Beyond Binary Verifiable Rewards: Behavioral Self-Rewarding for SWE Agents

Yiran Wu
Pennsylvania State University
yiran.wu@psu.edu

Andrew Zhao
Mistral AI
andrew.zhao@mistral.ai

Yurui Chang
Pennsylvania State University
yuruic@psu.edu

Qingyun Wu
AG2AI
qingyun@ag2.ai

Xuan Feng
Microsoft Research
xuafeng@microsoft.com

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 introduce Behavioral Self-Rewarding **(BSR)**, a reward-refinement method that preserves executable verification as the source of correctness while using the agent's own behavioral judgments to rank trajectories with the same verifier outcome. BSR converts pairwise trajectory comparisons into bounded auxiliary rewards, recovering learning signal from otherwise indistinguishable rollout groups. Experiments on repository-level software engineering tasks show that BSR consistently improves over verifier-only RL across Qwen3-14B and Qwen3-32B, with the largest benefit coming from distinguishing among failed trajectories. These results demonstrate that behavioral self-rewarding can effectively complement sparse verifiable rewards for training software engineering agents.

 **Code**  **Project Page**  **Logs**  **Models**

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 (Wu et al., 2024; 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; Wu et al., 2025; Xia et al., 2024; Wang et al., 2025a; Wu et al.).

This setting is also a natural fit for reinforcement learning with verifiable rewards. In RLVR, the reward is supplied by an automatic checker rather than a learned preference model: a mathematical answer can be matched against a solution, a program can be run against tests, and a software patch can be validated by executing a repository's test suite (Shao et al., 2024; Guo et al., 2025). Recent work extends this idea from single-turn reasoning to multi-step agentic RL, where a model learns 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 particularly compelling instance of this broader agentic RLVR setting because the interaction is long-horizon and behaviorally complex, while the final

outcome can often still be determined by an executable verifier (Pan et al., 2024; Wei et al., 2026; Jain et al., 2025; Golubev et al., 2025).

However, the same verifier that makes RL attractive also creates a severe sparsity problem. A binary test outcome gives a reliable answer to whether the final patch works, but it compresses a rich interaction trajectory into a single success or failure signal. This distinction is crucial for multi-step software-engineering agents, since failed trajectories can vary substantially in quality. Some failures make meaningful progress toward solving the issue, whereas others contain little useful progress, yet both receive reward zero. Under group-relative objectives such as GRPO or leave-one-out PPO, this problem becomes even sharper: when all rollouts for a task fail, the group has no reward contrast and therefore provides little or no policy-gradient signal, despite containing meaningful differences in process quality (Schulman et al., 2017; Shao et al., 2024; Chen et al., 2025a). Prior work has sought denser supervision through process supervision and step-aware verification, which reward intermediate reasoning quality (Li et al., 2023; Lightman et al., 2024; Wang et al., 2024). Reflection-based agents instead use feedback from previous attempts to guide future behavior (Shinn et al., 2023). Preference-based and AI-feedback methods provide another source of supervision when scalar rewards are difficult to specify (Christiano et al., 2017; Bai et al., 2022). These directions suggest that richer behavioral signals can complement final outcomes, but for RLVR in software engineering, the central challenge is to recover such signal without replacing the executable verifier that defines task success.

We view this challenge as one component of the broader goal of building self-evolving language agents: enabling an agent to extract useful learning signals from the interaction experience it generates. Recent RLVR systems show that models can improve through interaction with verifiable environments, while self-evolving approaches suggest that agents can increasingly generate and learn from their own experience when grounded by reliable feedback (Wang et al., 2025b; Zhao et al., 2026; Chang et al., 2026). Yet generating experience is not sufficient if much of that experience is reduced to indistinguishable binary outcomes. In parallel, self-rewarding and LLM-as-a-judge methods show that language models can provide useful evaluative signals for their own outputs or for outputs from other models (Yuan et al., 2024; Zheng et al., 2023; Liu et al., 2023). For software-engineering RL, this suggests a natural division of labor: the executable verifier determines whether an attempt succeeds, while model judgment distinguishes behavioral quality among attempts that the verifier already treats as equivalent. In this way, the agent can continuously improve from its own interaction experience by extracting supervision beyond binary outcomes, while retaining the reliable external signal that anchors correctness.

We propose *Behavioral Self-Rewarding* (BSR), a reward-refinement method for multi-step agentic RLVR. BSR keeps the executable verifier as the source of final correctness, but augments it with a small self-rewarding signal over trajectories that the verifier treats as equivalent. For each software-engineering task, the agent samples multiple trajectories and receives the ordinary test-based reward. Trajectories with the same verifier outcome are then compared within the same task using a rubric-based judge that evaluates observable debugging behavior rather than writing style or confidence. Conditioned on the task and its reference patch, the judge assesses whether each trajectory makes task-relevant progress and uses the available evidence effectively. These judgments are aggregated into a task-local ranking score and converted into a bounded auxiliary reward. As a result, one failed trajectory can be preferred over another failed trajectory, or one successful trajectory over another successful trajectory, without allowing model judgment to override the pass/fail ordering defined by the verifier.

BSR integrates naturally with group-relative RL by introducing contrast within verifier-outcome groups that would otherwise provide little or no learning signal under binary RLVR. This is particularly important for all-fail groups, where expensive interaction trajectories may contain useful behavioral differences despite receiving identical rewards. At the same time, the additional signal remains controlled: verifier success stays dominant, behavioral comparisons are restricted to trajectories with the same outcome, and no separately trained reward model is required. Empirically, BSR improves Qwen3-14B average accuracy from 32.1% to 34.7% and increases Qwen3-32B Pass@3 from 48.8% to 52.2% on SWE-Bench Verified. Our ablations further show that the largest benefit comes from distinguishing

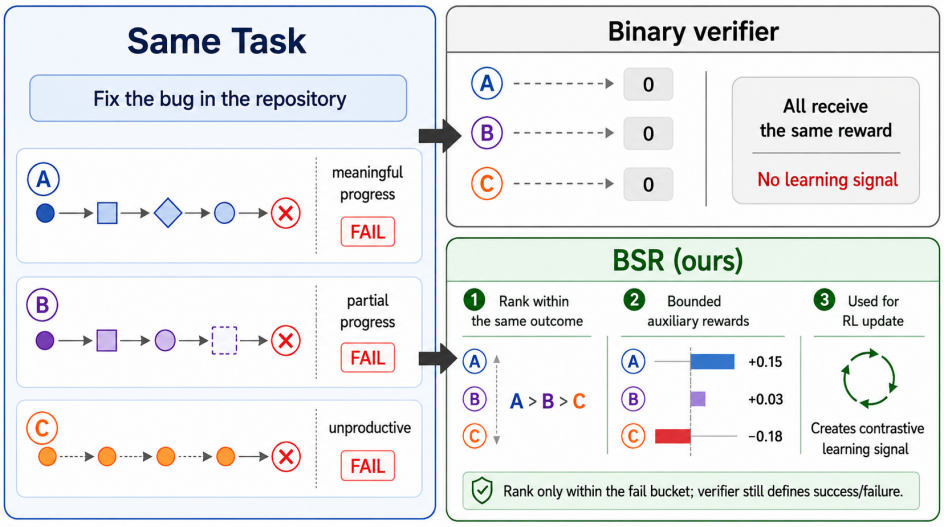


Figure 1: Overview of Behavioral Self-Rewarding (BSR). Binary rewards provide no contrast when trajectories share the same verifier outcome, while BSR ranks them within the outcome group to recover a bounded learning signal. [Current is a draft, final figure @ Yurui]

among failed trajectories, supporting the central motivation that behavioral self-rewarding can turn otherwise underutilized interaction experience into useful training signal while preserving the reliability of verifier-based RL.

2 Preliminaries

We consider repository-level software-engineering tasks under *Reinforcement Learning with Verifiable Rewards* (RLVR). A task x consists of an initial repository state, a natural-language issue description, and an executable verifier, such as a unit-test suite. A policy π_θ acts as an interactive software-engineering agent that interacts with the repository through developer tools, gathering evidence and modifying the code before producing a final patch. For each task, we sample a group of K trajectories,

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

where each trajectory τ_i records the full multi-step interaction between the agent and the software environment. After the trajectory terminates, the executable verifier returns 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 RLVR, this verifier-derived reward is used directly as the optimization signal, rather than being produced by a learned reward model. RLVR has been widely used for reasoning and coding tasks where correctness can be checked automatically through exact answers, execution, or tests (Shao et al., 2024; Guo et al., 2025). In the verifier-only setting, we write $R_i = R_{\text{ver}}(\tau_i)$ for the reward assigned to trajectory i.

Group Relative Policy Optimization (GRPO) optimizes a policy using multiple sampled outputs for the same input and computes advantages by normalizing rewards within the sampled group (Shao et al., 2024). For trajectory i, GRPO defines the group-relative advantage as

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

where $\epsilon > 0$ is a small constant for numerical stability. The resulting advantage is used in a clipped policy-gradient objective without fitting a separate value model. LOOP adapts the same group-relative principle to long-horizon interactive agents, using a PPO-style objective with a leave-one-out group baseline instead of a learned critic (Chen et al., 2025a). Its leave-one-out advantage is

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

With binary verifier rewards, group-relative objectives provide reward contrast only when a sampled group contains different verifier outcomes. If all rollouts fail, all rewards are zero; if all rollouts pass, all rewards are one. In either case, every trajectory receives zero group-relative advantage, so the group provides no learning signal from reward differences.

This reward collapse motivates a refinement signal beyond final verifier correctness. We use self-judgment to distinguish behavioral quality among trajectories with the same verifier outcome, while leaving correctness to the executable verifier. Prior work has shown that language models can provide useful evaluative feedback for their own outputs or for outputs from other models (Yuan et al., 2024; Zheng et al., 2023; Liu et al., 2023). In our setting, this feedback is used only as a bounded auxiliary signal, so the executable verifier remains the anchor for task correctness.

3 Method

We propose *Behavioral Self-Rewarding* (BSR), a reward-refinement method for self-evolving software-engineering agents. The central idea is to preserve the executable verifier as the source of task correctness while allowing the model to refine rewards using its own behavioral judgment. For each task, we first run the policy as an interactive agent and obtain the verifier outcome for each trajectory. We then provide a judge with the task, the reference patch, the verifier outcome, and compact representations of the trajectories. The reference patch is provided only to the post-rollout judge, not to the acting policy, and helps the judge distinguish progress toward the task-relevant failure mechanism from superficially plausible but irrelevant exploration. The judge does not decide whether a patch is correct; that remains the role of the verifier. Instead, it distinguishes trajectories that the verifier treats as equivalent, assigning a small auxiliary self-judge reward that reflects relative behavioral quality. The adjusted training reward follows the verifier-plus-self-reward form:

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

where r_i is a bounded self-judge reward. The verifier determines the base reward, and self-judgment only refines the reward among trajectories with the same verifier outcome.

This paradigm admits several instantiations. A judge can compare trajectories pairwise, rank an entire group jointly, or score each trajectory against an absolute behavioral rubric. In this work, we use pairwise ranking as the main proposal because it is local, easy to audit, and does not require the judge to maintain a globally calibrated quality scale across tasks. We refer to other potential variations in Appendix A.4.

3.1 Pairwise Behavioral Ranking

BSR is performed on top of group-relative RL. For each task x, the policy samples K trajectories, and the verifier assigns each trajectory an outcome

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

We split the trajectories into verifier-outcome buckets,

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

BSR uses an outcome-specific reward amplitude $a_y \geq 0$ for each bucket; setting $a_y = 0$ disables behavioral reward refinement for that outcome. When BSR is applied to a bucket, pairwise comparisons are restricted to trajectories with the same verifier outcome, ensuring that behavioral judgment refines trajectories within an outcome class without overriding the pass/fail ordering induced by the verifier.

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 $a_y \geq 0$, deadzone thresholds $\delta_y \geq 0$, softness parameters $\tau_y > 0$ for $y \in \{0, 1\}$

Sample trajectories $\{\tau_i\}_{i=1}^{K} \sim \pi_\theta(\cdot \mid x)$
for $i = 1, \ldots, K$ **do**
  $y_i \leftarrow V(\tau_i)$ ▷ $y_i = 1$ if the trajectory passes the verifier; otherwise $y_i = 0$
  $h_i \leftarrow \textsc{SummarizeTrajectory}(\tau_i)$
end for
for $y \in \{0, 1\}$ **do**
  $B_y \leftarrow \{i : y_i = y\}$ ▷ Bucket trajectories by verifier outcome
  if $a_y = 0$ or $|B_y| < 2$ **then**
    $r_i \leftarrow 0$ for all $i \in B_y$
    continue
  end if
  $\mathcal{C}_y \leftarrow \textsc{PairwiseCompare}(x, p^\star, \{h_i : i \in B_y\}, B_y, y)$ ▷ See Algorithm 3
  $\{r_i : i \in B_y\} \leftarrow \textsc{PairwiseRewardMap}(\mathcal{C}_y, B_y, a_y, \delta_y, \tau_y)$ ▷ See Algorithm 2
end for
for $i = 1, \ldots, K$ **do**
  $R_i \leftarrow y_i + r_i$ ▷ Add bounded self-reward to verifier reward
end for
return refined rewards $\{R_i\}_{i=1}^{K}$

Outcome-separated rewarding. The pass and fail buckets serve different purposes. In the failed bucket, the judge is asked to identify which failed trajectory shows more useful progress toward eventual success. This includes whether the agent searched the relevant part of the repository, gathered evidence, made directionally plausible edits, and failed in a recoverable way. In the successful bucket, the judge instead distinguishes the quality of success: whether the agent solved the task efficiently, made a targeted fix, avoided brittle or unnecessary changes, and verified the result. We keep these two comparisons separate because the meaning of a good failure is different from the meaning of a good success.

Trajectory summarization. Raw agent trajectories can be long, redundant, and may not fit within the judge's context length. Before judgment, each trajectory is converted into a compact representation. We remove the thinking section from all steps and keep the exact actions. We truncate observations when they exceed a token threshold. We also include deterministic unbiased statistics, such as the number of command executions before and after the final edit, to assist judgment. We provide the full summarization format in Appendix A.3.

Weighted pairwise comparisons. For each scheduled pair (i, j) within a bucket, the judge returns a preferred trajectory $w_{ij} \in \{i, j\}$ and a coarse strength label $\alpha_{ij} \in \{1, 2, 3\}$. The strength label is used as a comparison weight. A slight preference contributes less than a clear or strong preference, which allows the reward mapper to use more information than a binary win/loss label while avoiding fine-grained scalar judgments that are difficult to calibrate. To reduce positional artifacts, pair presentations are randomized during pair scheduling.

3.2 From Pairwise Comparisons to Rewards

For each outcome bucket B_y, let $\mathcal{C}_y$ denote the collection of pairwise comparisons actually performed by the comparison scheduler. Each comparison contains the pair (i, j), the preferred trajectory w_{ij}, and its strength α_{ij}. BSR converts these weighted comparisons into a bounded auxiliary reward.

The conversion has three stages: fit task-local ranking scores, denoise the normalized scores, and center and bound the resulting self-reward. We aggregate weighted pairwise comparisons into task-local scalar scores using a regularized weighted Bradley–Terry model (Bradley and Terry, 1952), described in Appendix A.2. The resulting scores are then normalized within each verifier-outcome bucket before applying the deadzone, centering, and bounding steps.

Task-local score normalization. The BTL scores are task-local: their absolute scale has no meaning across tasks or outcome buckets. For buckets with at least two trajectories, we standardize the scores within the verifier-outcome bucket:

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

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

Soft deadzone for noisy preferences. We next apply a soft deadzone to suppress small ranking differences that are likely to reflect judge noise rather than meaningful behavioral differences:

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

The threshold δ_y defines an indifference region around zero: trajectories whose normalized scores fall inside this region receive no auxiliary signal. The softness parameter τ_y controls how quickly the signal grows outside the deadzone and saturates for large score differences (Drucker et al., 1996; Donoho, 1995).

Centering and bounding. Let

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

be the set of trajectories outside the deadzone. Trajectories outside $\mathcal{A}_y$ receive zero auxiliary score. If $\mathcal{A}_y$ is empty, we set $q_i = 0$ for all $i \in B_y$. 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 bound them by their maximum absolute value within the active set:

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

For all inactive trajectories $i \notin \mathcal{A}_y$, we set $q_i = 0$. Thus $q_i \in [-1, 1]$. This step keeps the auxiliary signal contrastive within the active subset while controlling its magnitude.

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

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

where $a_y \geq 0$ is a small outcome-specific amplitude. The final training reward is

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

Because $q_i \in [-1, 1]$, the smallest possible reward for a passing trajectory is $1 - a_1$, while the largest possible reward for a failing trajectory is a_0. Therefore, choosing nonnegative amplitudes such that

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

guarantees that every verifier-passing trajectory receives a higher reward than every verifier-failing trajectory. Thus, self-judgment can refine behavior within each verifier outcome without changing the pass/fail ordering induced by the executable verifier.

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 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 reliable auxiliary signal
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 $q_i = 0$

Table 1: Performance comparison across Qwen3 model sizes and training budgets.

Size	Steps	Method	Avg. Acc. ± Std.	Max Acc.	Pass@3
14B	270	Baseline	31.0 ± 0.5	31.4	45.0
		BSR (ours)	$\mathbf{33.2 \pm 1.4}$	**34.8**	**46.6**
	400	Baseline	32.1 ± 1.1	33.4	46.6
		BSR (ours)	$\mathbf{34.7 \pm 1.0}$	**35.6**	**48.6**
32B	270	Baseline	34.4 ± 1.7	36.0	48.8
		BSR (ours)	$\mathbf{35.5 \pm 0.8}$	**36.2**	**52.2**

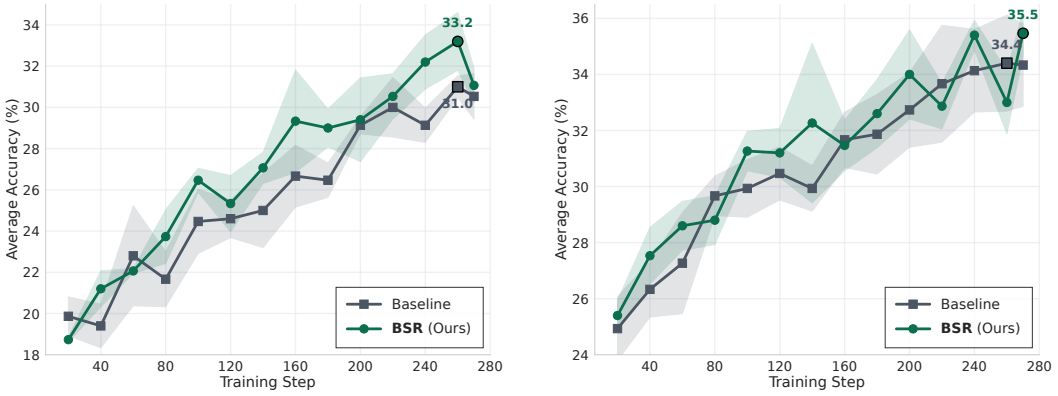
4 Experiments

Setup We adopt RLLM (Tan et al., 2025) for all experiments. We train on a filtered version of the R2E-Gym dataset (Jain et al., 2025) and evaluate on SWE-Bench Verified (Jimenez et al., 2024). For Qwen3-14B, we use 8 GPU nodes, each equipped with 8 A100-SXM4-40GB GPUs; for Qwen3-32B, we use 16 nodes. In the main BSR setting, we apply behavioral reward refinement only to the failure bucket, setting $a_0 = 0.15$ and $a_1 = 0$, with deadzone threshold $\delta_0 = 0.5$ and softness parameter $\tau_0 = 0.5$. We use the failure bucket because it performs similarly to or better than applying BSR to both outcome buckets while requiring fewer pairwise comparisons (see Section 4.3). Complete training and reward settings are provided in Appendix B.1.

4.1 Main Results

Table 1 reports peak performance at the end of each predefined training budget, with all other metrics in the same row evaluated at that checkpoint. BSR consistently outperforms the verifier-only baseline across both model sizes and training budgets. On Qwen3-14B trained for 270 steps, BSR improves average accuracy from 31.0% to 33.2%, maximum accuracy from 31.4% to 34.8%, and Pass@3 from 45.0% to 46.6%. With a longer 400-step training budget, the gain remains consistent: average accuracy increases from 32.1% to 34.7%, while Pass@3 improves from 46.6% to 48.6%. The improvement also transfers to the larger Qwen3-32B model. At 270 training steps, BSR raises average accuracy from 34.4% to 35.5% and Pass@3 from 48.8% to 52.2%. Figure 2 shows the corresponding learning curves, where BSR generally maintains an advantage over the baseline throughout training. Overall, these results indicate that behavioral reward refinement improves both average performance and the likelihood of sampling successful solutions across model scales.

Question: How does BSR change the agent's behavior? BSR changes the agent's behavior most clearly in the investigation and verification stages. As shown in Table 2, BSR more often reaches a reference-relevant file before editing, converts search results into downstream edits more frequently, and performs more post-edit testing. These gains come with



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

Figure 2: Evaluation on SWE-Bench Verified while training Qwen3-14B and Qwen3-32B. We evaluate with temperature = 1 and max_steps = 50 for 3 runs.

Statistic	Baseline	BSR (Ours)
Process metrics		
Edited file viewed before edit ↑	**0.385**	0.362
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**
Edit localization precision ↑	**0.314**	0.300
Selected workload statistics		
Average number of searches	4.74	**4.06**
Average number of tests	0.747	**0.797**
Average number of errors	13.38	**12.76**

Table 2: Selected trajectory behavior statistics over the full set of trajectories. Arrows indicate the preferred direction for process metrics. Full statistics and metric definitions are provided in Appendix B.3.

fewer searches, fewer error-producing actions, and shorter trajectory runtime, suggesting that BSR reduces unproductive exploration while improving the use of gathered evidence. The effect is not uniformly positive: the baseline remains stronger on edited-file inspection before modification and edit localization precision. **Overall, BSR shifts the agent toward a more effective investigate–read–edit–test workflow, with stronger early localization and verification but slightly weaker edit-level precision.** We provide the full set of trajectory behavior statistics and metric definitions in Appendix B.3.

Question: Any interesting observations or tips? For the 14B model, the fraction of trajectories terminating because they reach the maximum number of steps increases during training, while the truncation rate initially decreases. In the later stages, however, the truncation rate begins to rise again. We also observe that the entropy loss continues to increase throughout training. In practice, reducing the maximum number of steps is important for controlling training cost. Lowering the sampling temperature or adding KL regularization both lead to worse performance in our experiments. We lowered the maximum number of steps for training rollouts from 50 to 40 steps, because we find that more than 95% of success trajectories finish in less than 40 steps. Changing this improved performance and also speeds up training. Evaluation continues to use a maximum of 50 steps.

4.2 Ablation: The Effect of Deadzone and Softness

We ablate the reward-mapping function used to convert pairwise behavioral rankings into auxiliary rewards. As shown in Table 3, removing both the deadzone and softness gives the weakest final performance, reaching 32.0% best average accuracy and 45.6% pass@3. Across training checkpoints, this variant is almost always worse than the two variants that use a deadzone, indicating that directly using all ranking differences introduces noisy reward signal. Adding the deadzone improves best average accuracy to 32.6%, suggesting that suppressing small and ambiguous preference differences is important. Compared with the full method, removing softness has a smaller effect on average accuracy, but its pass@3 is worse later in training, suggesting that the soft bounded mapping helps stabilize the stronger candidates selected by multiple samples. The full BSR mapping performs best overall, reaching 33.2% best average accuracy and 46.6% pass@3. The full training curves are included in Appendix 8.

Reward mapping	Avg.	Pass@3
No deadzone, no softness	32.0	45.6
Deadzone, no softness	32.6	44.8
Deadzone + softness **(BSR)**	**33.2**	**46.6**

Table 3: Ablation on the reward-mapping design. Best average accuracy and pass@3 are reported in percentages.

4.3 Ablation: The Effect of Training Group

We also ablate which verifier-outcome bucket receives the BSR signal. As shown in Table 4, applying BSR only to the fail bucket performs best, reaching 33.2% best average accuracy and 46.6% Pass@3. Applying BSR to both the pass and fail buckets achieves a similar best average accuracy of 33.1%, but a lower Pass@3 of 44.6%. It also requires additional pairwise comparisons in the pass bucket, increasing judgment cost without improving performance. In contrast, applying BSR only to the pass bucket performs worse, reaching 32.0% best average accuracy and 44.0% Pass@3. These results support the central motivation of BSR: behavioral refinement is most useful among failing trajectories, where binary verifier rewards provide no contrast despite substantial differences in debugging quality. The full training curves are shown in Figure 7 in the appendix.

Training group	Avg.	Pass@3
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 on which verifier-outcome group receives BSR. Best average accuracy and pass@3 are reported in percentages.

4.4 Analysis: The quality of model judgment

We analyze whether the policy's pairwise judgments remain aligned with a stronger external judge during RL training. Every 10 training steps, we randomly sample 200 pairwise trajectory comparisons from the 14B ranking model and re-evaluate the same pairs using Kimi-2.5 as a proxy reference judge. Figure 3 shows that judgment alignment decreases over training: macro F1 is highest in the early checkpoints and gradually drops as the policy is optimized for solving software-engineering tasks rather than for maintaining calibrated judgment behavior. This result should be interpreted as diagnostic rather than absolute, since Kimi-2.5 is a stronger model but not a ground-truth oracle for trajectory quality. Nevertheless, the alignment remains substantially above random choice throughout training, indicating that the model continues to provide meaningful pairwise signal even as judgment calibration degrades. Importantly, this degradation does not invalidate the reward-refinement method: BSR still improves the trained policy despite relying on an imperfect self-judge. A straightforward way to remove this moving-judge effect would be to serve a separate frozen judge model throughout training, but this would require additional model serving and increase the system cost. We therefore use self-judgment as a lightweight reward-refinement mechanism and treat the alignment analysis as evidence about the reliability and limitations of this design.

Random reward control. To test whether the gains come from meaningful judgment signal rather than simply adding reward variance, we compare BSR against a random-reward control. In this variant, trajectories receive random auxiliary rewards sampled within the same amplitude range as BSR, but the rewards are not tied to pairwise behavioral ranking.

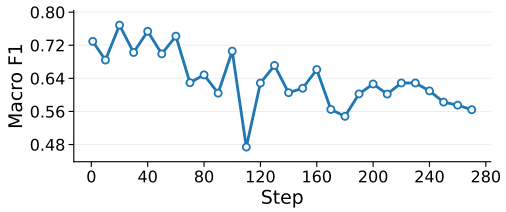


Figure 3: Judgment alignment over training. Macro F1 against Kimi-2.5 decreases but remains above random.

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.

As shown in Table 5, random rewards do not improve over the baseline and slightly reduce Pass@3. In contrast, BSR improves both average accuracy and Pass@3, suggesting that the benefit comes from rank-structured behavioral signal rather than from injecting arbitrary reward noise.

5 Related Work

Verifiable Rewards for Agentic RL. Reinforcement learning with verifiable rewards (RLVR) trains language models using task-specific automatic checkers rather than learned preference models. In mathematical reasoning, coding, and repository-level software engineering, such verifiers can often be constructed from exact answers, unit tests, execution results, or issue-specific validation suites. PPO provides the standard clipped policy-gradient foundation for many language-model RL methods (Schulman et al., 2017). GRPO removes the learned critic and normalizes rewards within groups of samples from the same prompt (Shao et al., 2024), while GSPO moves the optimization to the sequence level for more stable large-scale RL (Zheng et al., 2025). DeepSeek-R1 further demonstrated that verifiable rewards can induce strong reasoning behavior at scale (Guo et al., 2025). For long-horizon interactive agents, LOOP combines PPO-style clipping with a leave-one-out group baseline and no value network (Chen et al., 2025a). These optimizers are highly compatible with executable software-engineering tasks, but they inherit a common limitation: when all rollouts in a group receive the same verifier outcome, group-relative reward differences collapse. Our method is orthogonal to these optimizers. It modifies the scalar reward supplied to the optimizer while preserving the executable verifier as the primary reward source.

Repository-Level Software Engineering Agents. Software engineering has emerged as a natural domain for agentic language models because repository-level tasks require tool use, code navigation, multi-step planning, and executable validation. SWE-bench established real-world GitHub issue resolution as a benchmark for language-model software-engineering agents (Jimenez et al., 2024). Subsequent systems such as SWE-agent, AutoCodeRover, Agentless, and OpenHands explored different agent-computer interfaces, code-search pipelines, localization strategies, repair workflows, and generalist developer-agent platforms (Yang et al., 2024; Zhang et al., 2024; Xia et al., 2024; Wang et al., 2025a). CodePlan studies repository-level coding as a planning problem over interdependent edits (Bairi et al., 2024), while LocAgent improves code localization with graph-guided repository navigation (Chen et al., 2025c). Recent systems further scale inference and experience use: CodeMonkeys studies serial and parallel test-time compute for SWE-bench-style issue resolution (Ehrlich et al., 2025); Satori-SWE uses evolutionary test-time scaling for sample-efficient SWE solving (Zeng et al., 2025); SWE-Exp stores and reuses repair experience across issues (Chen et al., 2025b); and CWM releases an open-weights code model trained with world-model-style interaction traces and multi-task RL in coding and software-engineering environments (Copet et al., 2025). Complementing benchmark evaluation, Copilot Arena evaluates code LLMs through pairwise user preferences collected inside a developer environment (Chi et al., 2025). These works improve agents, interfaces, localization, memory, or inference-time search; our focus is different: extracting a training signal from verifier-equivalent trajectories during RL.

Executable Environments for SWE Agents. A parallel line of work builds executable environments and datasets for training and evaluating software-engineering agents. R2E turns arbitrary GitHub repositories into programming-agent environments (Jain et al., 2025). SWE-Gym constructs executable real-world SWE tasks for training agents and verifiers (Pan et al., 2024), and R2E-Gym scales procedural environment construction while combining execution-based and execution-free verification (Jain et al., 2025). debug-gym provides a text-based environment for interactive debugging (Yuan et al., 2025), and RefactorBench evaluates stateful reasoning in multi-file refactoring tasks (Gautam et al., 2025). SWE-rebench builds an automated pipeline for task collection and decontaminated evaluation (Badertdinov et al., 2026); SWE-smith scales synthetic task generation for SWE-agent training data (Yang et al., 2026); and GSO introduces challenging software-optimization tasks where agents must improve runtime efficiency under performance tests (Shetty et al., 2026). Surveys of LLMs for software engineering summarize the broader landscape of models, tasks, benchmarks, and evaluation practices (Zhang et al., 2026; Hou et al., 2024). These works supply the environments and evaluation settings in which verifier-based RL is possible. Our contribution is a reward-refinement layer that can be applied when these environments expose only sparse or low-distinguishability verifier outcomes.

Training SWE Agents with Reinforcement Learning. Recent work has begun to train software-engineering agents directly rather than relying only on prompting or supervised fine-tuning. SWE-RL scales RL to open software evolution data using rule-based rewards derived from developer changes (Wei et al., 2026). Long-context, multi-turn SWE RL trains agents in interactive, stateful environments where the repository state changes after each action (Golubev et al., 2025). Self-play SWE-RL further reduces dependence on human-written issues and tests by training an agent to inject and repair bugs in sandboxed repositories (Wei et al., 2025). Other work improves verification rather than policy optimization: SWE-RM provides execution-free reward-model feedback for SWE agents (KaShun Shum et al., 2025); Agentic Rubrics constructs codebase-grounded rubrics as contextual verifiers (Raghavendra et al., 2026); and R2E-Gym studies hybrid execution-based and execution-free verification (Jain et al., 2025). These methods address data, environments, or verifier design. Our method is complementary: it does not replace tests with a learned verifier or reward model. Instead, it keeps test success dominant and uses behavioral judgment only to rank trajectories that the executable verifier already treats as equivalent.

Process Supervision and Trajectory Preferences. Our reward construction is related to process supervision, preference-based learning, and ranking-based reward aggregation. Process supervision rewards intermediate reasoning quality rather than only final correctness; in mathematical reasoning, step-level process supervision can outperform outcome-only supervision (Lightman et al., 2024). Preference-based methods similarly use comparisons when absolute scalar labels are difficult to define. The Bradley–Terry model provides a classical method for aggregating pairwise comparisons into latent scores (Bradley and Terry, 1952), RankNet popularized pairwise learning-to-rank objectives (Burges et al., 2005), deep RL from human preferences showed that trajectory comparisons can be used to learn reward models (Christiano et al., 2017), and DPO optimizes language models directly from preference pairs (Rafailov et al., 2023). Related agent-safety work uses pairwise trajectory comparisons to train safer multi-step tool use (Agarwal et al., 2026). Self-rewarding and LLM-as-a-judge methods show that language models can provide useful evaluative signals for their own outputs or for outputs from other models (Yuan et al., 2024; Zheng et al., 2023; Liu et al., 2023). Our approach uses preference information in a narrower and more verifier-preserving way: preferences are local to a single task and verifier-outcome bucket, and they are converted into a small bounded auxiliary reward rather than a standalone reward model. Thus, the judge does not replace the executable verifier; it only refines trajectories that the verifier cannot distinguish.

6 Conclusion

We introduced verifier-anchored self-judgment for reinforcement learning of software engineering agents. The method preserves the binary executable verifier as the primary reward

while refining failed trajectories through local trajectory comparisons. By ranking failed attempts from the same task and mapping the result into a bounded auxiliary reward, the method recovers learning signal from all-fail groups that would otherwise be uninformative under group-relative RL. This provides a simple way to make software-engineering RL more sample-efficient without replacing test-based correctness with an unconstrained reward model.

References

Aradhye Agarwal, Gurdit Siyan, Yash Pandya, Joykirat Singh, Akshay Nambi, and Ahmed Awadallah. Learning when to act or refuse: Guarding agentic reasoning models for safe multi-step tool use. *arXiv preprint arXiv:2603.03205*, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. *arXiv preprint arXiv:2212.08073*, 2022.

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.

Yurui Chang, Yiran Wu, Qingyun Wu, and Lu Lin. Memcollab: Cross-agent memory collaboration via contrastive trajectory distillation. *arXiv e-prints*, pages arXiv–2603, 2026.

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.

Silin Chen, Shaoxin Lin, Yuling Shi, Heng Lian, Xiaodong Gu, Longfei Yun, Dong Chen, Lin Cao, Jiyang Liu, Nu Xia, et al. Swe-exp: Experience-driven software issue resolution. *arXiv preprint arXiv:2507.23361*, 2025b.

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, 2025c.

Wayne Chi, Valerie Chen, Anastasios Nikolas Angelopoulos, Wei-Lin Chiang, Aditya Mittal, Naman Jain, Tianjun Zhang, Ion Stoica, Chris Donahue, and Ameet Talwalkar. Copilot arena: A platform for code llm evaluation in the wild. *arXiv preprint arXiv:2502.09328*, 2025.

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.

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

Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian. Refactorbench: Evaluating stateful reasoning in language agents through code. In *International Conference on Learning Representations*, volume 2025, pages 43131–43162, 2025.

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.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

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.

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.

Binyuan Hui KaShun Shum, Jiawei Chen, XW Lei Zhang, Jiaxi Yang, Yuzhen Huang, Junyang Lin, and Junxian He. Swe-rm: Execution-free feedback for software engineering agents. *arXiv preprint arXiv:2512.21919*, 2025.

Yifei Li, Zeqi Lin, Shizhuo Zhang, Qiang Fu, Bei Chen, Jian-Guang Lou, and Weizhu Chen. Making language models better reasoners with step-aware verifier. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*, pages 5315–5333, 2023.

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.

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.

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.

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.

Manish Shetty, Naman Jain, Jinjian Liu, Vijay Kethanaboyina, Koushik Sen, and Ion Stoica. Gso: Challenging software optimization tasks for evaluating swe-agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

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.

Peiyi Wang, Lei Li, Zhihong Shao, Runxin Xu, Damai Dai, Yifei Li, Deli Chen, Yu Wu, and Zhifang Sui. Math-shepherd: Verify and reinforce llms step-by-step without human annotations. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9426–9439, 2024.

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.

Yiran Wu, Jiale Liu, Jieyu Zhang, Yaolun Zhang, Shilong Liu, Chi Wang, Mengdi Wang, Huazheng Wang, and Qingyun Wu. Position: Digital agents require unified agent-native environments. In *Forty-third International Conference on Machine Learning Position Paper Track*.

Yiran Wu, Tianwei Yue, Shaokun Zhang, Chi Wang, and Qingyun Wu. Stateflow: Enhancing llm task-solving through state-driven workflows. *arXiv preprint arXiv:2403.11322*, 2024.

Yiran Wu, Mauricio Velazco, Andrew Zhao, Manuel Raúl Meléndez Luján, Srisuma Movva, Yogesh K Roy, Quang Nguyen, Roberto Rodriguez, Qingyun Wu, Michael Albada, et al. Excytin-bench: Evaluating llm agents on cyber threat investigation. *arXiv preprint arXiv:2507.14201*, 2025.

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.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

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.

Xingdi Yuan, Morgane M Moss, Charbel El Feghali, Chinmay Singh, Darya Moldavskaya, Drew MacPhee, Lucas Caccia, Matheus Pereira, Minseon Kim, Alessandro Sordoni, et al. debug-gym: A text-based environment for interactive debugging. *arXiv preprint arXiv:2503.21557*, 2025.

Guangtao Zeng, Maohao Shen, Delin Chen, Zhenting Qi, Subhro Das, Dan Gutfreund, David Cox, Gregory Wornell, Wei Lu, Zhang-Wei Hong, et al. Satori-swe: Evolutionary test-time scaling for sample-efficient software engineering. *arXiv preprint arXiv:2505.23604*, 2025.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Shengcheng Yu, Weisong Sun, Yun Yang, and Zhenyu Chen. A survey on large language models for software engineering. *Science China Information Sciences*, 69(4):141102, 2026.

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.

Andrew Zhao, Yiran Wu, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. *Advances in Neural Information Processing Systems*, 38:105816–105879, 2026.

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 10 and 11). 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 12 and an example ranking response in Figure 13.

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

In the training, we follow the RLLM to setup our training. We use the filtered R2E-Gym dataset (4,578 tasks) for training and SWE-Bench Verified (500 tasks) as the evaluation benchmark. We periodically evaluate all methods on SWE-Bench Verified throughout training. For each predefined training budget, we report the checkpoint with the highest SWE-Bench Verified average accuracy within that budget. The same evaluation schedule and checkpoint-selection procedure are applied to all methods. We do not include an entropy regularization term in the training objective. Further, trajectories that reach max context, max steps, or timeout are masked out and are not used for update. See Table 7 for used hyperparameters. Training rollouts use a maximum of 40 agent steps, while SWE-Bench Verified evaluation uses a maximum of 50 steps.

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

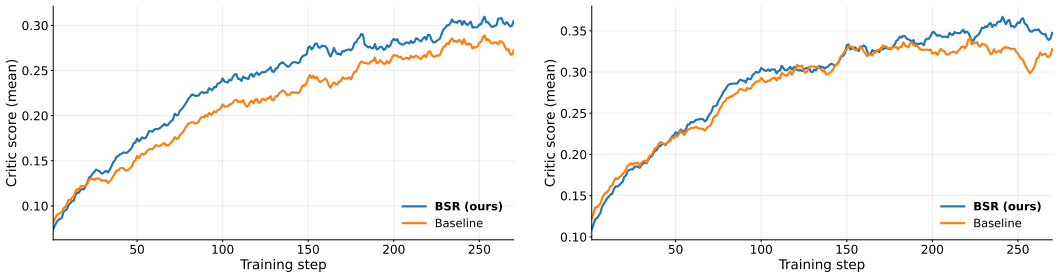
Kubernetes setting Environment provisioning is a major systems challenge for SWE-agent training, since each optimization step may require 512 concurrent environments (64 tasks $\times$ 8 rollouts). In our experiments, we manually configured a two-node Kubernetes cluster, with each node providing 128 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.

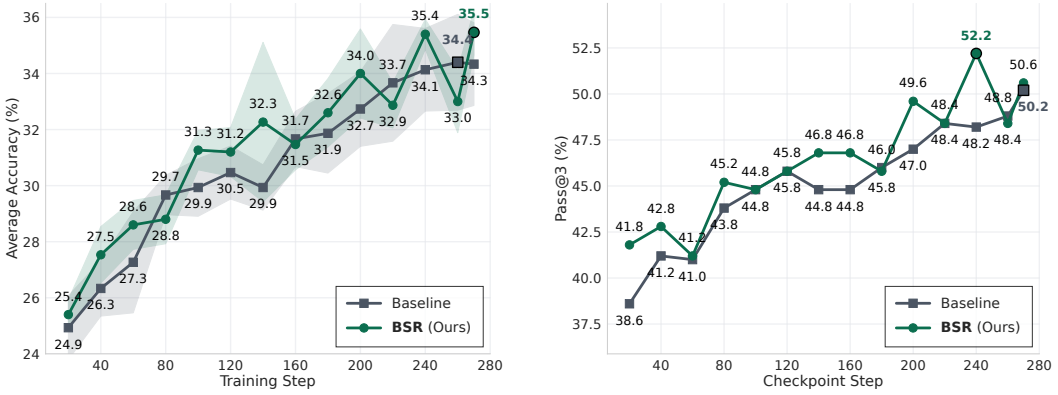
Additional setup instructions and configuration examples are included in our codebase.

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.



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

Figure 4: Mean critic scores on training set for BSR and the baseline. Curves are smoothed with EMA with a span of 31.

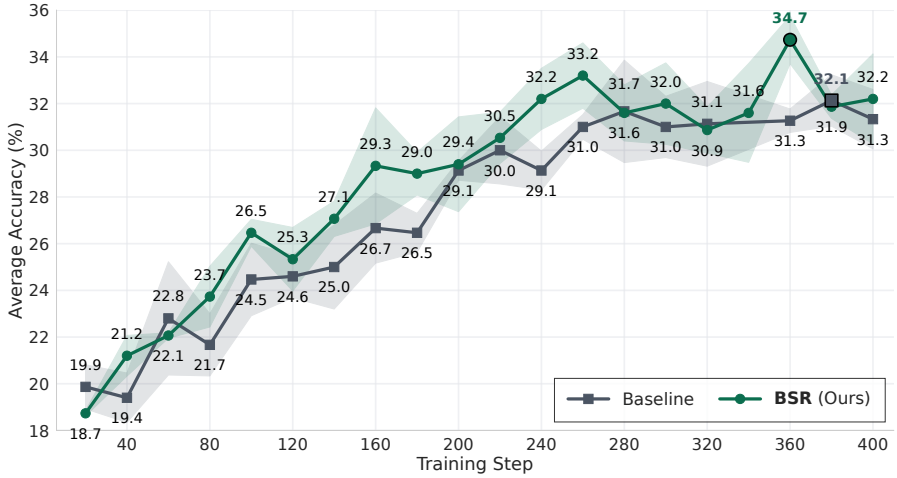


(a) Average Accuracy. (b) Pass@3.

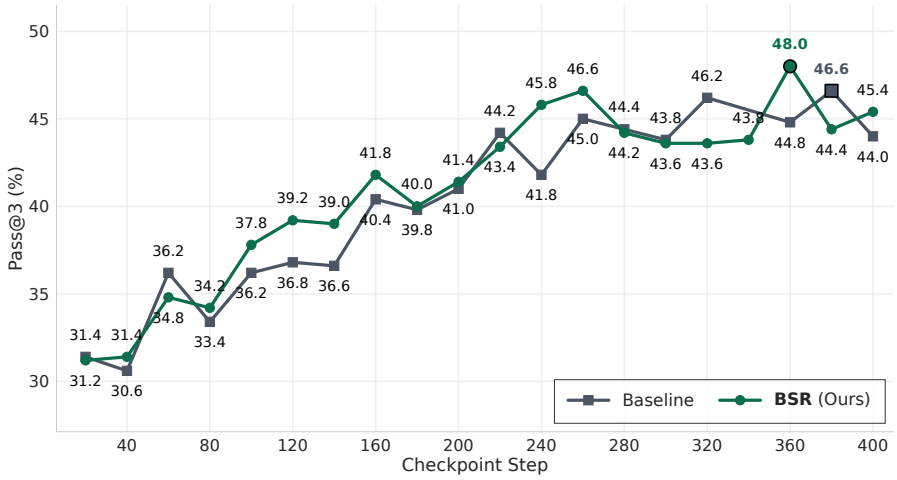
Figure 5: Detailed plot of evaluation on SWE-Bench Verified while training Qwen3-32B. We evaluate with temperature = 1 and max_steps = 50 for 3 runs.

B.2 Additional Results

We provide additional training curves in this section. Figure 4 shows the mean scores during training for both Qwen3-14B and Qwen3-32B. Figure 6 presents the full 400-step curves on SWE-Bench Verified for Qwen3-14B, while Figure 5 shows the detailed training curves for Qwen3-32B. Figures 7 and 8 provide the full results on SWE-Bench Verified for the verifier-outcome-group and reward-mapping ablations, respectively. [change plot]

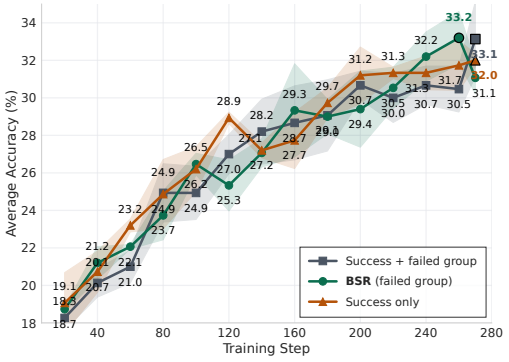


(a) Average Accuracy.

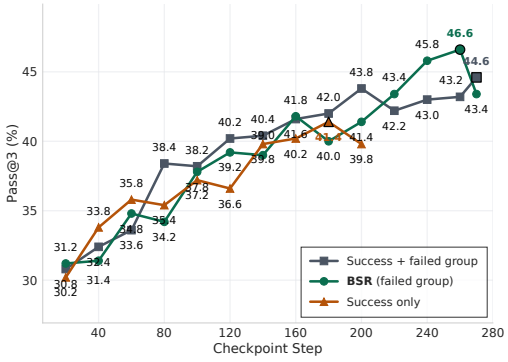


(b) Pass@3.

Figure 6: 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. At step 360, BSR reaches a high average accuracy of 34.7%, while the maximum score for baseline is 32.1%.

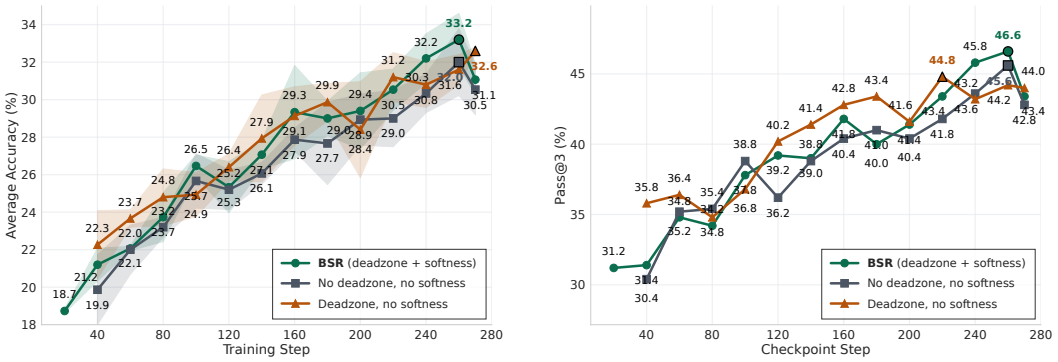


(a) Average accuracy.



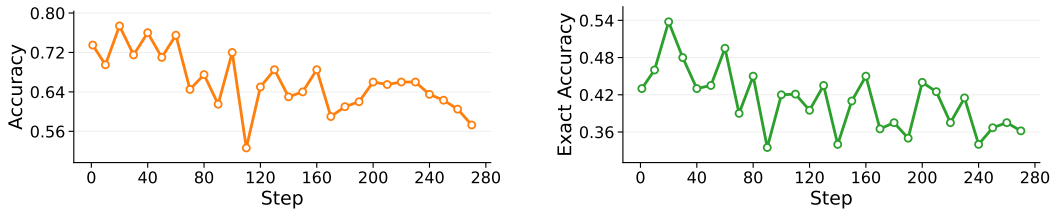
(b) Pass@3.

Figure 7: Training-group 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 8: 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.



(a) Overall accuracy. (b) Exact accuracy.

Figure 9: Additional judgment-alignment metrics over training. We compare the 14B ranking model against Kimi-2.5 on the same sampled pairwise comparisons used in Figure 3. Both metrics show the same general trend as macro F1: alignment decreases during task-oriented RL training but remains above random.

Finally, Figure 9 reports additional judgment-alignment metrics over training, complementing the macro-F1 analysis in Figure 3.

B.3 Trajectory Behavior Statistics

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. *Edited file viewed before edit* measures whether an edited file was inspected before modification. *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, and *edit localization precision* measures the fraction of edited files that overlap with the reference patch. We treat these process metrics as the primary behavioral indicators, while workload statistics provide supporting context.

Behavioral Differences. Table 8 shows that BSR improves several process metrics associated with investigation and verification. Compared with the baseline, BSR more often

Statistic	Baseline	BSR (Ours)
Process metrics		
Edited file viewed before edit ↑	**0.385**	0.362
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**
Edit localization precision ↑	**0.314**	0.300
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 8: 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.

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

Tradeoffs. The improvement is not uniform across all metrics. The baseline is stronger on *edited file viewed before edit* (0.385 vs. 0.362) and *edit localization precision* (0.314 vs. 0.300), suggesting somewhat tighter edit-level grounding and file-level targeting. Thus, BSR appears to improve early localization, search follow-through, and verification, while slightly weakening some measures of edit precision.

Conclusion. Overall, BSR changes the agent's behavior toward a more effective investigate–read–edit–test workflow: it reduces unproductive exploration, improves reference-file discovery and search-to-edit follow-through, and increases post-edit verification. Although the baseline remains stronger on some edit-level grounding metrics, the aggregate evidence suggests that BSR improves the overall debugging process rather than merely shortening trajectories.

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 10: System Prompt for Success Group Ranking.

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 11: System Prompt for Fail Group Ranking.

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 12: User Prompt for ranking.

<think>
Okay, let's compare the two trajectories. Both agents tried to fix the issue where an empty NPY_BLAS_ORDER environment variable caused a build failure. The correct fix was to modify the _parse_env_order function to handle empty strings by returning the base order with an empty unknown list.

Looking at Trajectory A: The agent started by searching for "_parse_env_order" and then viewed various parts of the system_info.py file. They viewed multiple sections, which might have been a bit redundant. Then, they made an edit to the _parse_env_order function, adding a check for an empty string. They created a test script and ran it, which worked. Their stats show some repeated actions and views, but they did eventually make the correct edit.

Trajectory B: The agent directly viewed the system_info.py file, found the _parse_env_order function using search, and then viewed the relevant sections. They made the same edit as A but with fewer steps. Their stats show fewer repeated actions and only one edit without prior view, which might be a minor issue. They also ran the test script once after the edit.

Both made the correct edit, but Trajectory B was more efficient. However, the key is that both failed to solve the task (though in reality, their edits worked). Wait, the problem says both ended up with wrong patches that failed tests. But according to the execution outputs, when they ran the test script, the output was correct (order and unknown were as expected). Maybe the actual test suite had more cases, but the user's example code passed.

But according to the problem statement, both trajectories failed. However, in their execution outputs, the test script ran successfully. Maybe the actual tests they were supposed to pass failed, but the example given here worked.

Assuming that the actual tests failed for both, but the example worked. However, the user's problem says both ended with wrong patches. So, perhaps the agents didn't fully address the issue.

But looking at the code changes, both added the check for empty string. So why did they fail? Maybe the test cases they ran were not comprehensive. However, the problem states that both failed.

In terms of debugging behavior, both agents found the relevant function and made the correct edit. But Trajectory B did it with fewer steps and less redundant viewing. Trajectory A viewed many parts of the file unnecessarily. So Trajectory B's approach was more efficient and direct.

Therefore, Trajectory B is better because it made the correct edit with fewer steps and less redundancy, even though both failed. But since the example test passed, maybe the actual issue was fixed, but other tests failed. However, the problem states they both failed.

But according to the rubric, for wrong-vs-wrong, prefer the one that made progress closer to the real bug. Both made the same edit. However, Trajectory B had fewer repeated actions and more efficient steps. So B is better.
</think>

Winner: B
Strength: 2

Reasoning:
Both trajectories applied the same correct fix (adding a check for empty strings in `_parse_env_order`), but **Trajectory B** demonstrated more efficient and focused debugging behavior:
1. **Progress toward the real bug:** Both agents correctly identified the `_parse_env_order` function as the root cause.
2. **Evidence gathering:** Trajectory B directly searched for `_parse_env_order` and viewed its implementation immediately, avoiding redundant file browsing (e.g., Trajectory A viewed 10+ sections of the file without clear purpose).
3. **Directional edits:** Both made the same correct edit, but Trajectory B did so with fewer steps and no repeated actions.
4. **Relevance of search path:** Trajectory B avoided unnecessary exploration (e.g., viewing unrelated code sections) and focused on the function in question.
5. **Failure recoverability:** Both tested their fix, but Trajectory B used fewer bash runs and had no repeated actions, indicating more streamlined problem-solving.

While both failed to fully resolve the task, **Trajectory B's efficiency and focus** make it the stronger choice.

Figure 13: Example Ranking Response from Qwen3-14B.