SWE-AAAV: BUILDING ADAPTIVE EXECUTABLE EVIDENCE FOR CODING AGENT REWARDS

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models have made rapid progress on software engineering tasks, where pre-defined unit tests are widely used as verifiable rewards for training coding agents. However, limited coverage and implementation-specific assertions can cause them to accept incorrect solutions or reject valid ones, and such misclassifications are difficult to eliminate at scale. In this paper, we propose SWE-AaaV, an Agent-as-a-Verifier framework that constructs adaptive executable evidence for coding agent rewards. Rather than directly judging patch correctness, it generates paired test scripts conditioned on the candidate and reference solutions. To reduce false negatives, the paired scripts accommodate implementation differences while testing the same problem-scoped behavior. To uncover false positives, the verifier agent uses white-box differential testing to find issue-relevant failures and regressions, with the reference solution serving as a behavioral oracle. A deterministic harness checks reference validity, base consistency, and resolved evidence, returning feedback for test revision. For validated evidence, execution on the candidate determines the final reward. To make this candidate-conditioned verification practical for reinforcement learning (RL), we integrate it with asynchronous reward computation and hybrid scheduling. In offline evaluation, SWE-AaaV improves verifier AUC by 4.55–5.98 points over the strongest model-based baselines. When used to provide RL rewards, SWE-AaaV improves SWE-bench Verified performance by 1.8 points over standard unit-test rewards, with only a 15% increase in wall-clock training time. These results demonstrate a practical approach to evaluation-time scaling for coding agent rewards.

1 INTRODUCTION

Large language models (LLMs) have rapidly advanced on software engineering (SWE) tasks, especially in coding agents capable of navigating repositories, editing files, and running tests (Wang et al., 2025b). A major driver of this progress is the use of unit tests as verifiable rewards (DeepSeek-AI, 2025): executing a *pre-defined test suite* against a model-generated patch provides rule-based feedback for rejection-sampling finetuning and reinforcement learning (RL). However, unit tests are faithful rewards only when they are faithful task verifiers (Qwen Team, 2026). In SWE datasets and benchmarks, the verifier usually comes from the test suite associated with a merged pull request (Jimenez et al., 2024). These tests are written to validate a specific change, rather than to define an implementation-agnostic criterion for solving the task (Huang et al., 2026; OpenAI, 2026a). Coverage gaps and implementation-specific assumptions persist when the same suite is reused across candidate solutions (Yu et al., 2025). The resulting reward errors can encourage models to exploit weaknesses in the tests instead of fixing the underlying issue (Whitfill et al., 2026).

Verifier misclassification in SWE tasks falls into two categories. *False negatives* occur when valid solutions are rejected. In OpenAI's manual review of SWE-bench instances (Jimenez et al., 2024), 61.1% were flagged for unit tests that may unfairly mark valid solutions as incorrect (Chowdhury et al., 2024). These errors arise from both narrow tests, which enforce implementation-specific details, and wide tests, which require behavior not specified in the problem description (OpenAI, 2026b). *False positives* occur when incorrect solutions are accepted since the tests do not cover their defects (Wang et al., 2026). DeepSWE (Huang et al., 2026) uses LLM-assisted analysis and extensive human review to check that accepted solutions are functionally correct. Careful benchmark curation

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.

can reduce these misclassifications, but large-scale SWE training datasets may not receive the same level of review (Badertdinov et al., 2025).

Recent work has explored execution-free verifiers to reduce dependence on pre-defined test suites. They include LLM-as-a-Judge (Zhuo, 2024; Tong & Zhang, 2024), trained critic models (Wang, 2025) and generative reward models (GenRMs) (Pan et al., 2025; Jain et al., 2025; Shum et al., 2026; Xu et al., 2025), rubric-based graders (Gunjal et al., 2026; Raghavendra et al., 2026), and the search-only agentic verifier (Zeng et al., 2026). However, many of these verifiers see only serialized surface-level information in the prompt without inspecting the repository. More importantly, the grading process depends entirely on model outputs without execution grounding. Even when calibrated, they often rely on supervised signals from the same imperfect unit-test verifiers (Shum et al., 2026).

In this paper, we propose SWE-AaaV, an Agent-as-a-Verifier framework that builds adaptive executable evidence for coding agent rewards. Its central design separates evidence construction from evidence acceptance. SWE-AaaV introduces a verifier agent that constructs paired test scripts after inspecting the candidate and reference solutions. The scripts test the same behavior required by the issue, but their setup and invocation can differ to accommodate alternative implementations. This flexibility helps avoid false negatives caused by implementation-specific assertions. To reduce false positives, the verifier agent performs white-box differential testing to find issue-relevant failures and regressions that identify behavioral discrepancies. A deterministic harness checks that the script passes on the reference, that both scripts agree on the base implementation, and that a positive result includes fail-to-pass evidence. These execution conditions guide iterative test refinement and determine whether the evidence can be used for reward assignment. For admissible evidence, the reward is determined by execution, not by verbal judgment.

We further make SWE-AaaV feasible as a training-time reward for RL. Unlike a pre-defined test suite, candidate-conditioned verification introduces per-rollout computation for repository inspection, test construction, and execution. A naive inline reward computation would therefore stall rollouts and make RL impractical. To address this, we implement asynchronous reward computation within a typical RL framework (Wang et al., 2025a), allowing workers to start new rollouts while SWE-AaaV verifies completed ones. We also adopt the hybrid scheduling strategy (MiniMax, 2026a) to further mitigate long-tail delays for training throughput.

We evaluate SWE-AaaV as a standalone verifier and as an RL reward. On the offline benchmark, it outperforms the strongest model-based baselines by 4.55–5.98 AUC points, with the largest reductions in false positive rate. Manual analysis of disagreements shows that SWE-AaaV can also identify misclassifications made by pre-defined test suites. When used as a training-time reward for RL on Qwen3-30B-A3B-Instruct (Qwen Team, 2025), SWE-AaaV improves SWE-bench Verified performance by 1.8 points over the standard unit-test reward, while the infrastructure optimizations limit the training-time increase to 15%. The results point to a practical approach to evaluation-time scaling for coding agent rewards: as SWE capabilities improve, the LLM itself can serve as a more powerful verifier without requiring additional training.

The main contributions of this paper are outlined as follows:

- We propose SWE-AaaV, an Agent-as-a-Verifier framework that constructs adaptive executable evidence through paired test generation and white-box differential testing.
- We integrate SWE-AaaV into SWE RL through asynchronous reward computation, enabling per-rollout agentic verification without substantially stalling rollouts.
- SWE-AaaV outperforms the strongest model-based baselines by 4.55–5.98 AUC points and improves resolved rate by 1.8 points over standard unit-test rewards, while limiting the increase in wall-clock training time to 15%.

2 Related Work

Verifiers for SWE tasks. Although the community has widely used data filtering and augmentation to improve the reliability of SWE rewards (Yu et al., 2025; Zhao et al., 2026; Chen et al., 2026), pre-defined test suites cannot eliminate structural defects in the construction of large-scale training instances. Another line of work studies execution-free verifiers, evolving from patch similarity (Wei et al., 2025) to model-based scoring. Beyond the prompt-based LLM-as-a-Judge methods (Zhuo,

2024; Tong & Zhang, 2024), critic models (Wang, 2025) and GenRMs (Pan et al., 2025; Jain et al., 2025; Shum et al., 2026; Xu et al., 2025) are trained and calibrated on binary signals from test suites to produce continuous scores. Agentic Rubrics uses a rubric agent to explore task-specific solution criteria (Raghavendra et al., 2026), while Dockerless employs sub-agents to gather code context from the repository (Zeng et al., 2026). In contrast, SWE-AaaV retains execution as the source of the final reward while using a verifier agent to construct executable evidence.

Agentic reward models. Equipping reward models with agentic capabilities is an emerging direction for evaluation. For factual and computational tasks, reliable judgments often require access to retrieval or execution tools rather than relying solely on parametric knowledge (Li et al., 2024; Peng et al., 2025). In coding tasks, Agent-as-a-Verifier has been primarily explored for frontend code generation (MiniMax, 2026b), where unconstrained implementations and interactive behaviors are difficult to capture with fixed evaluators (He et al., 2026; Lu et al., 2025). Inspired by these, we adapt this paradigm to SWE tasks to address the limitations of pre-defined test suites.

3 Problem Formulation

Software engineering instance. A standard SWE instance is formulated as

$$x = (I, \mathcal{R}_{base}, \mathcal{R}_{ref}, V_t), \tag{1}$$

where I is the problem statement, $\mathcal{R}_{base}$ is the repository at the base commit before the fix, $\mathcal{R}_{ref}$ is a reference solution, and V_t is a pre-defined test verifier.

Coding agent rollout. The interaction between a policy π_θ and the coding environment is modeled as a partially observable Markov decision process (Kaelbling et al., 1998). A rollout produces

$$z = (\tau, \mathcal{R}_{cand}) \sim \pi_\theta(\cdot \mid I, \mathcal{R}_{base}), \tag{2}$$

where τ is the interaction trajectory and $\mathcal{R}_{cand}$ is the resulting candidate solution.

Verifier. A verifier assigns a correctness score to a rollout. The underlying goal is to determine whether $\mathcal{R}_{cand}$ resolves the issue specified by I. We denote this ideal but inaccessible test oracle by

$$V^\star(I, z) = \mathbb{1}\left[\mathcal{R}_{cand} \text{ correctly resolves } I\right]. \tag{3}$$

Therefore, every computable verifier is a lossy approximation of $V^\star$ (Qwen Team, 2026).

For a decision threshold γ, let $\bar{V}(I, z) = \mathbb{1}\left[V(I, z) \geq \gamma\right]$. A false negative occurs when $\bar{V}(I, z) = 0$ but $V^\star(I, z) = 1$, while a false positive occurs when $\bar{V}(I, z) = 1$ but $V^\star(I, z) = 0$.

Pre-defined test verifier. The standard verifier V_t accepts a candidate solution when all pre-defined tests pass on $\mathcal{R}_{cand}$. SWE instances are commonly constructed to satisfy

$$(\exists\, v \in V_t : v(\mathcal{R}_{base}) = 0) \wedge (\forall\, v \in V_t : v(\mathcal{R}_{ref}) = 1). \tag{4}$$

This fail-to-pass requirement implicitly treats $\mathcal{R}_{ref}$ as a test oracle and V_t as a finite behavioral proxy for $\mathcal{R}_{ref}$. Since V_t is fixed at dataset-construction time, any blind spots or implementation-specific assumptions in the suite may lead to misclassifications across rollouts.

Model-based verifier. A model-based verifier directly predicts a score:

$$V_m(I, z) = f_\phi(I, z; \xi), \tag{5}$$

where ϕ denotes model parameters and ξ denotes optional evaluation-time information. Existing methods differ mainly in their inputs and scoring mechanisms:

- **LLM-as-a-Judge** uses $(I, \mathcal{R}_{cand})$, optionally together with $\mathcal{R}_{ref}$, and extracts a score from the structured model response.
- **Rubric-based grading** generally replaces I with a pre-generated checklist and aggregates the model's judgments over the checklist items.
- **Critic models** primarily condition on the rollout trajectory τ and use a regression head to predict a continuous score.
- **Generative reward models** condition on (I, z) and derive a continuous score from the probability of a designated output token.

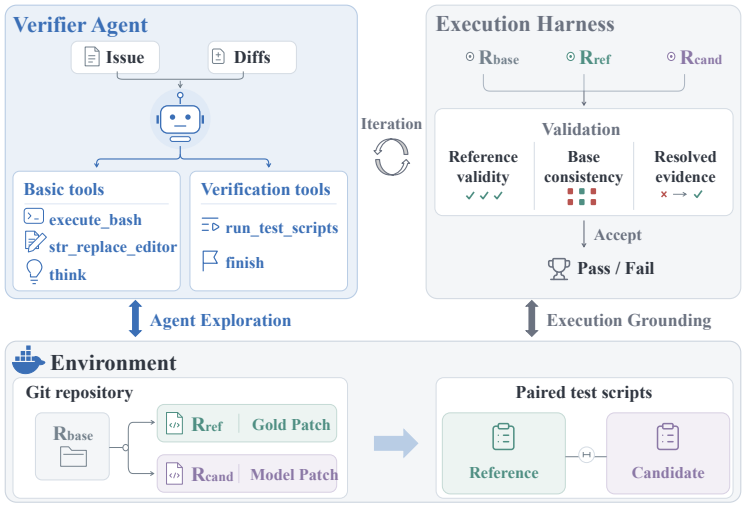


Figure 1: Overview of SWE-AaaV, which explicitly separates evidence construction from acceptance.

Agent-based verifier. Similar to Eq. (5), an agent-based verifier extends V_m from a single model invocation to a multi-turn process. The verifier agent can actively inspect the repository and condition its evaluation on the candidate solution. We view this paradigm as a natural form of evaluation-time scaling for reward computation, allocating additional inference and tool use to verification.

4 METHODOLOGY

Figure 1 shows the two stages of SWE-AaaV: test construction by the verifier agent and validation by a deterministic harness. Paired test scripts accommodate implementation differences, white-box code differences guide the search for informative test inputs, and a deterministic harness checks whether the resulting evidence is admissible. These two stages are iterated until the verifier agent submits evidence accepted by the harness or reaches the maximum number of assistant turns.

4.1 CANDIDATE-CONDITIONED TEST PAIRS

A pre-defined test verifier can reject a valid solution when its setup or invocation assumes implementation details that are not required by the problem statement. To accommodate alternative implementations, our verifier agent constructs a pair of test scripts after observing the candidate:

$$E = (s_{ref}, s_{cand}) = A_\phi\left(I, \mathcal{R}_{base}, \mathcal{R}_{ref}, \mathcal{R}_{cand}\right), \tag{6}$$

where A_ϕ denotes the verifier agent, and s_{ref} and s_{cand} are test scripts for the reference and candidate solutions, respectively. The problem statement I defines the scope of verification, and $\mathcal{R}_{ref}$ provides a behavioral oracle within that scope. The paired scripts are designed to test the same problem-scoped behavior, while allowing implementation-specific setup and invocation to differ. For example, the scripts may invoke different internal helpers while checking the same externally observable outcome.

4.2 EXECUTION-BASED EVIDENCE VALIDATION

Generated test scripts must be checked before they can determine a reward. Let $e(s, \mathcal{R}) \in \{0, 1\}$ denote the binary execution result of script s on repository state $\mathcal{R}$, with $e = 1$ when all tests pass. For a pair E, denote the execution results by

$$\begin{aligned} u &= e(s_{ref}, \mathcal{R}_{ref}), & v &= e(s_{cand}, \mathcal{R}_{cand}), \\ b_{ref} &= e(s_{ref}, \mathcal{R}_{base}), & b_{cand} &= e(s_{cand}, \mathcal{R}_{base}). \end{aligned} \tag{7}$$

The harness imposes three admissibility conditions through the custom `run_test_scripts` tool:

- **Reference validity.** We require $u = 1$: the reference-side script must pass on the behavioral oracle. A failure indicates that the submitted evidence cannot yet be used to evaluate the candidate and must be revised.
- **Base consistency.** We require $b_{ref} = b_{cand}$: the paired scripts must agree when evaluated on the same base implementation. This check detects differences between the scripts that manifest on their common baseline. Base agreement is a consistency check of test results, not a guarantee of semantic equivalence.
- **Resolved evidence.** If $v = 1$, we additionally require $b_{cand} = 0$. A positive reward must therefore include evidence that distinguishes the candidate from the unfixed base. Tests that pass on both cannot alone support a positive reward. This requirement is asymmetric: when $v = 0$, the base may pass, allowing evidence of regressions that appear only after applying the candidate patch.

Together, these conditions define the admissibility predicate

$$\mathcal{H}(E) = \mathbb{1}[u = 1]\,\mathbb{1}[b_{ref} = b_{cand}]\,\mathbb{1}[v = 0 \vee b_{cand} = 0]. \tag{8}$$

For admissible evidence, the verifier returns

$$V_a(I, z) = v, \qquad \text{when } \mathcal{H}(E) = 1. \tag{9}$$

Thus, evidence acceptance and candidate acceptance are distinct: $\mathcal{H}(E) = 0$ requests further evidence refinement, while $\mathcal{H}(E) = 1$ permits either reward according to the candidate execution result. When $\mathcal{H}(E) = 0$, the harness returns a diagnostic observation, including the violated condition and terminal output needed for the agent to refine the scripts. The harness checks execution outcomes but cannot certify issue relevance or complete coverage. An admissible evidence may therefore still miss defects or encode an out-of-scope requirement.

4.3 Adaptive White-box Differential Testing

The verifier agent searches for informative evidence through white-box differential testing. It has access to the complete base, reference, and candidate repository states, with both solutions derived from the same base. The code differences guide hypotheses about issue-relevant behavioral discrepancies and regressions in affected functionality. These hypotheses are instantiated as targeted inputs and assertions in the paired scripts. With the reference solution as a behavioral oracle, the verification task is reduced to the simpler task of constructing informative test inputs.

In addition to the basic interaction tools of the OpenHands scaffold (Wang et al., 2025b), the agent can execute intermediate evidence through the harness and revise it using the resulting diagnostic observation. Violations of reference validity or base consistency direct attention to invalid tests or inconsistent adaptation. A candidate pass without fail-to-pass evidence calls for tests that distinguish the proposed fix from the base. This feedback makes evidence construction an iterative search constrained by executable checks.

Each execution restores the target repository revision, with the test scripts maintained separately from the evaluated code. This allows the evidence to evolve while the evaluated implementations remain fixed. The agent may inspect additional code, revise the scripts, and repeat validation within a fixed assistant-turn budget. Final submission is accepted only when the evidence satisfies $\mathcal{H}(E) = 1$, after which Eq. (9) determines the reward. The process also terminates if the turn budget is exhausted.

5 Asynchronous Reward Computation

In modern RL systems (Wang et al., 2025a), reward computation is commonly treated as part of the rollout pipeline and executed serially after each trajectory is completed. However, candidate-conditioned verification adds repository inspection, test construction, and execution to every completed rollout. Figure 2(a) illustrates the resulting dependency: rollout worker 1 cannot begin τ_5 until verification of τ_3 finishes. A slow verification therefore delays both its own reward and the generation of subsequent trajectories assigned to the same worker.

As shown in Figure 2(b), we move SWE-AaaV off the rollout critical path through **asynchronous reward computation**. Once a rollout completes, its verification job is dispatched to a separate pool

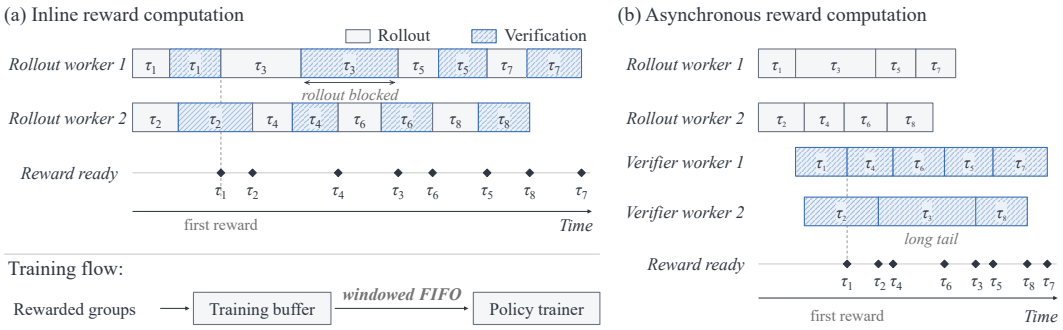


Figure 2: Infrastructure optimizations with asynchronous reward computation and windowed FIFO.

of verifier workers, while the rollout worker immediately proceeds to generate subsequent trajectories. In the figure, rollout worker 1 can generate τ_5 and τ_7 while verification of τ_3 is still in progress. This overlap removes verification-induced gaps from the rollout timelines and allows generation and verification to proceed concurrently.

Long-tail latency also affects the order in which rewards become available. In Figure 2(b), the rewards for τ_4 and τ_6 are ready before that for τ_3, despite being generated later. Under strict FIFO scheduling, a completed group must additionally wait for earlier groups that are still pending. Thus, even with asynchronous verification, the trainer can remain idle while the buffer already contains completed groups. To reduce this additional waiting between the training buffer and policy trainer, we adopt windowed FIFO scheduling (MiniMax, 2026a). An unfinished group still waits for all its rewards, but does not block the consumption of other completed groups in the same window. Ordering remains strict across window boundaries, so a slow group can only be overtaken by groups within its window. Larger windows provide better tolerance to long-tail delays but can increase rollout staleness.

6 Experimental Setup

We evaluate SWE-AaaV from two complementary perspectives. First, we measure verifier quality on an offline benchmark of candidate solutions, comparing against representative model-based verifiers. Second, we use SWE-AaaV as the correctness reward for SWE RL and evaluate whether it leads to a stronger policy model compared to the pre-defined test verifier. Details of benchmark construction, verifier configurations, and training settings are provided in Appendix B.

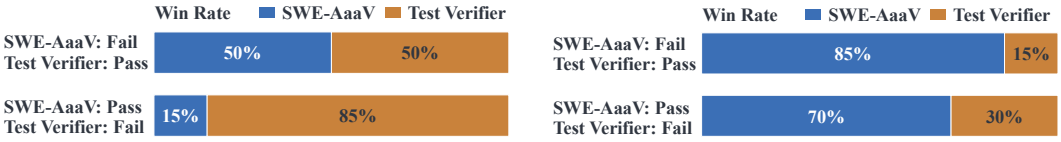
Offline verifier evaluation. We construct an offline verifier benchmark from trajectories sampled on SWE-bench Verified using Qwen3-Coder-30B-A3B (Qwen Team, 2025) with the OpenHands scaffold. We use the binary outcome of the pre-defined unit tests as a proxy label, benefiting from the manual screening of SWE-bench Verified (Chowdhury et al., 2024). These labels remain imperfect, and we examine disagreements through manual adjudication. The resulting benchmark contains 618 examples and is exactly balanced between the two proxy labels.

We compare SWE-AaaV against four representative model-based verifiers. **Critic models** (Wang, 2025) predict correctness from the rollout trajectory using a regression head. **GenRMs** (Pan et al., 2025) use SWE-Gym OpenHands-32B-Verifier, a generative reward model trained for SWE verification. **LLM-as-a-Judge** (Jain et al., 2025) directly scores correctness from the problem statement together with the candidate and reference patches. **Rubric-based grading** (Raghavendra et al., 2026) evaluates candidates using task-specific rubrics. We further evaluate **SWE-AaaV w/o reference**, which removes access to the reference solution. For prompting-based verifiers and SWE-AaaV, we evaluate Qwen3-Coder-30B-A3B, Qwen3.6-35B-A3B, Qwen3.6-Plus (Qwen Team, 2025), and DeepSeek-V4.1-Flash (DeepSeek-AI, 2026).

We report **AUC**, false-negative rate (**FNR**) and false-positive rate (**FPR**). All results are averaged over four runs. When the maximum number of rounds is reached, SWE-AaaV prioritizes the test execution on the candidate solution for reward assignment. If no valid execution result is available, we mark the verification as incorrect. For continuous scores, we select a single decision threshold that maximizes accuracy on the benchmark.

Table 1: Offline verifier evaluation against unit-test proxy labels, reported as percentages (%).

Graders	Methods	AUC↑	FNR↓	FPR↓
OpenHands-32B-Critic	Critic Models	58.62	25.57	55.34
OpenHands-32B-Verifier	GenRMs	70.14	22.65	48.54
Qwen3-Coder-30B-A3B	LLM-as-a-Judge	63.96	12.14	59.95
	Rubric-based grading	**68.72**	25.57	**44.66**
	SWE-AaaV	61.49	**11.00**	66.02
Qwen3.6-35B-A3B	LLM-as-a-Judge	66.99	18.85	47.17
	Rubric-based grading	73.66	16.91	45.47
	SWE-AaaV	**78.56**	**15.13**	**27.75**
Qwen3.6-Plus	LLM-as-a-Judge	75.40	18.77	30.42
	Rubric-based grading	75.87	19.74	37.86
	SWE-AaaV	**80.42**	**18.45**	**20.71**
DeepSeek-V4.1-Flash	LLM-as-a-Judge	79.13	15.67	25.57
	Rubric-based grading	78.67	15.21	36.89
	SWE-AaaV	**85.11**	**13.59**	**16.18**



(a) SWE-AaaV with Qwen3.6-35B-A3B.

(b) SWE-AaaV with DeepSeek-V4.1-Flash.

Figure 3: Manual adjudication of disagreements between the test verifier and SWE-AaaV, with 20 samples per disagreement category for each backbone.

RL training. We evaluate training-time reward quality on SWE-bench Verified. The RL training set contains 6,211 SWE-rebench (Badertdinov et al., 2025) instances after excluding repositories that overlap with evaluation. For the initial policy, we filter 27,634 trajectories from CoderForge-Preview (Ariyak et al., 2026) to finetune Qwen3-30B-A3B-Instruct-2507, excluding repositories used for either RL training or evaluation. We use Qwen3.6-35B-A3B as the verifier backbone.

We compare the standard pre-defined test verifier and SWE-AaaV as sources of correctness rewards. Following SWE-RM (Shum et al., 2026), the trajectory reward is 0 when unfinished, otherwise

$$R = \lambda + (1 - \lambda)\, r_c, \tag{10}$$

where $\lambda = 0.1$ provides a small completion bonus and r_c is the binary correctness reward. We optimize the policy with GSPO (Zheng et al., 2025) without a KL-shaped reward or loss. Both reward settings share the same SFT initialization, training data, rollout budget, and optimizer settings.

7 Results

7.1 Offline Verifier Evaluation

Comparison with model-based verifiers. Table 1 shows that the benefit of agentic verification depends on the backbone. With Qwen3-Coder-30B-A3B, SWE-AaaV reaches only 61.49% AUC and accepts 66.02% of proxy-negative candidates. This permissive behavior suggests that the verifier agent must be capable of identifying relevant failure modes and translating them into effective tests. The trained GenRM achieves 70.14% AUC, outperforming all three verifiers using this backbone.

With the stronger backbones, SWE-AaaV outperforms the model-based baselines by 4.55–5.98 AUC points. The main improvement is in rejecting incorrect candidates, resulting in a reduction in FPR of 9.39 to 17.72 percentage points. This pattern is consistent with stronger backbones producing more discriminative tests through white-box differential testing. The reported results also reflect residual failures in evidence construction, e.g., 3.9% of verification runs with Qwen3.6-Plus terminate without passing harness validation.

Table 2: Ablation results of SWE-AaaV.

Grader	Variants	AUC↑	FNR↓	FPR↓
Qwen3.6-35B-A3B	SWE-AaaV	**78.56**	15.13	**27.75**
	w/o adaptation	74.03	19.74	32.20
	w/o validation	70.67	21.04	37.62
	w/o reference	65.61	**8.09**	60.68
DeepSeek-V4.1-Flash	SWE-AaaV	**85.11**	**13.59**	**16.18**
	w/o adaptation	82.93	15.86	18.28
	w/o validation	82.00	15.45	20.55
	w/o reference	80.22	14.89	24.68

Disagreement analysis with the test verifier. The pre-defined unit-test outcome serves only as a proxy label and is not a fully reliable ground truth. For each of Qwen3.6-35B-A3B and DeepSeek-V4.1-Flash used for SWE-AaaV, we sample 20 disagreements where the unit tests pass and 20 where they fail. Two annotators adjudicate whether the candidate resolves the issue, resolving annotation disagreements through discussion.

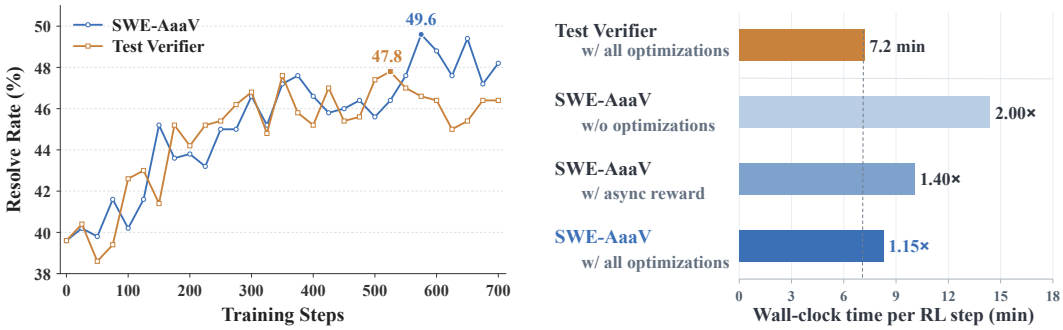
Figure 3 shows a pronounced asymmetry for Qwen3.6-35B-A3B. When unit tests pass but SWE-AaaV rejects the candidate, each verifier is correct in 10 of 20 cases. The unit-test errors reflect uncovered edge cases, incomplete fixes, and regressions, whereas erroneous rejections by SWE-AaaV often arise from out-of-scope assertions or assumptions tied to the reference implementation. When unit tests fail but SWE-AaaV accepts, however, SWE-AaaV is correct in only 3 of 20 cases. The remaining candidates contain unresolved issue behavior or pass-to-fail regressions, showing that adaptive testing can still overlook decisive negative evidence. With DeepSeek-V4.1-Flash, SWE-AaaV is correct in 17 of 20 cases where unit tests pass but SWE-AaaV rejects the candidate, and in 14 of 20 cases where unit tests fail but SWE-AaaV accepts. Thus, this backbone can identify both erroneous acceptances and erroneous rejections by the pre-defined tests in the sampled disagreements. These findings complement the offline results: higher agreement with proxy labels can coexist with correcting some of their errors. These adjudications diagnose how the verifiers fail, but do not estimate either verifier's overall error rate or the prevalence of test-suite defects in the full benchmark.

Representative case studies are provided in Appendix C. The observed failures point to complementary sources of evidence: pre-defined tests provide existing issue and regression checks, while candidate-conditioned tests can probe coverage gaps and accommodate alternative implementations. Exposing the pre-defined tests to the verifier agent is therefore a natural direction for future work.

7.2 Ablation Study

Table 2 examines three components using Qwen3.6-35B-A3B and DeepSeek-V4.1-Flash. SWE-AaaV **without adaptation** requires a single shared test script instead of paired scripts. AUC drops by 4.53 and 2.18 points, respectively, and both error rates increase. The shared script is less able to accommodate valid implementation differences and can also limit the construction of discriminative tests. SWE-AaaV **without validation** removes both the `run_test_scripts` tool and the validation logic attached to `finish`. AUC decreases by 7.89 points for Qwen and 3.11 points for DeepSeek, with FPR increasing by 9.87 and 4.37 points. These results support the role of enforced validation in preventing unreliable evidence from determining rewards. This variant still allows code execution through `execute_bash`. The loss therefore isolates the value of mandatory evidence checks and their feedback, rather than the availability of execution itself. SWE-AaaV **without reference** removes access to the reference solution and produces the largest AUC drop for both backbones. For Qwen, FPR rises sharply from 27.75% to 60.68%, indicating more permissive judgments rather than better discrimination. DeepSeek is more resilient, yet its FPR still increases from 16.18% to 24.68%. Without a behavioral oracle, the agent must infer expected behavior as well as generate informative inputs, reintroducing the test-oracle problem (Barr et al., 2015; Rajan, 2026). Trajectory inspection also reveals reliance on candidate-provided tests or reproduction scripts in this setting.

The smaller ablation losses with DeepSeek-V4.1-Flash suggest that a stronger verifier backbone can partially compensate for missing components (Rajasekaran, 2026), while the full verifier remains the most accurate configuration for both backbones.



(a) RL training curves of resolved rate.

(b) RL training efficiency.

Figure 4: RL comparison of two verifiers, evaluated on SWE-bench Verified.

7.3 RL Results

Figure 4a shows that SWE-AaaV also provides a stronger training signal when used as the correctness reward for SWE RL. Starting from the same SFT policy with a resolved rate of 39.6%, RL with the pre-defined test verifier reaches a best observed rate of 47.8%, while SWE-AaaV reaches 49.6%, a gain of 1.8 percentage points. Both runs make comparable progress early in training, while SWE-AaaV achieves further gains and maintains higher resolved rates through the remaining training steps. This demonstrates that the offline verifier improvements can translate into a stronger policy when directly optimized as a training-time reward. Notably, SWE-AaaV with Qwen3.6-35B-A3B does not yet match the verifier quality achievable through the substantial manual curation behind SWE-bench Verified. However, such curation is difficult to reproduce for large-scale RL data such as SWE-rebench, where verifier defects are much harder to eliminate instance by instance. In this regime, candidate-conditioned executable verification provides a practical way to improve reward quality beyond the inherited pre-defined tests without requiring comparable human effort. Moreover, the verification quality of SWE-AaaV can further benefit from stronger SWE capabilities of its verifier backbone, leading to a promising evaluation-time scaling.

The stronger reward incurs substantially higher verification cost: the verifier agent averages roughly 50 turns per trajectory and involves more execution-oriented tool calls. Figure 4b measures how the two infrastructure optimizations reduce the resulting wall-clock overhead. The test-verifier baseline averages 7.2 minutes per RL step and already uses both asynchronous reward computation and windowed FIFO scheduling (Section 5). Without either optimization, SWE-AaaV requires approximately $2\times$ the baseline's training time, as verification latency limits both rollout and training throughput. Asynchronous reward computation moves verification off the rollout critical path, reducing training time to approximately $1.4\times$ the baseline. Adding windowed FIFO scheduling further reduces this ratio to $1.15\times$ by allowing completed groups to be consumed without waiting for every slow predecessor within the window. These wall-clock gains rely on overlapping rollout generation with verification on a dedicated GPU pool, and do not eliminate the additional inference compute required for verification.

8 Conclusion

In this paper, we propose SWE-AaaV, an Agent-as-a-Verifier framework that constructs adaptive executable evidence for coding agent rewards. By separating flexible evidence construction from deterministic execution-based reward determination, SWE-AaaV enables candidate-conditioned paired test generation and white-box differential testing. We demonstrate that SWE-AaaV improves verifier quality over model-based baselines and provides a stronger training-time reward, yielding a 1.8-point improvement on SWE-bench Verified over standard unit-test rewards. We further introduce asynchronous reward computation and adopt hybrid scheduling to limit the additional wall-clock training cost. More broadly, SWE-AaaV suggests a promising direction for evaluation-time scaling: as coding models become stronger, increasing the computation devoted to verification may provide a practical path toward more reliable and scalable reward signals.

AI USE STATEMENT

In this work, we used generative AI tools for assisting with code implementation. We have not used generative AI tools for developing conceptual frameworks, proposing or refining hypotheses, designing experiments, or interpreting results, and tasks involving mathematical proofs are not applicable to this work. Additionally, we used generative AI tools for assisting with manuscript writing and language editing. We have reviewed all AI-assisted work. Final implementation and writing decisions were made by the authors. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work studies automated verification using existing software-engineering datasets and LLMs. More reliable rewards may improve coding agents, but passing generated tests does not guarantee that a patch is correct or secure. The verifier can inherit errors from reference solutions and miss defects outside the tested behaviors. Practical use should execute generated code in isolated, resource-limited environments and retain independent testing and human review before deployment. Use of the datasets, repositories, and model services should respect their licenses and access terms. Additional verifier inference also incurs computational and environmental costs, and the reported wall-clock savings do not imply lower total compute consumption.

REPRODUCIBILITY STATEMENT

To support reproducibility, Sections 4 and 5 describe the verification procedure, deterministic evidence checks, and RL scheduling strategy. Appendix A provides implementation details of the execution environment and tool interfaces. Section 6 and Appendix B document data sources and filtering, model backbones, benchmark construction, evaluation protocols, and training configurations. The prompts for SWE-AaaV and baselines are provided in Appendix D.

REFERENCES

Anthropic. Claude Sonnet 4.5 system card. https://www.anthropic.com/claude-sonnet-4-5-system-card, 09 2025.

Alpay Ariyak, Junda Zhang, Junxiong Wang, Shang Zhu, Federico Bianchi, Sanjana Srivastava, Ashwinee Panda, Siddhant Bharti, Chenfeng Xu, John Heo, Xiaoxia Shirley Wu, James Zou, Percy Liang, Leon Song, Ce Zhang, Ben Athiwaratkun, Zhongzhu Zhou, and Qingyang Wu. CoderForge-Preview: Sota open dataset for training efficient agents, 02 2026. URL https://www.together.ai/blog/coderforge-preview.

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. In *NeurIPS*, San Diago, CA, USA / Mexico City, Mexico, 2025. Curran Associates, Inc.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Trans. Software Eng.*, 41(5):507–525, 2015.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, Hao Ge, Zongmeng Zhang, Zeyu Cui, Dayiheng Liu, Jingren Zhou, Jianling Sun, Junyang Lin, and Binyuan Hui. SWE-Universe: Scale real-world verifiable environments to millions. *CoRR*, 2602.02361, 2026.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified. https://openai.com/index/introducing-swe-bench-verified/, 08 2024.

DeepSeek-AI. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645(8081):633–638, 2025.

DeepSeek-AI. Deepseek-v4.1-flash: Pushing the limits of kv cache compression. *CoRR*, 2609.19969, 2026.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. In *ICLR*, Rio de Janeiro, Brazil, 2026. OpenReview.net.

Zehai He, Wenyi Hong, Zhen Yang, Ziyang Pan, Mingdao Liu, Xiaotao Gu, and Jie Tang. Vision2Web: A hierarchical benchmark for visual website development with agent verification. In *ICML*, Seoul, South Korea, 2026.

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

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, An Yang, Rui Men, Fei Huang, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report. *CoRR*, 2409.12186, 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. In *COLM*, Montreal, Canada, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world Github issues? In *ICLR*, Vienna, Austria, 2024. OpenReview.net.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *SOSP*, pp. 611–626, Koblenz, Germany, 2023. ACM.

Lei Li, Yekun Chai, Shuohuan Wang, Yu Sun, Hao Tian, Ningyu Zhang, and Hua Wu. Tool-augmented reward modeling. In *ICLR*, Vienna, Austria, 2024. OpenReview.net.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. WebGen-Bench: Evaluating LLMs on generating interactive and functional websites from scratch. In *Neurips*, San Diago, CA, USA / Mexico City, Mexico, 2025. Curran Associates, Inc.

MiniMax. The MiniMax-M2 series: Mini activations unleashing max real-world intelligence. *CoRR*, 2605.26494, 2026a.

MiniMax. Why we built VIBE bench: Rethinking evaluation for real workloads. `https://www.minimax.io/news/why-we-built-vibe-bench-rethinking-evaluation-for-real-work`, 01 2026b.

OpenAI. Separating signal from noise in coding evaluations. `https://openai.com/index/separating-signal-from-noise-coding-evaluations/`, 07 2026a.

OpenAI. Why SWE-bench Verified no longer measures frontier coding capabilities. `https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/`, 02 2026b.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *ICML*, volume 267, Vancouver, BC, Canada, 2025. PMLR / OpenReview.net.

Hao Peng, Yunjia Qi, Xiaozhi Wang, Zijun Yao, Bin Xu, Lei Hou, and Juanzi Li. Agentic reward modeling: Integrating human preferences with verifiable correctness signals for reliable reward systems. In *ACL*, pp. 15934–15949, Vienna, Austria, 2025. Association for Computational Linguistics.

Qwen Team. Qwen3 technical report. *CoRR*, 2505.09388, 2025.

Qwen Team. The verification horizon: No silver bullet for coding agent rewards. *CoRR*, 2606.26300: 1–51, 2026.

Mohit Raghavendra, Anisha Gunjal, Bing Liu, and Yunzhong He. Agentic rubrics as contextual verifiers for SWE agents. In *ACL*, pp. 15265–15290, San Diego, California, USA, 2026. ACL.

Shreshth Rajan. Auditing reward hackability in code rl training environments. *CoRR*, 2606.16062: 1–11, 2026.

Prithvi Rajasekaran. Harness design for long-running application development. `https://www.anthropic.com/engineering/harness-design-long-running-apps`, 03 2026.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *CoRR*, 1909.08053:1–15, 2019.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, X. W., Jiaxi Yang, Yuzhen Huang, Junyang Lin, and Junxian He. SWE-RM: execution-free feedback for software engineering agents. In *ICLR*, Rio de Janeiro, Brazil, 2026. OpenReview.net.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, Zhiming Mao, Xinyu Wang, Lifeng Shang, and Haoli Bai. SWE-Lego: Pushing the limits of supervised fine-tuning for software issue resolving. *CoRR*, 2601.01426:1–20, 2026.

Weixi Tong and Tianyi Zhang. CodeJudge: Evaluating code generation with large language models. In *EMNLP*, pp. 20032–20051, Miami, FL, USA, 2024. ACL.

Weixun Wang, Shaopan Xiong, Gengru Chen, Wei Gao, Sheng Guo, Yancheng He, Ju Huang, Jiaheng Liu, Zhendong Li, Xiaoyang Li, Zichen Liu, Haizhou Zhao, Dakai An, Lunxi Cao, Qiyang Cao, Wanxi Deng, Feilei Du, Yiliang Gu, Jiahe Li, Xiang Li, Mingjie Liu, Yijia Luo, Zihe Liu, Yadao Wang, Pei Wang, Tianyuan Wu, Yanan Wu, Yuheng Zhao, Shuaibing Zhao, Jin Yang, Siran Yang, Yingshui Tan, Huimin Yi, Yuchi Xu, Yujin Yuan, Xingyao Zhang, Lin Qu, Wenbo Su, Wei Wang, Jiamang Wang, and Bo Zheng. Reinforcement learning optimization for large-scale learning: An efficient and user-friendly scaling library. *CoRR*, 2506.06122:1–16, 2025a.

Xingyao Wang. SOTA on SWE-Bench verified with inference-time scaling and critic model. `https://www.openhands.dev/blog/sota-on-swe-bench-verified-with-inference-time-scaling-and-critic-model`, 04 2025.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, and et al. OpenHands: An open platform for AI software developers as generalist agents. In *ICLR*, Singapore, 2025b. OpenReview.net.

You Wang, Michael Pradel, and Zhongxin Liu. Are "solved issues" in SWE-bench really solved correctly? an empirical study. In *ICSE*, Rio de Janeiro, Brazil, 2026.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: advancing LLM reasoning via reinforcement learning on open software evolution. In *NeurIPS*, San Diago, CA, USA / Mexico City, Mexico, 2025. Curran Associates, Inc.

Parker Whitfill, Cheryl Wu, Joel Becker, and Nate Rush. Many SWE-bench-Passing PRs would not be merged into main. `https://metr.org/notes/2026-03-10-many-swe-bench-passing-prs-would-not-be-merged-into-main/`, 03 2026.

Junjielong Xu, Boyin Tan, Xiaoyuan Liu, Chao Peng, Pengfei Gao, and Pinjia He. Scalable supervising software agents with patch reasoner. *CoRR*, 2510.22775, 2025.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. In *NeurIPS*, San Diago, CA, USA / Mexico City, Mexico, 2025. Curran Associates, Inc.

Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. UTBoost: Rigorous evaluation of coding agents on SWE-Bench. In *ACL*, pp. 3762–3774, Vienna, Austria, 2025. Association for Computational Linguistics.

Wenhao Zeng, Yuling Shi, Xiaodong Gu, Chao Hu, Chaofan Wang, Yuhao Cui, Hongting Zhou, Mengnan Qi, Jianqiao Wangni, Zhaojian Yu, Shuzheng Gao, Kai Cai, and Shilin He. Dockerless: Environment-free program verifier for coding agents. *CoRR*, 2606.28436:1–20, 2026.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Xin Zhao, Ruihua Song, Yuan Zhang, Peng Wang, Cheng Chen, Jirong Wen, and Kai Jia. Immersion in the GitHub universe: Scaling coding agents to mastery. *CoRR*, 2602.09892, 2026.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization. *CoRR*, 2507.18071:1–7, 2025.

Terry Yue Zhuo. ICE-Score: Instructing large language models to evaluate code. In *Findings of EACL*, pp. 2232–2242, St. Julian's, Malta, 2024. ACL.

A Verifier Agent Implementation

For each verification instance, SWE-AaaV provides a unified executable environment shared by the verifier agent and the deterministic harness. We materialize the base $\mathcal{R}_{base}$, reference $\mathcal{R}_{ref}$, and candidate $\mathcal{R}_{cand}$ as immutable Git revisions within a single local repository, with both the reference and candidate commits derived from the same base commit. Before executing commit operations, any Git hooks are disabled to avoid project-specific installation or validation procedures that may introduce failures unrelated to verification.

Our verifier scaffold is adapted from the OpenHands v0.52.1 (Wang et al., 2025b). It retains three basic interaction tools: `execute_bash` for arbitrary bash commands, `str_replace_editor` for file inspection and modification, and `think` for explicit reasoning. We additionally expose the execution harness through the `run_test_scripts` tool (Section 4.2), allowing the agent to validate intermediate evidence and refine it from execution feedback. The original `finish` action is routed through the same interface: verification terminates only when the submitted evidence passes the harness checks; otherwise, the resulting observation is returned to the agent for refinement.

To make repository exploration efficient, the commit-based environment supports progressive disclosure of implementation differences. The agent initially receives only file-level change summaries between the commits, analogous to `git diff --name-status`, and retrieves finer-grained files, functions, or diffs only when needed. This is particularly useful for model-generated patches, which may contain auxiliary reproduction or testing artifacts that are difficult to filter reliably with static rules. Lightweight diff summaries therefore provide effective search guidance while reducing unnecessary context consumption and attention distraction.

Moreover, we employ two optimizations to reduce repeated test execution during iterative refinement. The checks follow a fast-fail order, immediately returning once an earlier admissibility condition is violated and skipping unnecessary checkouts and executions. We also cache the latest execution results of the test scripts. Since repository commits are immutable, the cached results can be safely reused unless the script has changed since its previous execution.

B Additional Experimental Details

B.1 Model Backbones

We use the following model backbones, with parameter counts distinguishing total model size from parameters activated per token:

- **Qwen3-Coder-30B-A3B.** An open-weight coding model with 30.5B total parameters and 3.3B activated parameters, used to sample candidate solutions and as an offline verifier backbone.
- **Qwen3-30B-A3B-Instruct-2507.** An open-weight instruction-tuned model with 30.5B total parameters and 3.3B activated parameters, used to initialize the policy before SFT and RL.
- **Qwen3.6-35B-A3B.** An open-weight model with 35B total parameters and 3B activated parameters, used for offline verification and as the verifier backbone during RL.
- **Qwen3.6-Plus.** A closed-source model with 397B total parameters and 17B activated parameters, used as an offline verifier backbone.
- **DeepSeek-V4.1-Flash.** An open-weight model with 552B backbone parameters and an additional 196B Engram parameters, activating 8B parameters per token during prefill and 16B during decoding. We use it as an offline verifier backbone.

B.2 Offline Verifier Evaluation

Benchmark construction. We construct the offline verifier benchmark from coding-agent trajectories sampled on SWE-bench Verified using Qwen3-Coder-30B-A3B with the OpenHands scaffold. For each issue, we sample four trajectories, each subject to a maximum of 100 interaction rounds and a context limit of 128K tokens. For each candidate solution, we run the benchmark's pre-defined unit-test verifier and use its binary outcome as a proxy label.

We treat these labels as approximate rather than as ground truth. As discussed in Section 3, pre-defined test suites may have limited coverage or encode implementation-specific assumptions, potentially producing false positives or false negatives. These proxy labels provide a scalable, common reference for quantitative comparisons of verifiers. For cases where SWE-AaaV disagrees with the pre-defined verifier, two annotators manually assess whether the candidate solution actually resolves the issue. Any disagreements between the annotators are resolved through discussion to reach a consensus.

To prevent issues with more sampled candidates from disproportionately influencing the benchmark, we retain at most one candidate that passes the pre-defined tests and one that fails them for each issue. The resulting benchmark contains 618 candidates, with 309 positive and 309 negative proxy labels.

B.3 Baselines and Ablations

We reproduce rubric-based grading (Raghavendra et al., 2026) and the SWE-Gym verifier (Pan et al., 2025) following their original papers. The prompts used are provided in Appendix D.

- **LLM-as-a-Judge.** This baseline directly assesses whether a candidate patch resolves the reported issue without executing code. The grading model receives the problem statement, candidate patch, and reference patch, and outputs a binary judgment indicating whether the issue is resolved. We evaluate multiple grading models, as reported in Table 1.
- **Rubric-based grading.** Following the original paper, we use OpenHands with Claude-4.5-Sonnet (Anthropic, 2025) to generate task-specific rubrics. The grading models reported in Table 1 then assess candidate patches using these rubrics.
- **Critic model.** We include OpenHands-32B-Critic (Wang, 2025), which predicts a continuous correctness score from the coding-agent trajectory using a regression head.
- **GenRMs.** We use SWE-Gym OpenHands-32B-Verifier, a generative reward model finetuned from Qwen2.5-Coder-32B-Instruct (Hui et al., 2024) for software engineering verification. Unlike the prompting-based grading baselines, this verifier has been explicitly trained to predict solution correctness on SWE tasks.

The ablation variants are as below:

- **SWE-AaaV w/o adaptation.** The agent writes a single test script shared by the reference and candidate implementations, rather than a pair of implementation-adapted scripts. Reference access and harness validation are retained.

- **SWE-AaaV w/o validation.** We remove the `run_test_scripts` tool and disable the validation invoked by `finish`. The agent retains reference access, paired-script generation, and the basic interaction tools, including `execute_bash` for code execution, but receives no enforced harness feedback and may finish without satisfying the admissibility checks.
- **SWE-AaaV w/o reference.** To isolate the role of the reference solution, we evaluate an ablation that removes $\mathcal{R}_{ref}$ from the verifier agent's environment. The verifier can still inspect the repository and candidate solution to construct executable tests, but cannot compare the candidate against a reference implementation or use the reference as a behavioral oracle. This ablation isolates the additional benefit of reference-grounded white-box differential testing over repository-level agentic verification alone.

Implementation. For SWE-AaaV, each verification episode allows at most 100 assistant turns and a context window of up to 128K tokens. The ablations use the same agent configuration except for the component removed in each variant. Unless otherwise specified, all models use their officially recommended inference hyperparameters.

We repeat each experiment four times and report the mean of each evaluation metric: AUC, FNR and FPR. For the reported offline metrics, we treat a candidate patch that passes the pre-defined unit tests as a positive instance, following the proxy-label convention above. Let $y_i \in \{0, 1\}$ denote the label of candidate i, and let s_i denote its verifier score. For a decision threshold τ, the predicted label is $\hat{y}_i(\tau) = \mathbb{1}[s_i \geq \tau]$, where $\mathbb{1}[\cdot]$ is the indicator function. We denote the resulting numbers of true positives, true negatives, false positives, and false negatives by TP, TN, FP, and FN, respectively.

- **Area under the ROC curve (AUC).** AUC measures how well verifier scores distinguish positive from negative proxy labels across decision thresholds. Let $\mathcal{P} = \{i : y_i = 1\}$ and $\mathcal{N} = \{i : y_i = 0\}$. The empirical AUC is

$$\mathrm{AUC} = \frac{1}{|\mathcal{P}|\,|\mathcal{N}|} \sum_{i \in \mathcal{P}} \sum_{j \in \mathcal{N}} \left(\mathbb{1}[s_i > s_j] + \frac{1}{2} \mathbb{1}[s_i = s_j] \right). \tag{11}$$

  Thus, AUC is the probability that a randomly selected proxy-positive candidate receives a higher score than a randomly selected proxy-negative candidate, with ties receiving half credit. AUC does not require selecting a decision threshold.

- **False negative rate (FNR).** FNR measures the fraction of proxy-positive candidates rejected by the verifier:

$$\mathrm{FNR} = \frac{\mathrm{FN}}{\mathrm{TP} + \mathrm{FN}}. \tag{12}$$

- **False positive rate (FPR).** FPR measures the fraction of proxy-negative candidates accepted by the verifier:

$$\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{TN} + \mathrm{FP}}. \tag{13}$$

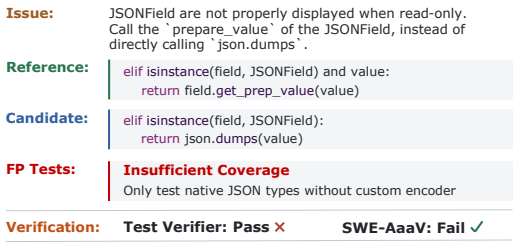
For baselines that output continuous-valued rewards, we select the threshold that maximizes classification accuracy on the full benchmark:

$$\tau^* \in \arg\max_{\tau} \frac{1}{n} \sum_{i=1}^{n} \mathbb{1}[\hat{y}_i(\tau) = y_i], \tag{14}$$

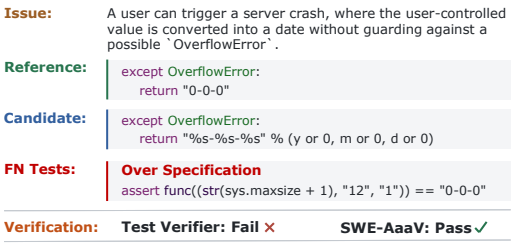
where n is the number of evaluated candidates. We then use the selected threshold τ^* to compute FNR and FPR. These metrics characterize a benchmark-selected operating point and may be optimistic relative to performance under an independently calibrated decision threshold. Verifiers that output binary judgments require no threshold selection.

B.4 RL Training Details

Training and evaluation data. We evaluate on SWE-bench Verified and construct the RL training set from SWE-rebench (Badertdinov et al., 2025). We apply the standard fail-to-pass filtering procedure (Jimenez et al., 2024) and exclude all instances from repositories represented in SWE-bench Verified to ensure repository-level separation between training and evaluation. The resulting RL training set contains 6,211 instances.



(a) An FP example: `django__django-12308`.



(b) An FN example: `django__django-16667`.

Figure 5: Two misclassification examples of the pre-defined test verifier from SWE-bench Verified.

Policy initialization and SFT. In preliminary experiments, neither Qwen3-30B-A3B-Instruct-2507 nor Qwen3-Coder-30B-A3B showed a clear trend of performance improvement under RL with standard unit-test rewards. Motivated by this observation, we perform supervised finetuning (SFT) before RL. We construct the SFT corpus from successful trajectories in CoderForge-Preview (Ariyak et al., 2026), generated by Qwen3-Coder-480B-A35B on SWE-smith (Yang et al., 2025) and R2E-Gym (Jain et al., 2025). To maintain repository-level separation, we exclude trajectories from all repositories used in subsequent RL training or final evaluation. This yields 27,634 successful trajectories, on which we finetune Qwen3-30B-A3B-Instruct-2507 for three epochs. During SFT, we apply step-level error masking to turns with invalid tool-call formatting: these turns are excluded from the supervised loss, while the remaining valid steps in each trajectory are retained (Tao et al., 2026). The resulting checkpoint serves as the common initialization for all subsequent RL experiments.

We use Qwen3.6-35B-A3B as the verifier backbone during RL. Its role is to evaluate candidate patches produced by policy rollouts and provide correctness rewards, rather than to serve as a teacher for direct policy distillation. The verifier backbone and the policy model are therefore decoupled. The verifier need not be more capable than the policy model in general. The capability requirement is to assess candidate patches using the privileged information available during evaluation.

Reward formulation. Our RL comparison isolates the effect of the correctness verifier, with all other training settings held fixed. The baseline obtains the correctness reward r_c from the pre-defined test verifier, whereas SWE-AaaV uses the adaptive executable verifier introduced in Section 4.

Following SWE-RM (Shum et al., 2026), we define the trajectory-level reward as Equation 10, where $\lambda = 0.1$ provides a small reward for episode completion, and $r_c \in \{0, 1\}$ denotes the correctness outcome returned by the corresponding verifier.

Policy optimization. We optimize the policy using GSPO, with neither a KL-based reward penalty nor a KL regularization term in the optimization loss. Each training batch contains 8 groups of 8 sampled trajectories, for a total of 64 trajectories. Each rollout is limited to 100 assistant turns and a maximum context length of 128K tokens with the OpenHands scaffold. These settings are identical across the runs with the test verifier and SWE-AaaV.

System implementation. Our RL implementation builds on the ROLL framework (Wang et al., 2025a), using vLLM (Kwon et al., 2023) for rollout inference and Megatron (Shoeybi et al., 2019) for policy training. All GPU workloads run on NVIDIA H800 GPUs, with separate pools of 32 GPUs each allocated to rollout inference, policy training, and verifier backbone inference.

For both standard unit-test rewards and SWE-AaaV, completed trajectories are submitted to the asynchronous reward computation pipeline described in Section 5, enabling rollout generation and verifier execution to proceed concurrently. The windowed FIFO buffer has a capacity equivalent to two training batches.

C Case Studies

Figure 5 illustrates two unit-test misclassifications identified in the disagreement analysis (Section 7.1). The cases show how candidate-conditioned testing can uncover missing coverage and avoid implementation-specific assertions.

False positive from insufficient coverage. In Figure 5a, the provided tests exercise only standard JSON-serializable values and fail to distinguish the candidate's direct use of `json.dumps` from the required field-specific serialization behavior. By inspecting the difference between the reference and candidate implementations, SWE-AaaV identifies the missing support for custom encoders and constructs a targeted test that exposes the defect. The candidate passes the pre-defined suite but fails this additional behavioral check, illustrating how white-box differential testing can reveal incomplete fixes beyond existing coverage.

False negative from implementation-specific assertions. In Figure 5b, the issue requires preventing an `OverflowError`-induced crash, whereas the pre-defined test additionally requires the exact sentinel value returned by the reference solution. The candidate handles the exception but returns a different non-crashing value, causing the test to fail. SWE-AaaV checks the problem-scoped behavior and correctly accepts this alternative implementation. This case illustrates the importance of using the reference as a behavioral oracle within the issue scope, without treating every implementation detail as a requirement.

D PROMPTS OF SWE-AAAV AND BASELINES

The prompts of rubric-based grading (Raghavendra et al., 2026), the critic model (Wang, 2025), and the GenRM (Pan et al., 2025) are based on their original papers. Below are the prompts of SWE-AaaV and LLM-as-a-Judge.

SWE-AaaV

```
system_template: |-
You are a software-engineering test author for an automated evaluation system.
Your task is to create reliable behavioral test scripts that allow the evaluation
     system to decide whether the Candidate implementation resolves the given issue.

<CONTEXT>
You are given a Python repository with three commits:
- Base: the original state before the fix;
- Reference: a correct implementation built on top of Base;
- Candidate: the implementation to evaluate, also built on top of Base.
</CONTEXT>

<CORE_PRINCIPLES>
- The issue description defines the scope;
- Reference is the behavioral oracle for in-scope behavior, not a code template;
- Candidate should be evaluated by valid behavioral tests, not implementation
     similarity;
- Your tests must be delta-driven: focus on in-scope behavioral differences from
     Reference -> Candidate.
</CORE_PRINCIPLES>

<CONSTRAINTS>
- You MUST write exactly these two scripts: `/workspace/test_reference.py` and `/
     workspace/test_candidate.py`;
- Use the `run_test_scripts` tool to validate the scripts using the evaluation system;
- If the evaluation system reports a problem, revise the scripts and try again.
</CONSTRAINTS>
""".strip()

instance_template: |-
Consider the following issue description:

<issue_description>
{{ problem_statement }}
</issue_description>

The Python repository has three predefined commits and is initially checked out at
     Reference:

<repo_path>
{{ repo_path }}
</repo_path>

<commits>
```

```
- Base: `{{ base_commit }}`, the original state before the fix;
- Reference: `{{ reference_commit }}`, a correct implementation built on top of Base;
- Candidate: `{{ candidate_commit }}`, the implementation to evaluate, also built on
    top of Base.
</commits>

Changed files are shown by `git diff --name-status ...`:

<changed_files>
# Base -> Reference
{{ reference_file_changes }}

# Base -> Candidate
{{ candidate_file_changes }}

# Reference -> Candidate
{{ delta_file_changes }}
</changed_files>

Your task is to create delta-driven behavioral test scripts that allow the automated
    evaluation system to determine whether Candidate resolves the issue.

Follow exactly these phases:

* Phase 1. Define issue scope
Read the issue description and inspect Reference behavior to define the issue scope.

In scope:
- explicit requirements in the issue description;
- reasonable behavioral implications clearly supported by both the issue description
    and Reference;
- regressions of existing external behavior that should remain unchanged.

Out of scope unless clearly required by the issue:
- extra Reference behavior unrelated to the issue;
- internal implementation styles that do not affect external behavior, such as helper
    APIs and refactoring.

* Phase 2. Delta-driven test planning
Start from the Reference -> Candidate code difference, disregarding changes in docs and
     test scripts.
Inspect code as needed and identify possible behavioral differences between Reference
    and Candidate.

* Phase 3. Delta-derived in-scope hypotheses
Filter the possible behavioral differences based on the issue scope from Phase 1.
Create a small set of in-scope hypotheses.

Each hypothesis should be:
- motivated by the Reference -> Candidate delta, or by the original issue when no
    meaningful delta risk exists;
- able to reveal behavioral divergence if Candidate does not match Reference;
- externally observable and testable;

Examples:
- H1: Candidate may fail for an implicit edge case;
- H2: Candidate may break an existing behavior that should remain unchanged.

Check again and discard hypotheses that only concern out-of-scope implementation
    details.

* Phase 4. Create `/workspace/test_reference.py`
The script must:
- write at least one test for each in-scope hypothesis;
- be executable directly with Python and not require any testing frameworks like pytest
     or unittest;
- catch exceptions appropriately so that all tests can run;
- avoid background servers or long-running processes;
- print enough information to understand what was tested and why it failed;
- print a final machine-readable result line in JSON format that maps stable test names
     to booleans.

Example format for a test script:
```python
import json
from some_package import relevant_module

def test1():
```
```

```
    print("# Test H1 with empty input")
    try:
        test_input = "input for H1"
        # the behavior to test
        result = relevant_module.in_scope_function(test_input)
        expectation = "expected output"

        # check if the behavior is expected
        if result == expectation:
            print("PASS")
            return True
        else:
            print("FAIL")
            print(f"Expected: {expectation!r}")
            print(f"Got: {result!r}")
            return False

    except Exception as e:
        print("ERROR")
        print(f"{type(e).__name__}: {e}")
        return False

if __name__ == "__main__":
    # print one final machine-readable result line in JSON format
    test_res = {}
    test_res["H1_empty_input"] = test1()
    print(json.dumps(test_res))
```

```
* Phase 5. Create `/workspace/test_candidate.py`
Create this script by copying `/workspace/test_reference.py`.
If Reference and Candidate use different names or structures for out-of-scope
     implementation details only, modify the script with minimal replacement.
The script must test exactly the same behavior and use exactly the same result
     dictionary keys.

* Phase 6. Test validation and repair loop
Use the `run_test_scripts` tool to validate the scripts using the evaluation system.

Revise the scripts when reporting these issues:
- Invalid test scripts: MUST print a final machine-readable result line in JSON format;
- Failure to satisfy test oracle: Reference is the behavioral oracle for in-scope
     behavior, where all tests MUST pass;
- Inconsistent test scripts: The test results of these two scripts MUST be consistent
     on Base;
- No resolved evidence: MUST include at least one test that fails on Base to reproduce
     the issue.

Review the hypotheses alongside the test results, and also revise the scripts when
     receiving unexpected results.
```

LLM-as-a-Judge

```
system_template: |-
You are a software-engineering expert. Your task is to judge whether the Candidate
     implementation resolves the issue.

<CONTEXT>
You are given:
- an issue description;
- Reference: a correct implementation;
- Candidate: the implementation to evaluate.
</CONTEXT>

<CORE_PRINCIPLE>
- The issue description defines the evaluation scope.
- Reference is the behavioral oracle for in-scope behavior, not a code template;
- Candidate should be judged by valid behavioral evidence, not implementation
     similarity.
</CORE_PRINCIPLE>

<JUDGE_SCOPE>
In scope:
- explicit requirements in the issue description;
```

```
- reasonable behavioral implications clearly supported by both the issue description
    and Reference;
- regressions of existing external behavior that should remain unchanged.

Out of scope unless the issue clearly requires them:
- extra Reference behavior unrelated to the issue;
- internal implementation styles that do not affect external behavior, such as helper
    APIs and refactoring.
</JUDGE_SCOPE>

<VERDICT_CRITERIA>
- `resolved = true` if Candidate resolves the issue: it MUST match Reference on in-
    scope behavior and avoid relevant regressions.
- Otherwise, `resolved = false`.
</VERDICT_CRITERIA>

<JSON_SCHEMA>
You must respond with a JSON object containing two fields:
{
    "justification": (string) Concise explanation of the verdict.
    "resolved": (boolean) The final boolean verdict.
}
</JSON_SCHEMA>

instance_template: |-
Consider the following issue description:

<issue_description>
{{ problem_statement }}
</issue_description>

The Reference implementation:

<reference>
{{ gold_patch }}
</reference>

The Candidate implementation to evaluate:

<candidate>
{{ model_patch }}
</candidate>

Your task is to judge whether the Candidate implementation resolves the issue.

Let's think step by step, and output the JSON verdict directly.
```