs.

For Qwen3-14B, we train on 8 GPU nodes, each containing 8 A100-SXM4-40G GPUs. For Qwen3-32B, we use 16 nodes. Each training run takes 6 to 8 days.

Training observations. For Qwen3-14B, step-limit terminations become more frequent during training, while the truncation rate initially decreases and later rises. The logged entropy loss also increases. We observe that more than 95% of successful trajectories finish in fewer than 40 steps, motivating a reduction of the training rollout limit from 50 to 40 steps. In our experiments, this change improves performance and reduces training time, whereas lowering the sampling temperature or adding KL regularization reduces performance. Evaluation retains a maximum of 50 steps.

Kubernetes setting. Environment provisioning is a major systems challenge for SWE-agent training, since each optimization step may require 512 concurrent environments (64 tasks $\times$ 8 rollouts). In our experiments, we manually configured a two-node Kubernetes cluster, with each node providing 128 CPU cores and approximately 500 GB of memory. To reduce startup latency, we pre-pull and cache all required container images. We also use small per-pod resource requests (80 m CPU and 300 MiB memory), allowing the cluster to schedule a large number of concurrent environments. Several configuration details are important in practice:

- Ensure that pods have outbound network access when required by the task.

- Increase the maximum number of pods allowed per node to support the desired level of parallelism.
- Configure Kubernetes and containerd image-garbage-collection and disk-eviction policies to avoid removing frequently used cached images.
- Maintain a stable and automatically recoverable connection between the GPU training machines and CPU environment nodes.
- Promptly clean up completed or failed pods to prevent resource leakage.

Additional setup instructions and configuration examples are included in our codebase.

Performance optimization. To reduce end-to-end training time, we overlap environment provisioning with other parts of the training pipeline. First, while the model is being updated for the current batch, we asynchronously pre-create the environments required by the next batch. This hides a substantial portion of environment startup latency behind model optimization. Second, we remove the synchronization barrier between environment creation and trajectory generation. Rather than waiting for all environments in a batch to become ready, each rollout begins as soon as its own environment has been created. This allows fast-starting environments to enter trajectory generation immediately and reduces idle time caused by slow environment initialization or other stragglers.

Training-time overhead. BSR reuses the policy-serving infrastructure for post-rollout comparisons, without a separately deployed reward model. In the measured Qwen3-14B configuration, BSR required 19.86% more wall-clock training time than verifier-only RL under the same training and serving GPU allocation. This additional judging is performed during training; evaluation uses the acting policy without BSR judge calls. While our method results in overhead, however our method is extremely easy to apply without additional computation resources and setup. Also, extending verifier-only training from 270 to 400 steps does not recover the BSR gain: BSR at 270 steps already exceeds the 400-step baseline in average accuracy. This suggests that the improvement is not explained simply by additional optimization steps, although a fully compute-matched comparison remains an interesting direction. We note that more selective comparison scheduling, improved batching, and greater overlap between ranking and other training stages could further reduce the cost.

Table 8: Terminal-Bench 2.1 performance (%) over 89 tasks and three evaluation trials.

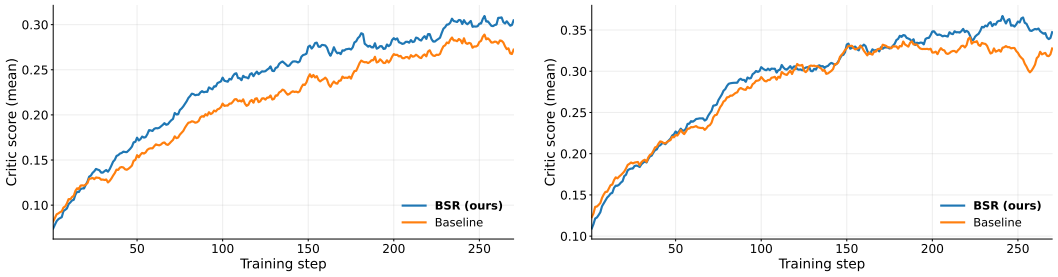
Size	Budget	Method	Avg. Acc.	Pass@3
14B	270	Baseline	7.12	14.61
		BSR (ours)	**7.87**	**15.73**
	400	Baseline	6.74	14.61
		BSR (ours)	**8.24**	**15.73**
32B	270	Baseline	4.87	8.99
		BSR (ours)	**5.62**	**13.48**

B.2 Additional Evaluation on Terminal-Bench 2.1

We additionally evaluate BSR on Terminal-Bench 2.1. Each configuration is evaluated on 89 tasks with three independent evaluation trials. Table 8 reports average accuracy across the three trials and Pass@3, the fraction of tasks solved by at least one trial. BSR consistently improves both average accuracy and Pass@3 across all evaluated model sizes and training budgets, demonstrating that the gains extend beyond the primary evaluation setting.

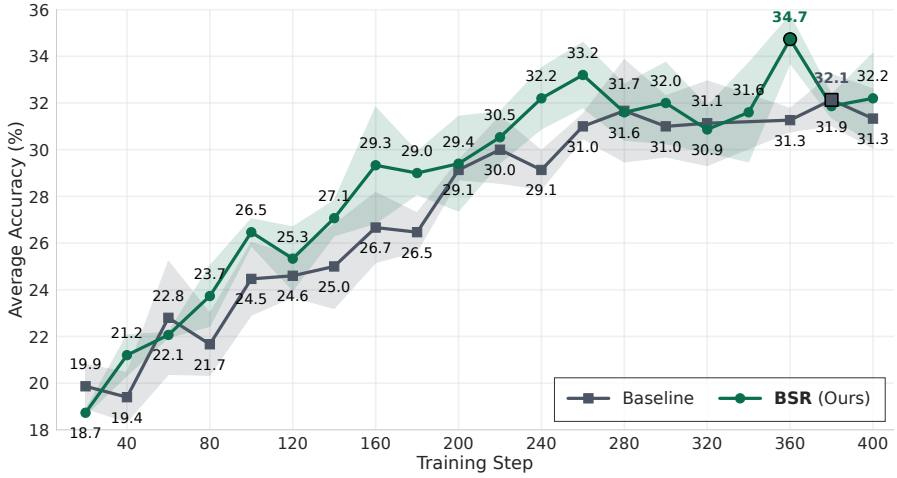
B.3 Additional Results

We provide additional training curves in this section. Figure 3 shows the mean scores during training for both Qwen3-14B and Qwen3-32B. Figure 4 presents the full 400-step

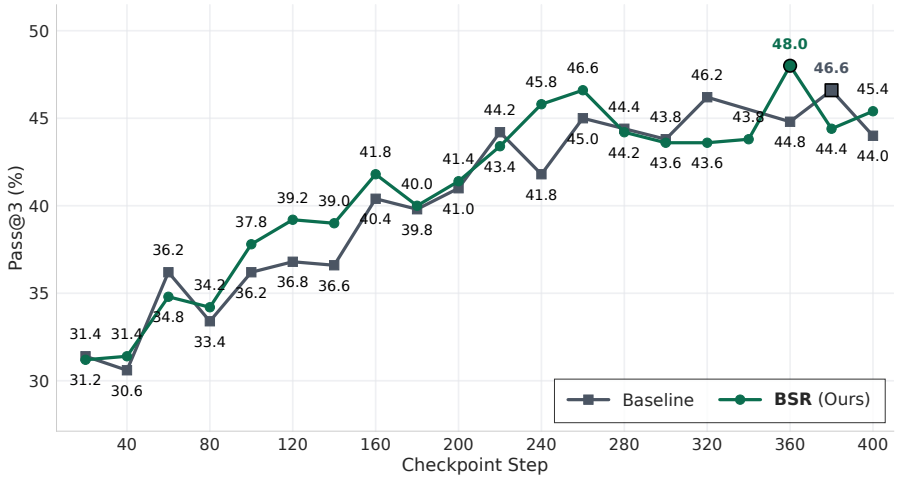


(a) 14B model. (b) 32B model.

Figure 3: Mean critic scores on training set for BSR and the baseline. Curves are smoothed with EMA with a span of 31 (Note here the critic score is just the score from the execution verifier).

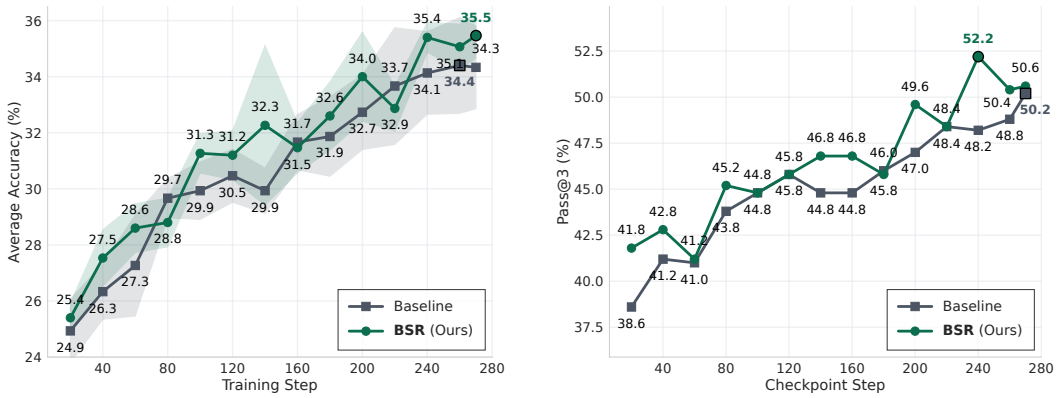


(a) Average Accuracy.



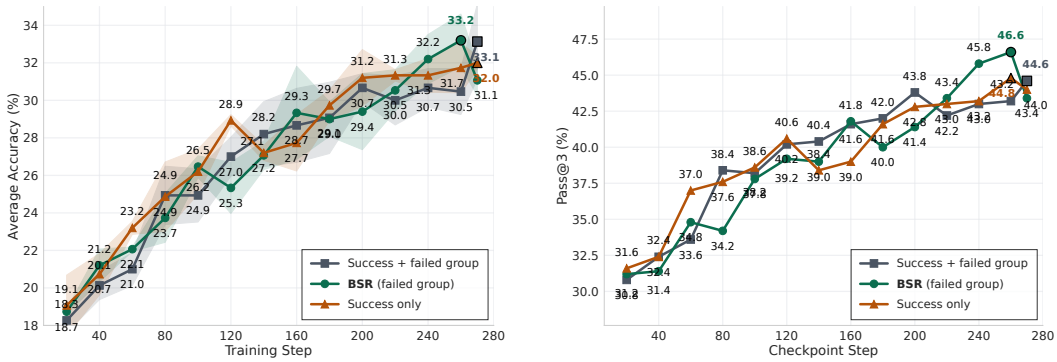
(b) Pass@3.

Figure 4: Evaluation on SWE-bench Verified while training Qwen3-14B for long run of 400 steps. We evaluate with temperature = 1 and max_steps = 50 for 3 runs.



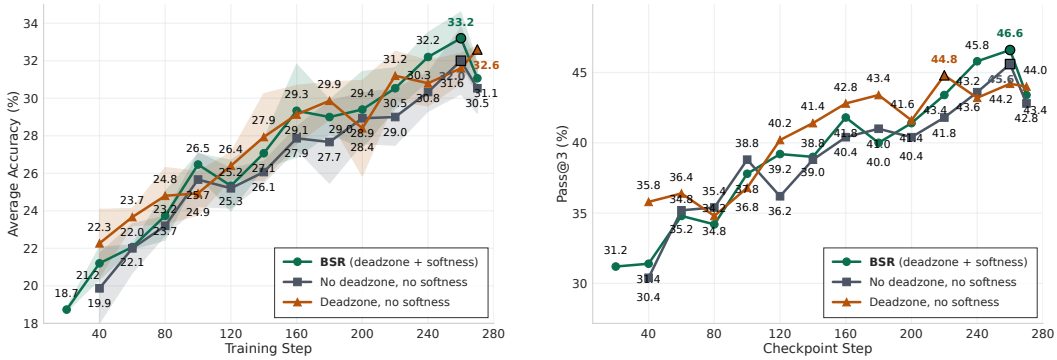
(a) Average Accuracy. (b) Pass@3.

Figure 5: Detailed plot of evaluation on SWE-bench Verified while training Qwen3-32B. We evaluate with temperature = 1 and max_steps = 50 for 3 runs.



(a) Average accuracy. (b) Pass@3.

Figure 6: Verifier-outcome-bucket ablation on SWE-bench Verified while training Qwen3-14B. We compare applying BSR to failed trajectories only, successful trajectories only, and both successful and failed trajectories.



(a) Average accuracy. (b) Pass@3.

Figure 7: Reward-mapping ablation on SWE-Bench Verified while training Qwen3-14B. We compare the full BSR reward mapping against variants without the deadzone and/or softness.

Statistic	Baseline	BSR (Ours)
Process metrics		
Reference file viewed before first edit ↑	0.733	**0.748**
Search-to-edit conversion ↑	0.465	**0.494**
Post-edit test rate ↑	0.301	**0.311**
Duplicate command rate ↓	0.209	**0.208**
Basic workload statistics		
Average total steps	28.90	28.42
Average first edit step	5.33	5.24
Average number of searches	4.74	4.06
Average number of reads	5.39	5.56
Average number of edits	12.46	12.16
Average number of tests	0.747	0.797
Average number of post-edit tests	0.746	0.797
Average number of errors	13.38	12.76
Average unique files read	1.97	2.11
Average unique files edited	3.02	3.17
Average search result files per search	7.20	7.12
Average observation lines per step	32.21	31.85
Average total time (s)	1132.28	1068.55
Average total tokens	18283.74	18572.82
Trajectory count	1500	1500

Table 9: Appendix trajectory behavior statistics over the full set of trajectories. Arrows indicate the preferred direction for process metrics. Basic workload statistics are descriptive and should be interpreted in context.

curves on SWE-Bench Verified for Qwen3-14B, while Figure 5 shows the detailed training curves for Qwen3-32B. Figures 6 and 7 provide the full results on SWE-Bench Verified for the verifier-outcome-bucket ablation and reward-mapping ablations, respectively.

B.4 Trajectory Behavior Statistics

We analyze 1,500 evaluation trajectories per method on SWE-bench Verified from Qwen3-14B training logs and report both workload statistics and debugging-process metrics.

Metrics Used. We report two groups of trajectory-level statistics. The first group contains basic workload statistics, including the number of steps, first edit step, numbers of search/read/edit/test actions, post-edit tests, errors, unique files read and edited, runtime, and token usage. These statistics describe how the agent spends computation, but are not always directly comparable: for example, fewer steps may indicate efficiency or premature stopping, while more searches may indicate either careful investigation or unfocused exploration.

The second group contains process metrics that more directly reflect debugging behavior. *Reference file viewed before first edit* measures whether the agent localized a reference-relevant file before editing. *Search-to-edit conversion* measures whether search results led to downstream edits. *Post-edit test rate* measures whether the agent verified its changes after editing. *Duplicate command rate* captures repeated commands. We treat these process metrics as the primary behavioral indicators, while workload statistics provide supporting context.

Behavioral Differences. Table 9 shows that BSR improves several process metrics associated with investigation and verification. Compared with the baseline, BSR more often views a reference-relevant file before the first edit (0.748 vs. 0.733), has higher search-to-edit conversion (0.494 vs. 0.465), and performs more post-edit testing (0.311 vs. 0.301). It also slightly reduces duplicate commands (0.208 vs. 0.209). The workload statistics are consis-

tent with this shift: BSR uses fewer steps, performs fewer searches, reads slightly more, runs more tests, encounters fewer errors, and completes trajectories faster.

Summary. BSR shows stronger reference-oriented investigation, search-to-edit follow-through, and post-edit verification. These aggregate patterns are consistent with changes in investigation and verification behavior, with workload statistics providing complementary evidence about how the agent allocates its debugging effort.

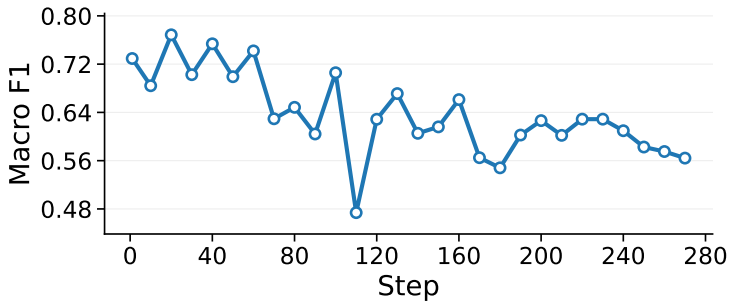


Figure 8: Agreement with Kimi-2.5 on sampled pairwise trajectory comparisons over training, measured by macro F1. At each evaluation point, 200 on-policy comparison pairs are sampled and re-evaluated using Kimi-2.5 as an external reference judge.

B.5 Judge Quality Investigation

We evaluate the policy's pairwise judgments by measuring agreement with Kimi K2.5 (Team et al., 2026) on 200 sampled comparisons every 10 training steps. As shown in Figure 8, the self-judge maintains meaningful agreement with the external judge throughout training, with macro F1 generally around 0.6–0.7. Although agreement decreases somewhat at later checkpoints, it remains substantial and the corresponding BSR models continue to improve in downstream performance. This suggests that the policy's judgments remain useful for reward refinement, while improving the stability of self-judgment over training is an interesting direction for future work.

You are a strict evaluator for software engineering agents. You will be shown two candidate trajectories produced by two agents solving the same programming task. Each trajectory contains the agent's actions, and outcomes.

Each trajectory may also end with a compact stats block. Attend to those metrics as supporting evidence about efficiency and behavior, but do not treat them as the sole basis for judgment.
Both trajectories ultimately succeeded and produced a correct patch that passes all tests.
Your goal: compare the two trajectories and decide which one demonstrates better debugging and code-editing behavior that, if reinforced during training, is more likely to improve future task accuracy.
Use a rubric-based pairwise judgment. Evaluate only observable behavior in the trajectories. Do not judge writing quality.
For correct-vs-correct comparisons, prefer the trajectory whose behavior is more efficient, causally grounded, and safer to reinforce for future generalization.

Rubric for correct trajectories:
1. Failure-surface investigation:
- Did it investigate the right failure surface?
- Did it look in the right code paths, files, or behaviors?
2. Evidence-based diagnosis:
- Did it use evidence to narrow uncertainty before changing code?
- Did it run useful checks or inspections to distinguish between plausible causes?
3. Targeted and efficient execution:
- Were actions and edits targeted and efficient?
- Did it avoid unnecessary exploration, wasted motion, or overly broad changes? 4. Root-cause alignment:
- Did the patch address the root cause rather than only symptom-matching? - Was the reasoning connected to why the fix should work?
5. Safety and regression awareness:
- Did it preserve existing behavior and avoid likely regressions?
- Did it avoid risky or unnecessary edits?

Useful trajectory stats to attend to when they are present:
- num_bash_runs_before_last_edit and num_bash_runs_after_last_edit: useful for checking whether the agent validated before changing code and whether it verified after making edits.
- num_repeated_actions: repeated identical actions are usually a negative signal.
- num_edits_without_prior_view: edits made without first viewing the file are usually a negative signal.

Ignore:
- explanation style
- verbosity
- confidence
- stylistic fluency
- superficial similarity to a known patch unless it directly reflects better debugging behavior

Always pick one winner: "A" or "B".
Then assign a coarse strength-of-preference score:
- 1 = slight
- 2 = clear
- 3 = strong

Strength should reflect how much better the winner is in this pair, not absolute trajectory quality.

Output format (first give your thinking and then the result):
<think>your analysis here</think>
Winner: A or B
Strength: 1 or 2 or 3

Figure 9: System Prompt for Success-Bucket Pairwise Comparison.

You are a strict evaluator for software engineering agents. You will be shown two candidate trajectories produced by two agents solving the same programming task. Each trajectory contains the agent's actions, and outcomes.

Each trajectory may also end with a compact stats block. Attend to those metrics as supporting evidence about efficiency and behavior, but do not treat them as the sole basis for judgment.
Both trajectories ultimately failed and produced a wrong patch that fails one or more tests. Your goal: compare the two trajectories and decide which one demonstrates better debugging and code-editing behavior that, if reinforced during training, is more likely to improve future task accuracy.
Use a rubric-based pairwise judgment. Evaluate only observable behavior in the trajectories. Do not judge writing quality.
For wrong-vs-wrong comparisons, prefer the trajectory that demonstrates more useful bug-finding behavior and is closer to eventual success, even though it did not solve the task.

Rubric for wrong trajectories:
1. Progress toward the real bug:
- Did it make progress toward the real bug, even if it failed?
- Did it move closer to the relevant code path, mechanism, or failure cause?
2. Evidence gathering and discriminative checks:
- Did it gather useful evidence or run discriminative checks?
- Did it reduce uncertainty in a meaningful way?
3. Directional quality of edits and actions:
- Were edits directionally sensible rather than random or cargo-culted?
- Were commands, inspections, and modifications plausibly useful?
4. Relevance of search path:
- Did it avoid digging deeper into irrelevant paths?
- Did it avoid getting stuck in unproductive loops or distractions?
5. Informative and recoverable failure:
- Did it fail in an informative and recoverable way?
- Did it avoid introducing extra damage, confusion, or regressions?

Useful trajectory stats to attend to when they are present:
- num_bash_runs_before_last_edit and num_bash_runs_after_last_edit: useful for checking whether the agent gathered evidence before editing and tested or probed after edits.
- num_repeated_actions: repeated identical actions are usually a negative signal.
- num_edits_without_prior_view: edits made without first viewing the file are usually a negative signal.

Ignore:
- explanation style
- verbosity
- confidence
- stylistic fluency
- superficial similarity to a known patch unless it directly reflects better debugging behavior

Always pick one winner: "A" or "B".
Then assign a coarse strength-of-preference score:
- 1 = slight
- 2 = clear
- 3 = strong

Strength should reflect how much better the winner is in this pair, not absolute trajectory quality.

Output format (first give your thinking and then the result):
<think>your analysis here</think>
Winner: A or B
Strength: 1 or 2 or 3

Figure 10: System Prompt for Failure-Bucket Pairwise Comparison.

Below are two candidate agents solving the same programming task.
GitHub Issue to be addressed:
{task}
{golden_patch_section}

Trajectory A:
{traj_a}

Trajectory B:
{traj_b}

Figure 11: User Prompt for ranking.

<think> Okay, let's compare the two trajectories. Both agents tried to solve the same problem but ended up with wrong patches. The goal is to determine which one showed better debugging and code-editing behavior.

First, looking at Trajectory A. The agent started by exploring the file structure, found the relevant multi.py file, and searched for 'get_indexer' to locate the method. They then viewed specific code sections to understand where the method was used. After several steps, they added a new 'get_indexer' method in the MultiIndex class to check for level mismatches. They created a test script and ran it, which raised the correct error. The stats show that they did one bash run after the last edit, which is good because it tests the change. There was one edit without prior view, but overall, their steps were methodical, focusing on the right files and methods.

Now, Trajectory B. The agent also explored the files, found the same multi.py and base.py. They tried modifying the base.py's get_indexer method to check for MultiIndex levels. However, they hit an import error because they changed the import statements to relative imports, which caused a circular dependency. They tried multiple times to fix the import issues, leading to repeated actions and errors. The stats show many repeated actions (11) and some edits without prior viewing. The agent's approach led to more errors and didn't resolve the import problem, even after several attempts.

Comparing the two, Trajectory A made progress toward the real bug by correctly identifying the MultiIndex's get_indexer method and adding the check there. Their edits were directional and sensible, leading to a test that showed the error. Trajectory B, while attempting to modify the base class, introduced import issues and struggled with circular dependencies, leading to more errors and less progress toward the actual solution. Even though both failed, A's approach was closer to the correct fix and had more effective debugging steps.
</think>

Reasoning:
Trajectory A demonstrated more effective debugging and code-editing behavior despite both failing.

1. **Progress toward the real bug:**
- Trajectory A directly identified the 'MultiIndex' class in 'multi.py' and focused on the 'get_indexer' method, which is the correct location for the fix. They added a method to check for level mismatches, which aligns with the issue description.
- Trajectory B attempted to modify the 'get_indexer' method in the base 'Index' class but introduced a circular import error due to incorrect relative imports, which deviated from the actual problem (which requires a 'MultiIndex'-specific fix). 2. **Evidence gathering and discriminative checks:**
- Trajectory A systematically searched for 'get_indexer' in 'multi.py' and examined code context to locate the relevant method. They created a test script and ran it to validate their change, which produced the expected error.
- Trajectory B made multiple edits to 'base.py' without fully understanding the import structure, leading to repeated errors (e.g., 'ImportError'). Their approach lacked discriminative checks to isolate the root cause.
3. **Directional quality of edits and actions:**
- Trajectory A's edits were focused on the 'MultiIndex' class and its 'get_indexer' method, which is the correct code path. Their final edit added a clear, targeted check for level mismatches.
- Trajectory B's edits were less directional. They modified the 'Index' class's 'get_indexer' method, which is not the correct location for 'MultiIndex'-specific logic. Their repeated attempts to fix import issues showed poor progress.
4. **Relevance of search path:**
- Trajectory A avoided irrelevant paths (e.g., 'base.py' was only explored for context, not as the primary fix location).
- Trajectory B repeatedly edited 'base.py' and 'multi.py' without resolving the circular import, getting stuck in a loop of unproductive fixes.
5. **Informative and recoverable failure:**
- Trajectory A's failure was recoverable: their test script raised the expected 'TypeError', confirming their logic was correct. The failure was due to incomplete implementation (e.g., not handling all edge cases).
- Trajectory B's failure was due to a cascading import error that they could not resolve, which was less informative and harder to recover from.
Conclusion:
Trajectory A's approach was more methodical, targeted, and closer to the correct solution. Their edits were meaningful and validated through testing, while Trajectory B's attempts were hindered by circular dependencies and less relevant modifications.
Winner: A
Strength: 3

Figure 12: Example Ranking Response from Qwen3-14B.