FIX–TEST COLLAPSE: DIAGNOSING AND RESOLVING SELF-CONFIRMING VERIFICATION IN CODE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Self-testing lets code agents inspect execution feedback and repair errors, yet successful local validation can precede failure on held-out tests. We call this failure **Fix–Test Collapse** and distinguish tests that miss bugs from misinterpretation of outputs that expose them. We propose **Falsification-Guided Self-Testing**, which combines protocol SFT, action-level reinforcement learning, and trajectory rewards under a single-trajectory interface that withholds reference outputs and hidden verdicts from the policy. Offline scorers reward diagnostic tests on contest tasks, and an issue-grounded judge, updated with on-policy hard negatives, supplies the diagnostic signal on repository tasks. Across three model families, pass rates improve by up to 34.7% on CodeContests and 19.5% on LiveCodeBench relative to untrained tool-enabled policies, and Qwen3.5-35B reaches 48.3% resolution on 731 SWE-bench Pro tasks and 28.3% on the complete 113-task DeepSWE split. In a matched control starting from an identical SFT checkpoint, falsification RL outperforms final-only RL on all three paired training seeds, by 7.7 points on Pro and 10.6 points on DeepSWE on average. Training also raises self-test coverage and lowers collapse among locally validated submissions; this conditional reliability measure is distinct from an all-instance failure rate.

1 INTRODUCTION

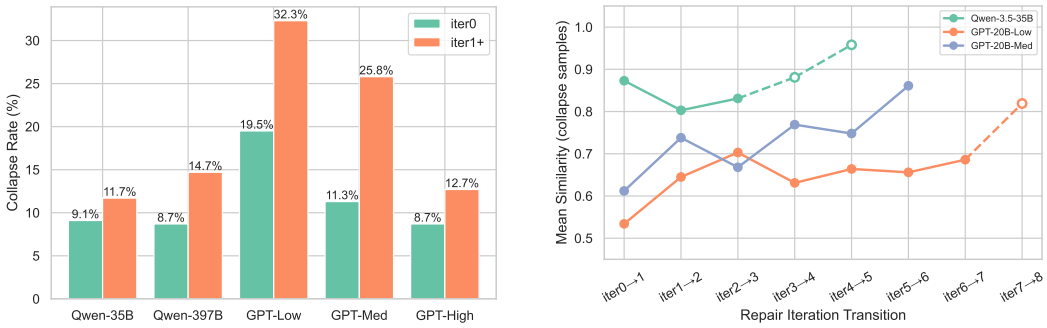
Code agents increasingly alternate between writing code, executing tests, interpreting tool feedback, and deciding whether to repair or submit (Schick et al., 2023; Yang et al., 2024; Pan et al., 2024; Yang et al., 2025). Execution grounds this process in observed behavior, but does not guarantee reliable verification: the same model can generate both an incorrect implementation and tests that confirm its mistaken interpretation of the task. Self-correction is particularly difficult when the model must also produce and interpret its own feedback (Madaan et al., 2023; Shinn et al., 2024; Huang et al., 2023).

We study **Fix–Test Collapse**: an agent treats local verification as successful, submits its candidate, and fails the held-out evaluator. This observable event can arise because the tests miss the relevant error or because the model misjudges execution outputs that already expose it. The shared assumptions of the coder, tester, and output interpreter are a possible source of these failures; the collapse event alone does not identify their cause. Separating input quality from output interpretation makes the verification bottleneck actionable.

Our setting has two constraints. First, the agent acts within a **single trajectory**, rather than selecting among independently sampled solutions. Second, its interaction is **reference-output-free**: it receives its own execution outputs and logs, but no reference outputs or hidden correctness verdicts. This restriction applies to policy-visible feedback, not offline training supervision: benchmark tests, reference programs, and final patch evaluators may assign rewards after execution. The distinction is essential in repository repair, where the agent must construct an issue-grounded verification signal from the current checkout.

Falsification-Guided Self-Testing trains agents to discover evidence against their current candidate and repair from that evidence. Supervised fine-tuning first teaches the self-testing protocol; action-level reinforcement learning then shapes testing, decisions, and repair, followed by end-to-end trajectory training. Contest tasks use reference executions to score test discriminativeness offline. Repository tasks instead use a learned rubric judge, updated with on-policy hard negatives to address

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.



(a) Fix–Test Collapse by iteration stage.

(b) Similarity between consecutive repair patches.

Figure 1: Failure modes of self-testing code agents during repair. In (b), similarity is normalized token-level Levenshtein similarity between consecutive repair patches; hollow markers and dashed segments denote $n < 5$ collapse samples; solid markers denote $n \geq 5$. Qwen3.5-397B and GPT-OSS-20B-High are omitted from (b) because they retain only four and two collapse samples at the first transition.

Table 1: Accuracy and collapse on TACO-Verified under a manually triggered self-testing loop (every instance enters the loop). Oracle inputs are benchmark failure inputs supplied without reference outputs. *Collapse incidence*: fraction of all instances where self-generated tests pass but benchmark tests fail. *Recovered by Oracle*: fraction of collapsed instances repaired with oracle inputs. †Output truncation artifact; not a controlled comparison of reasoning effort.

	Qwen3.5		GPT-OSS		
Parameter Size	35B	397B	20B	20B	20B
Reasoning Effort	–	–	Low	Med	High
Pass@1 (no repair)	71.6	85.4	63.9	79.2	57.8†
After Self-generated Tests	73.9	88.5	71.8	83.7	59.3
After Oracle Benchmark Inputs	78.5	94.4	83.3	93.9	64.4
Collapse incidence	5.9	7.9	22.3	10.9	3.0
Recovered by Oracle	59.6	73.0	44.9	70.1	66.7

errors exposed by the evolving policy, and a collapse penalty that discourages false local validation of unsuccessful patches.

We evaluate contest-style code generation across three model families and center repository evaluation on SWE-bench Pro and DeepSWE. SWE-bench Verified results and paired uncertainty summaries appear in Appendix A; the earlier Pro language breakdown and the reward ablations appear in Appendices B and H. Our contributions are: (i) a decomposition of false local validation into test-generation and output-judgment failures; (ii) a falsification-guided curriculum with an explicit policy/scorer information boundary and a dynamic repository judge; and (iii) an evaluation of correctness, verification behavior, and the limits of repository generalization, including a matched post-SFT control with three paired training seeds.

2 Fix–Test Collapse

Let $L(\tau) = 1$ mean that the agent submits after treating valid local verification on the final candidate as successful, and let C_K denote held-out correctness. We define the collapse event as

$$E(\tau) = \mathbf{1}[L(\tau) = 1 \wedge C_K = 0]. \tag{1}$$

Under an input-only interface, "pass" denotes the agent's judgment of the observed output, not a reference verdict returned by the executor. A trajectory without valid local verification is not counted as collapsed; it is an unverified submission. We distinguish the all-instance incidence $P(E = 1)$ from

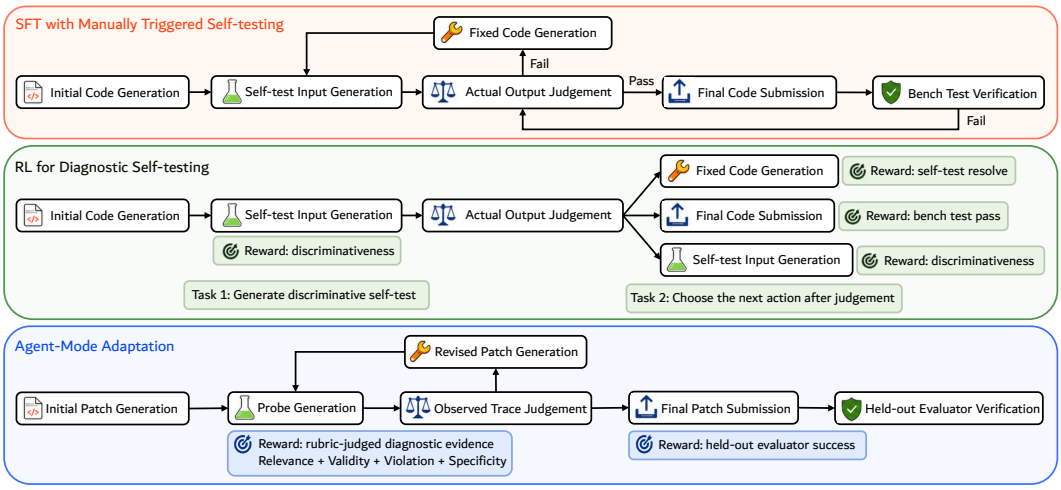


Figure 2: Overview of Falsification-Guided Self-Testing.

the conditional error rate $P(C_K = 0 \mid L = 1)$ and local-validation coverage $P(L = 1)$. Self-test invocation is reported separately and need not equal validation coverage.

Controlled diagnosis. On TACO-Verified (Li et al., 2023), every instance enters a manually triggered self-testing loop. Table 1 compares initial solutions, self-test-driven repair, and repair with benchmark failure inputs supplied without reference outputs. Oracle inputs are supplied to failed tasks, including both collapsed and unverified failures; the recovery statistic separately conditions on collapse. These inputs recover a substantial fraction of collapsed cases, indicating that verification is an important bottleneck without excluding limitations in repair. The observation includes Qwen3.5 (Team, 2026) and GPT-OSS-20B (Agarwal et al., 2025); the high-effort GPT-OSS condition suffers from output truncation, so it does not establish a clean effect of reasoning depth.

Failure decomposition. Within collapse cases analyzable using reference executions, we distinguish inputs that never expose a candidate error from outputs that expose an error but are accepted by the policy. Table 4 reports the aggregate decomposition. Attribution is performed on tests of the final submitted candidate: a collapse is a judgment error if at least one valid test exposes a discrepancy that the policy accepts; otherwise it is a weak-input case. Tests on superseded candidates do not determine final-candidate attribution. Cases without enough reference or action information to apply this rule are excluded from the two-way decomposition, but remain in collapse incidence. Figure 1 shows the stage-conditioned collapse and patch-similarity observations.

3 Falsification-Guided Self-Testing

3.1 Information Boundary and Trajectory Objective

Given a specification p, the policy generates f_0, proposes inputs, observes execution traces, and chooses to probe, repair, or submit. A trace contains the input, runtime status, stdout, and stderr of the current candidate. Reference traces and scorer labels are never inserted into the policy context.

For contest tasks, the implemented offline scorer defines test discriminativeness by

$$\delta(f, x) = \mathbf{1}[v_p(x) = 1 \wedge s(f^*(x)) = \mathrm{OK} \wedge \phi(t(f, x)) \neq \phi(t(f^*, x))], \tag{2}$$

where $v_p(x)$ indicates independently validated task-input constraints, f^* is an offline reference program, s is execution status, and ϕ returns whitespace-normalized stdout for successful execution and a status token otherwise. A diagnostic test input (DTI) satisfies $\delta = 1$. Invalid reference executions are non-diagnostic. Novelty means that the normalized input has not previously appeared in the trajectory. Input validity and task-specific output-checking details are recorded in Appendix C.

This scorer measures *discriminativeness*, not whether the policy can recognize the error. *Judgeability* is the policy-side ability to interpret the observed trace and choose a useful action. We train that ability through decision and repair tasks rather than treating output disagreement as a direct measure of recognizability. Let $D(\tau) = 1$ if any executed self-test exposes a DTI, and let C_0, C_K be initial and final benchmark correctness. The contest reward is

$$r(\tau) = (2C_K - 1) + \alpha C_0 + \lambda(1 - C_0)C_K D(\tau), \tag{3}$$

with $\alpha = 0.6$ and $\lambda = 0.2$. Initially correct successful trajectories receive more reward than initially incorrect trajectories that discover a DTI and recover. The diagnostic bonus is gated by final success; a failed submission receives no such bonus even when a diagnostic test was found. Co-occurrence of a DTI and successful repair does not alone prove that the evidence caused the repair.

3.2 CURRICULUM

The trajectory objective couples several behaviors: generating a candidate, finding an input that exposes an error, interpreting the resulting trace, and producing a successful repair. A policy that has not learned the interaction protocol may rarely sample complete diagnostic-recovery trajectories. We therefore organize training into protocol SFT, action-level RL, and end-to-end trajectory RL (Figure 2). The curriculum progressively shifts from demonstrated interactions to autonomous decisions while preserving the same policy-visible execution interface.

Stage 1: SFT with manually triggered self-testing. We construct demonstrations offline from training problems by generating an initial program, requesting input-only self-tests, executing the current candidate, and continuing with testing, repair, or submission. The data mix three trajectory types. *Clean-pass* trajectories submit an initially correct program after local verification and pass benchmark evaluation. *Self-test-repair* trajectories expose an error through self-generated inputs, revise the program using the observed trace, and eventually submit a benchmark-correct solution. *Collapse-recovery* trajectories start from a locally validated but incorrect candidate; benchmark failure inputs selected offline provide an opportunity to continue repair.

In collapse-recovery demonstrations, the policy receives the selected inputs and the candidate's actual outputs, not the reference outputs or benchmark verdict. Reference executions and final benchmark outcomes are used offline to construct and validate the demonstrations. The three trajectory types teach complementary behavior: submit when verification is sufficient, repair when execution exposes an error, and continue investigating when the original self-tests are inadequate. The model learns the interaction sequence and action formats from these examples.

Stage 2, Phase A: unified action-level RL. Starting from the SFT policy, we construct local training tasks by exposing a prefix of a Stage 1 interaction and sampling the next action, training the same policy on three skills. *Tester* tasks generate inputs conditioned on the problem, current program, and available history; the offline scorer in Eq. (2) executes them and labels candidate–reference disagreements. Discovering $n > 0$ DTIs earns $\min(1 + 0.1(n - 1), 1.5)$, an incorrect candidate with no discovered DTI receives zero, and a correct candidate with a valid self-test receives $+1$. *Repair* tasks condition the revision on the problem, current code, and actual execution evidence: a fix that eliminates every previously exposed diagnostic failure in the supplied prefix receives $+1$, and an unsuccessful fix -0.5.

Decision tasks train the choice among submitting, fixing, and requesting further tests: submission is scored against benchmark tests offline, repair uses the failure-removal signal, and a further-test request receives $+1$ for finding a novel DTI and -0.5 if it is duplicate or non-diagnostic, with novelty checked against the history of normalized inputs. Reference traces and scoring labels remain outside the policy context; Appendix C gives the reward table and operational definitions. Together, these signals target complementary parts of the trajectory objective (diagnostic discovery, correction once evidence is available, and the choice to investigate, revise, or stop), supplying local feedback before the policy must coordinate all three skills within an autonomous rollout.

Stage 2, Phase B: end-to-end multi-turn RL. We then initialize full rollouts from raw problem descriptions. The policy generates its own initial program and controls subsequent testing, repair, and submission without externally supplied failure inputs. Each tool call returns the candidate's

runtime status and actual outputs, withholding reference outputs and hidden correctness verdicts. GRPO optimizes Eq. (3) over the resulting trajectories: final correctness supplies the primary signal, αC_0 favors a correct initial solution, and the diagnostic bonus rewards initially incorrect trajectories that discover a DTI and finish correctly. Thus the local skills learned in Phase A are optimized jointly under the consequences of the policy's own earlier actions. The fixed-budget initialization comparison in Appendix D evaluates the role of SFT in making these training trajectories accessible.

3.3 Repository adaptation with a dynamic judge

The agent interacts with a repository through file inspection, edits, shell commands, and issue-directed probes. For a probe x, execution output o, and log context ℓ, the judge estimates whether the trace exposes a genuine violation of the issue. Four hard gates require issue relevance, valid execution, a recognizable violation, and issue-specific failure. Two soft filters score the strength of the behavioral check and whether a correct issue fix would plausibly satisfy it. The verdict is

$$V_{p,j}(x, o, \ell) = \mathbf{1}\left[\bigwedge_{g\in\{\text{rel,exec,recog,spec}\}} G_g = 1 \wedge S_{\text{oracle}} \geq 0.6 \wedge S_{\text{fix}} \geq 0.6\right]. \tag{4}$$

Here j identifies the judge version used for reward assignment. Oracle strength requires a concrete check of issue-relevant behavior; fix plausibility rejects invented requirements and overly specific implementation assumptions. Malformed outputs, skipped tests, collection errors, and unrelated infrastructure failures are non-diagnostic.

Judge training and updating. The judge is Qwen3.5-4B initialized through distillation from Claude Opus 4.6. The initial labeled set contains 4,000 primary hard-gate examples, split 3,600/400, with complete six-dimension labels on the same probe pool. Each dimension is queried separately at temperature zero and aggregated deterministically. During repository RL, high-reward on-policy probes are reviewed for rubric satisfaction without genuine issue-grounded evidence, and labeled hard negatives are incorporated into judge updates. The dynamic judge is part of training; the final policy does not invoke it at evaluation time. Its inputs exclude gold patches, reference outputs, and hidden resolution labels. The earlier schedule updates after policy steps 60, 120, and 170; the extended schedule updates after every 50 steps through step 350 of a 400-step continuation. Each update uses 400 newly labeled hard negatives and 400 replayed valid positives. Appendix E specifies version assignment and audit sampling: in blinded audits at the eight 50-step calibration boundaries, 80.6% of judge-accepted probes sampled before each update were genuinely diagnostic, rising to 94.3% immediately after the updates, a mitigation of reward hacking rather than immunity to it.

Let $D_{\text{agent}}(\tau) = 1$ when any probe receives a positive verdict from the judge version used for that rollout. We use

$$r_{\text{agent}}(\tau) = (2C_K - 1) + \alpha C_0 + \lambda(1 - C_0)C_K D_{\text{agent}}(\tau) - \beta E(\tau), \tag{5}$$

where $\beta = 0.3$. Repository C_0 is defined by a verification-independent draft boundary. Each training rollout begins with a read-only inspection phase followed by one explicit initial-patch submission under a fixed generation budget; the runtime freezes that patch before enabling execution and repair tools, and a separate offline testbed evaluates the frozen snapshot after rollout completion, with the verdict never returned to the policy. If no patch is produced, $C_0 = C_K = 0$. Every comparison arm uses the same boundary. This definition removes a timing incentive that an operational alternative would create: if C_0 were instead snapshotted at the first issue-directed verification command, two successful trajectories making the same initially incorrect patch and eventual correction could receive rewards 1.2 ($C_0 = 0$, $D_{\text{agent}} = 1$) or 1.6 ($C_0 = 1$) depending only on when the first test is run, a 0.4 difference that need not reflect a better initial draft. Where such a boundary cannot be enforced, removing the repository αC_0 term is more interpretable than treating a policy-chosen verification boundary as a fixed initial-quality measure. The diagnostic term rewards evidence discovery in successful repair trajectories; the collapse term penalizes unsuccessful patches submitted with false local validation. Because repository diagnostic labels are approximate, this additional shaping addresses a different signal quality than the exact execution-disagreement scorer.

4 EXPERIMENTS

4.1 SETUP

Training. Contest training uses TACO and CodeContests (Wang et al., 2025b) training problems. Repository adaptation uses R2E-Gym-V1 (Jain et al., 2025). The extended configuration reserves 4,000 training tasks and 200 validation tasks from 11 repositories and runs 400 additional policy updates, retaining dynamic judge updates. Both learned continuation arms start from the same model-specific earlier repository checkpoint at step 200; final-success-only RL therefore controls the effect of rewards during continuation, conditional on the shared earlier training. Checkpoints are selected by validation resolution, with ties broken by fewer tool calls and then an earlier step. Exact task-ID and normalized-issue matches to evaluation tasks are excluded; shared repositories are allowed and do not establish task independence.

Evaluation. We evaluate CodeContests and LiveCodeBench (Jain et al., 2024), and report SWE-bench Pro (Deng et al., 2025) and DeepSWE (Huang et al., 2026) in the main text. Pro uses a 731-task public-subset protocol with the language counts in Appendix B. DeepSWE uses the complete official test split of 113 tasks; every condition uses this same inventory, rates weight tasks equally, and no language-stratified subsampling is applied. The benchmark provides Harbor-format task environments and program-based verifiers. SWE-bench Verified (Jimenez et al., 2023) is supplementary (Appendices A, H, I, and J). Repository runs use OpenHands (Wang et al., 2024); a rollout is capped at 200 assistant interaction turns, 200 shell executions, 65,536 generated tokens, and two hours.

Baselines and metrics. Official denotes the untrained tool-enabled policy; Trigger forces a self-test. CodeT (Chen et al., 2022) samples ten candidates and ten test-generation calls (the 20 calls in Table 2), and ReVeal (Jin et al., 2025) is a trained self-verification baseline. We add inference-time adaptations of Self-Edit (Zhang et al., 2023) and Revisit Self-Debugging (Chen et al., 2025); these do not reproduce a separately trained Self-Edit editor. Repository Adv. Prompt instructs the agent to falsify its patch. Final-success-only RL is the matched continuation control in Table 2, and the SFT + final-only RL baseline in Table 3 isolates the post-SFT contribution of the repository falsification-RL package under matched policy and environment budgets. Final success, turns, and verification metrics have distinct denominators. Contest uncertainty is the sample standard deviation over three runs, not a confidence interval; paired contest intervals cannot be recovered from these marginal summaries. The Pro language intervals in Table 8 use 10,000 task-level bootstrap replicates with seed 0.

4.2 BENCHMARK RESULTS

Competitive programming. Table 2 shows gains over Official of 19.0, 11.8, and 5.1 percentage points on CodeContests and 14.2, 11.5, and 5.0 points on LiveCodeBench. Qwen3.5-35B reaches 73.8/87.2 with 3.28/4.13 turns, compared with CodeT's 75.4/88.5 using 20 assigned calls; this is a call-count comparison, not a token- or wall-clock-normalized cost claim. Ours also exceeds the two inference-time adaptations for every model, surpassing the stronger Revisit Self-Debugging by 2.2 to 3.0 points on CodeContests and 2.0 to 4.6 points on LiveCodeBench while using fewer turns; protocols and run-level deviations are in Appendix D.1. For Gemma, Trigger reduces CodeContests pass rate from 67.9% to 63.0% (-4.9 points), while LiveCodeBench increases from 83.8% to 84.9% (+1.1 points). Thus compulsory self-testing has no uniform benefit across these settings. These aggregate differences do not identify whether the CodeContests decline comes from uninformative tests, output misinterpretation, or harmful revisions; that attribution requires a paired trajectory audit.

Repository-level comparison. The extended comparison in Table 2 raises Pro resolution from 37.2 to 48.3 for Qwen, 29.3 to 38.3 for GPT-OSS, and 31.3 to 41.0 for Gemma: gains of 11.1, 9.0, and 9.7 percentage points, or 6.3, 6.0, and 6.7 points over the matched final-success-only continuation. On the complete DeepSWE test split, resolutions rise from 10/113 to 32/113 for Qwen, 6/113 to 24/113 for GPT-OSS, and 8/113 to 27/113 for Gemma (28.3%, 21.2%, and 23.9%). The extended Qwen gain corresponds to a task-level paired difference of +11.1 points with 95% interval [7.9, 14.2] (language breakdown in Table 8); these intervals are single-training-seed, and the GPT-OSS and Gemma gains are point estimates whose paired uncertainty cannot be determined from marginal success totals. DeepSWE includes every task in the official split, but complete coverage does not

Table 2: Joint benchmark comparison (%). Contest entries are means over three runs; repository columns report resolved rates under the extended protocol on 731 Pro tasks and the complete 113-task DeepSWE split. Ours uses contest-trained checkpoints for contest columns and repository-adapted checkpoints for repository columns. Bold: best per model and benchmark (ties included); dashes: no result. Earlier repository results are in Appendix B.

	CodeContests		LiveCodeBench		SWE-bench Pro	DeepSWE
Method	Pass ↑	Turns	Pass ↑	Turns	Resolved ↑	Resolved ↑
Qwen3.5-35B						
Official	54.8	1.12	73.0	1.42	37.2	8.8
Trigger	58.9	2.97	76.8	3.04	–	–
CodeT	**75.4**	20.00	**88.5**	20.00	–	–
ReVeal	68.0	4.76	83.8	5.55	–	–
Self-Edit	66.4	3.27	82.8	4.22	–	–
Revisit Self-Debugging	71.6	4.97	85.2	5.69	–	–
Final-success-only RL	–	–	–	–	42.0	19.5
Ours	73.8	3.28	87.2	4.13	**48.3**	**28.3**
GPT-OSS-20B						
Official	51.2	1.76	69.1	1.85	29.3	5.3
Trigger	57.2	3.60	78.6	3.50	–	–
CodeT	61.5	20.00	**80.6**	20.00	–	–
ReVeal	58.4	4.32	75.4	4.80	–	–
Self-Edit	55.6	4.12	74.6	3.72	–	–
Revisit Self-Debugging	60.0	5.03	76.0	6.34	–	–
Final-success-only RL	–	–	–	–	32.3	13.3
Ours	**63.0**	3.77	**80.6**	3.21	**38.3**	**21.2**
Gemma4-26B						
Official	67.9	4.99	83.8	4.41	31.3	7.1
Trigger	63.0	4.55	84.9	3.88	–	–
CodeT	70.9	20.00	87.2	20.00	–	–
ReVeal	68.2	5.12	84.8	4.69	–	–
Self-Edit	65.2	4.77	85.2	3.56	–	–
Revisit Self-Debugging	70.4	6.23	86.0	6.76	–	–
Final-success-only RL	–	–	–	–	34.3	15.9
Ours	**73.0**	4.34	**88.8**	4.00	**41.0**	**23.9**

Table 3: Post-SFT Qwen3.5-35B comparison. Resolved rates (%) are averaged across three paired training seeds (42, 43, 44), each evaluated at evaluation seed 0 on the same 731 Pro tasks and complete 113-task DeepSWE split. The increment is computed before rounding. These checkpoints are separate from the continuation checkpoints in Table 2.

Method	Pro resolved rate	DeepSWE resolved rate
SFT + final-only RL	39.7	15.9
SFT + falsification RL	47.4	26.5
Falsification increment (pp)	+7.7	+10.6
Per-seed increment (pp; seeds 42/43/44)	+7.3/+8.2/+7.7	+9.7/+10.6/+11.5

establish robustness across training seeds. The earlier Qwen repository comparison is summarized in Appendix B.

Post-SFT falsification control. The continuation control above conditions on an already adapted policy. Table 3 therefore compares two independent Qwen repository training arms from the identical protocol-SFT checkpoint, before any other RL: both receive 600 full-trajectory repository updates under the same data schedule, optimizer, policy budget, and execution limits, and differ only in reward, $2C_K - 1$ versus Eq. (5). Training seeds 42, 43, and 44 are paired across arms, with evaluation seed 0 for each trained policy. The increment is positive for every seed, ranging from 7.3 to 8.2 points on Pro and from 9.7 to 11.5 on DeepSWE, but three training seeds provide only a limited

Table 4: CodeContests verification behavior under the autonomous tool-enabled interface (%; 165 tasks × 20 rollouts, $N = 3{,}300$ per condition, seeds 0–19). *Self-test*: invocation coverage. *Collapse*: error rate among locally validated submissions (E/L). Weak input and judgment error are shares of reference-analyzable collapse cases. Stage 2 includes Stage 1 SFT followed by RL.

Model	Training	Self-test ↑	Collapse ↓	Weak input	Judgment err.
Qwen3.5-35B	Official	8.5	78.6	27.3	72.7
	+ Stage 1	96.4	60.4	24.6	75.4
	+ Stage 2	98.2	30.5	45.4	54.6
GPT-OSS-20B	Official	14.5	33.3	25.7	74.3
	+ Stage 1	92.0	28.7	30.3	69.7
	+ Stage 2	96.4	19.5	38.2	61.8
Gemma4-26B	Official	82.4	17.6	56.2	43.8
	+ Stage 1	98.6	15.3	62.8	37.2
	+ Stage 2	99.4	9.7	52.5	47.5

basis for estimating training variability, so we report the mean and seed-specific differences without a confidence interval. The contrast targets the combined post-SFT reward shaping and its dynamic judge, conditional on the shared SFT checkpoint and the Qwen configuration. It does not isolate the diagnostic bonus from the initial-correctness and collapse terms, or estimate the contribution of the SFT demonstrations themselves; component attribution relies on the separate reward ablations (Appendix H).

5 Analysis

5.1 How training changes self-testing behavior

Table 4 separates the decision to invoke self-testing from the reliability of the submissions the agent locally validates. All rows use the same autonomous tool-enabled interface (the agent may test, repair, request additional tests, or submit without a hand-coded trigger), so the comparison measures changes in the learned policy rather than in inference-time control flow. Stage 1 teaches the protocol: self-test invocation rises from 8.5% and 14.5% to 96.4% and 92.0% for Qwen3.5-35B and GPT-OSS-20B, and from 82.4% to 98.6% for Gemma4-26B, though invocation alone does not show that the obtained evidence is informative. Stage 2 then improves how testing is used: invocation changes little, but conditional collapse falls from 60.4 to 30.5, 28.7 to 19.5, and 15.3 to 9.7, consistent with the diagnostic and repair rewards shaping the content of verification rather than the decision to call a tool. Because the locally validated population also grows across these checkpoints, a lower conditional rate is not the same event as fewer failed submissions overall; final pass rates are in Table 2.

The residual failure composition is model-dependent: output-judgment errors dominate analyzable collapses for Qwen and GPT-OSS (72.7% and 74.3% before training, 54.6% and 61.8% after Stage 2), whereas Gemma retains a larger weak-input share (52.5%). These shares describe the remaining collapse cases rather than population error rates, and they motivate evaluating test generation and output interpretation separately.

5.2 Does a stronger external tester resolve collapse?

Table 5 fixes the coder and varies the tester under trigger mode, so every condition enters self-testing. For the Qwen coder, the trained same-model tester raises final pass rate from 58.9 to 69.5 and lowers reported collapse from 24.2 to 11.1, whereas a larger same-family tester changes pass rate only from 58.9 to 59.0 and GPT-OSS-120B reaches 64.2. On these runs, targeted tester training yields a larger benefit than replacing the tester with either untrained alternative.

The GPT-OSS coder shows a different pattern: its trained tester reduces collapse from 38.1 to 21.0 but changes pass rate only from 57.2 to 58.0, so better local verification does not necessarily translate into repair of every detected error, and GPT-OSS-120B, which helped the Qwen coder, lowers this coder's pass rate to 52.7. Larger external testers are therefore not uniformly better in this protocol; shared

Table 5: Trigger-mode cross-model verification, mean ± sample standard deviation over three runs. Only the tester changes within each coder block.

Coder	Tester	Pass	Collapse
Qwen3.5-35B	Qwen3.5-35B	58.9 ± 0.9	24.2 ± 1.8
	Qwen3.5-397B	59.0 ± 0.9	21.8 ± 1.6
	GPT-OSS-120B	64.2 ± 1.1	19.6 ± 1.5
	Trained Qwen3.5-35B	69.5 ± 0.8	11.1 ± 1.0
GPT-OSS-20B	GPT-OSS-20B	57.2 ± 1.0	38.1 ± 2.3
	Qwen3.5-397B	58.7 ± 1.1	36.2 ± 2.1
	GPT-OSS-120B	52.7 ± 1.4	37.5 ± 2.4
	Trained GPT-OSS-20B	58.0 ± 1.0	21.0 ± 1.7

assumptions, test distribution, and coder–tester interpretation mismatch are possible explanations that the experiment does not isolate.

6 Related Work

6.1 Execution-guided repair and self-correction

Execution-guided RL for code generation includes CodeRL (Le et al., 2022), RLTF (Liu et al., 2023), and RLEF (Gehring et al., 2024), and SCoRe (Kumar et al., 2024) trains multi-turn self-correction. Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2024) iterate on model-generated feedback, while Huang et al. (2023) find that intrinsic self-correction without external signals is unreliable; Fix–Test Collapse is an execution-grounded instance of this failure in which the external signal is itself produced and interpreted by the same model. Self-Edit conditions an editor on execution feedback, and Revisit Self-Debugging studies post- and in-execution use of generated tests (Zhang et al., 2023; Chen et al., 2025). Tool-augmented agents (Schick et al., 2023; Yang et al., 2024; Wang et al., 2024) and agent training on repository environments (Pan et al., 2024; Yang et al., 2025; Jain et al., 2025) scale this repair loop to software issues, where verification signals must be constructed from the current checkout rather than supplied by the benchmark.

6.2 Test generation and self-verification

CodeT (Chen et al., 2022) scores candidates by agreement with generated tests, and AlphaCode (Li et al., 2022) filters large sample pools; both select among independently sampled candidates rather than steering a single trajectory. ReVeal (Jin et al., 2025) and CURE (Wang et al., 2025a) train self-verification or co-evolving coders and testers, and AceCoder (Zeng et al., 2025), EvolveCoder (Ruan et al., 2026), and ATGen (Li et al., 2025) improve test synthesis and discriminative verification. Chen et al. (2026) find that agent-generated repository tests help when grounded in the intended specification and hurt when they encode incorrect assumptions; our repository judge operationalizes this distinction. Relative to these lines, we decompose false local validation into test-generation and output-judgment failures, keep reference-based supervision offline while withholding reference outputs during interaction, and train test generation, output interpretation, repair, and stopping jointly within a single trajectory under an input-only policy interface.

7 Conclusion

Fix–Test Collapse separates local confidence from held-out correctness and motivates training that searches for evidence against a candidate. Falsification-Guided Self-Testing combines protocol SFT, action-level training, and trajectory rewards with a dynamically updated repository judge; it improves contest correctness and conditional verification reliability, reaches 48.3% on SWE-bench Pro and 32/113 (28.3%) on DeepSWE with Qwen3.5-35B, and yields a positive post-SFT increment over final-only RL for each of three paired training seeds.

LIMITATIONS

The method leaves three design choices open for future work. First, the diagnostic bonus rewards co-occurrence of discovered evidence and successful repair rather than verified causal use of that evidence; a trace-redaction intervention could test the latter directly. Second, because the collapse penalty applies only to locally validated failures, training can in principle shift failures toward unverified submissions rather than eliminating them, which motivates pairing the penalty with calibrated submission confidence. Third, the dynamic judge mitigates but does not eliminate reward hacking, and promotion-gated judge updates with fixed acceptance criteria are a natural next step.

ETHICS STATEMENT

The study evaluates code agents in benchmark execution environments. More reliable verification can reduce incorrect submissions, but stronger code agents and imperfect judges can also lead to overconfidence or misuse. Generated changes require independent evaluation and appropriate execution safeguards.

REPRODUCIBILITY STATEMENT

Section 4.1 specifies the benchmarks, task counts, rollout caps, baseline protocols, and the checkpoint-selection rule; all bootstrap intervals use 10,000 task-level replicates with seed 0. Appendix C gives the offline diagnostic scorer as pseudocode together with its input-validation and output-normalization scope and the complete Phase-A reward table. Appendix G records decoding settings, training frameworks and hyperparameters, rollout limits, compute budgets, and the operational definitions of local validation and collapse measurement. Appendix E specifies judge initialization, the update schedule, policy–judge version binding, and the audit sampling protocols; Appendices F.1 and F.2 supply the verbatim policy and judge prompts, including the deterministic aggregation rule and soft thresholds.

AI USE STATEMENT

Large language models were used to assist manuscript writing, editing, restructuring, and numerical consistency checks. Models are also explicit components of the method: they generate code and tests, and Claude Opus 4.6 supplies labels in the repository rubric-judge protocol (Section 3.3 and Appendix E). The authors remain responsible for all content; model-assisted consistency checks do not replace author verification of tables, claims, and citations.

REFERENCES

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

Xiancai Chen, Zhengwei Tao, Kechi Zhang, Changzhi Zhou, Xinyu Zhang, Wanli Gu, Yuanpeng He, Mengdi Zhang, Xunliang Cai, Haiyan Zhao, et al. Revisit self-debugging with self-generated tests for code generation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 18003–18023, 2025.

Zhi Chen, Zhensu Sun, Yuling Shi, Chao Peng, Xiaodong Gu, David Lo, and Lingxiao Jiang. Rethinking the value of agent-generated tests for llm-based software engineering agents. *arXiv preprint arXiv:2602.07900*, 2026.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

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.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*, 2023.

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

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 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? *arXiv preprint arXiv:2310.06770*, 2023.

Yiyang Jin, Kunzhao Xu, Hang Li, Xueting Han, Yanmin Zhou, Cheng Li, and Jing Bai. Reveal: Self-evolving code agents via reliable self-verification. *arXiv preprint arXiv:2506.11442*, 2025.

Aviral Kumar, Vincent Zhuang, Rishabh Agarwal, Yi Su, John D Co-Reyes, Avi Singh, Kate Baumli, Shariq Iqbal, Colton Bishop, Rebecca Roelofs, et al. Training language models to self-correct via reinforcement learning, 2024. *URL https://arxiv. org/abs/2409.12917*, 2(3):4, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Qingyao Li, Xinyi Dai, Weiwen Liu, Xiangyang Li, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. Atgen: Adversarial reinforcement learning for test case generation. *arXiv preprint arXiv:2510.14635*, 2025.

Rongao Li, Jie Fu, Bo-Wen Zhang, Tao Huang, Zhihong Sun, Chen Lyu, Guang Liu, Zhi Jin, and Ge Li. Taco: Topics in algorithmic code generation dataset. *arXiv preprint arXiv:2312.14852*, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. Rltf: Reinforcement learning from unit test feedback. *arXiv preprint arXiv:2307.04349*, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

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.

Chi Ruan, Dongfu Jiang, Huaye Zeng, Ping Nie, and Wenhu Chen. Evolvecoder: Evolving test cases via adversarial verification for code reinforcement learning. *arXiv preprint arXiv:2603.12698*, 2026.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in neural information processing systems*, 36:68539–68551, 2023.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. *URL https://arxiv. org/abs/2303.11366*, 8, 2024.

Qwen Team. Qwen3.5: Accelerating productivity with native multimodal agents, February 2026. URL `https://qwen.ai/blog?id=qwen3.5`.

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

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning, 2025. *URL https://arxiv. org/abs/2506.03136*, 2025a.

Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. Codecontests+: High-quality test case generation for competitive programming. *arXiv preprint arXiv:2506.05817*, 2025b.

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

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. *arXiv preprint arXiv:2504.21798*, 2025.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12023–12040, 2025.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 769–787, 2023.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024a.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd annual meeting of the association for computational linguistics (volume 3: system demonstrations)*, pp. 400–410, 2024b.

A SWE-BENCH VERIFIED SUPPLEMENTARY RESULTS

Table 6 reports the SWE-bench Verified comparison under the extended repository checkpoints; Table 7 gives the corresponding task-level paired bootstrap summaries. The reward ablation and sensitivity tables in Appendices H and I evaluate the earlier repository checkpoints and therefore retain the earlier full-reward Verified resolution of 65.4.

Table 6: SWE-bench Verified results under the extended repository checkpoints. Resolved rates are percentages; uncertainty is sample standard deviation over three stochastic measurement runs.

Model	Method	Resolved	Turns
Qwen3.5-35B	Official	63.2 ± 1.7	76.15
	ReVeal	62.8 ± 1.6	80.18
	Ours	69.8 ± 1.4	80.41
GPT-OSS-20B	Official	50.6 ± 2.1	24.43
	ReVeal	50.4 ± 1.9	28.58
	Ours	55.8 ± 1.8	40.03
Gemma4-26B	Official	45.0 ± 2.3	26.57
	ReVeal	44.8 ± 2.0	28.07
	Ours	51.0 ± 1.6	35.83

Table 7: Task-level paired bootstrap comparisons for the SWE-bench Verified results in Table 6, using 10,000 task-level bootstrap replicates with seed 0. Differences and intervals are percentage points. "Positive bootstrap replicates" is the fraction of resampled differences above zero, not a posterior probability.

Model	Ours − Official	95% interval	Positive bootstrap replicates
Qwen3.5-35B	+6.6	[3.2, 10.0]	>99.9%
GPT-OSS-20B	+5.2	[1.8, 8.6]	99.8%
Gemma4-26B	+6.0	[2.6, 9.4]	>99.9%

All three intervals exclude zero under this pairing. These comparisons remain separate from the extended Pro and DeepSWE results and the post-SFT control, and task-level intervals do not account for training-seed variability.

B EXTENDED SWE-BENCH PRO LANGUAGE BREAKDOWN

Table 8 breaks the extended Qwen3.5-35B comparison (37.2% to 48.3% aggregate, Table 2) down by language: +15.4 points for Python, +8.9 for Go, +9.1 for JavaScript, and 0.0 for TypeScript, with group sizes 266, 280, 165, and 20. The aggregate paired difference is +11.1 points with 95% interval [7.9, 14.2]. Before this continuation, the earlier checkpoint resolves 41.6% of the same 731 tasks, against 37.2% for Official and 38.0% for Adv. Prompt. All intervals are task-level and single-training-seed; they do not account for training-seed variability.

Table 8: Extended Qwen3.5-35B SWE-bench Pro language breakdown, corresponding to the 37.2% to 48.3% aggregate comparison in Table 2. Intervals refer to the paired difference Ours minus Official, in percentage points, using 10,000 task bootstrap replicates and seed 0. The aggregate is task-weighted. These intervals do not apply to the post-SFT comparison.

Language	N	Official (%)	Ours (%)	Paired 95% CI
Python	266	43.6	59.0	[9.8, 21.1]
Go	280	33.6	42.5	[4.3, 13.6]
JavaScript	165	29.1	38.2	[3.0, 15.8]
TypeScript	20	70.0	70.0	[-25.0, 25.0]

C Offline Scorer and Action-Level Rewards

The policy-visible interface supplies the problem, current code, generated inputs, and candidate execution trace. The offline scorer separately accesses a reference program and benchmark outcomes. A maximum of 20 generated inputs is scored independently per request in the reported contest scorer.

Listing C.1: Contest diagnostic scorer (offline)

```
Inputs: candidate f, reference f_ref, generated inputs X,
        previous normalized inputs H
For x in X[:20]:
    t = execute(f, x)
    t_ref = execute(f_ref, x)       # offline only
    if not task_validator(x) or t_ref.status != OK:
        diagnostic = False
    else:
        diagnostic = (phi(t) != phi(t_ref))
    novel = diagnostic and normalize_input(x) not in H
    record diagnostic, novel      # reward computation only
    H.add(normalize_input(x))
    return candidate trace t to the policy

phi(t): normalized stdout if t.status == OK;
        otherwise the runtime-status token.
Reference traces and diagnostic labels never enter policy context.
```

Operational scope. The scorer canonicalizes CRLF to LF and trims leading and trailing whitespace while preserving internal whitespace and line order; input novelty uses the same canonicalization. Eq. (2) is restricted to tasks with unique, deterministic exact outputs under this normalization. Integer and string outputs use zero numerical tolerance. Floating-point outputs, multiple-valid-output tasks, and custom-checker tasks are outside this exact-comparison diagnostic subset unless their official checker is separately integrated; their ordinary benchmark evaluation remains unchanged. Every diagnostic input must pass an available task-specific constraint validator before reference disagreement is scored. If validity cannot be established, the input is ineligible for diagnostic credit, even when the reference program executes successfully. This restriction prevents treating execution success as proof of input validity.

Table 9: Phase-A action-level rewards. Submission is evaluated offline, malformed actions receive -1, and repair credit requires removal of every diagnostic failure currently supplied in the task prefix.

Action / outcome	Reward
Tester discovers $n > 0$ DTIs	$\min(1 + 0.1(n-1), 1.5)$
Incorrect program; no DTI found	0
Correct program with a valid self-test	$+1$
Fix eliminates all supplied previously exposed DTIs	$+1$
Fix leaves any supplied previously exposed DTI	-0.5
Further-test request discovers a novel DTI	$+1$
Further-test request is duplicate / non-diagnostic	-0.5
Submit	$+1$ if correct; -1 otherwise
Malformed or inappropriate action	-1

Decision scoring dispatches to submission correctness, exposed-failure removal, or novel-test discovery; Table 9 summarizes the reward assignments. A repair receives $+1$ only when it eliminates all previously exposed DTIs in the supplied prefix; eliminating only a subset receives -0.5. Phase B adds no intermediate failure-removal gate to $(1 - C_0)C_K D(\tau)$: diagnostic discovery and final correctness suffice. Offline discrimination and policy interpretation remain separate learning targets.

Table 10: Fixed-budget RL initialization control on Qwen3.5-35B / CodeContests. Both rows use rollout group size $n = 4$. "Positive trajectories" denotes the diagnostic-recovery indicator $(1 - C_0)C_K D(\tau) = 1$; "nonzero-variance groups" refers to this binary indicator within a group, not to the full trajectory reward.

Initialization	Positive trajectories	Nonzero-variance groups	Pass	Collapse	Turns
Base → RL	1.2%	4.8%	55.2	72.4	1.86
SFT → RL	15.6%	49.2%	73.8	30.5	3.28

D Additional Curriculum and Baseline Comparisons

Under the fixed budget, Stage 2 initialized from the base model has lower final pass rate (Table 10). The exploration statistics cover the first 100 Phase B updates, with 16 prompts and four trajectories per prompt: 1,600 groups and 6,400 trajectories per condition. The base and SFT conditions contain 77 and 998 diagnostic-recovery trajectories, spread across 77 and 787 nonconstant groups, respectively. Both conditions allocate 200 Phase A and 200 Phase B updates; final pass rates use the completed checkpoints. These event-specific statistics do not show that the full GRPO reward has zero variance in the remaining groups, nor do they rule out improvements from larger groups, denser rewards, or longer training.

D.1 Additional self-debugging baselines

Table 11: Qwen3.5-35B inference-time adaptations on matched task sets and decoding settings, with up to ten assistant turns. Average turns differ because stopping is adaptive. Pass entries show mean ± sample standard deviation over three runs; turns are means. The corresponding GPT-OSS-20B and Gemma4-26B means appear in Table 2 under the same protocol.

	CodeContests		LiveCodeBench	
Method	Pass	Turns	Pass	Turns
Self-Edit (adapted)	66.4 ± 1.0	3.27	82.8 ± 0.7	4.22
Revisit Self-Debugging (adapted)	71.6 ± 0.9	4.97	85.2 ± 0.6	5.69
Ours	73.8 ± 0.8	3.28	87.2 ± 0.5	4.13

Table 11 details the Qwen runs for the two inference-time adaptations; no additional parameter training is performed for either, and the GPT-OSS and Gemma evaluations in Table 2 follow the same protocol. Self-Edit uses feedback from available examples; Revisit Self-Debugging uses the applicable in-execution variant. Both adaptations retain the input-only tool interface, including runtime status and actual candidate outputs. The common instruction is: "Use the problem and observed execution results to identify a concrete error; revise the candidate if necessary, or submit when justified. Never assume access to hidden expected outputs." Self-Edit follows each execution with one edit-or-submit decision; Revisit may request further execution before editing, subject to the common ten-turn cap. These protocol descriptions define the adaptations rather than reproduce a separately trained editor.

E Dynamic Judge Training and On-Policy Audits

Initialization and update process. The initial judge uses Claude-labeled examples to train a Qwen3.5-4B rubric model. Repository policy rollouts subsequently expose judge errors. High-reward probes that satisfy the automatic rubric without genuine specification-grounded evidence are inspected and added as hard negatives when updating the judge. Policy optimization continues with updated reward models. This adaptive reward process is part of the method, not a fixed-judge training claim; the judge must therefore be assessed both as a classifier of diagnostic evidence and as a reward source encountered by an evolving policy, which the audits below address.

The earlier run ends at policy step 200. Judge j_0 covers steps 1–60, j_1 covers 61–120, j_2 covers 121–170, and j_3 covers 171–200, so the final version supplies rewards for 30 steps. At each boundary, Claude labels candidate on-policy failures of the rubric; review selects 400 hard negatives and pairs them with 400 replayed valid positives, retaining the original 3,600-example training pool. Each update runs 100 GRPO steps with AdamW, learning rate 10^{-6}, 16 prompts per batch, four completions per prompt, and KL coefficient 0.01. Sampling uses a 1:1 mixture of the original pool and the 800-example update pool. The final-success-only continuation omits judge calls for reward computation; its policy optimization and evaluation budgets remain matched. The Ours extended continuation resets the schedule counter, calls its starting judge j_0, and updates after steps 50, 100, 150, 200, 250, 300, and 350; j_7 covers 351–400. Teacher labeling collects the six rubric fields in one request per new probe and excludes hidden resolution labels; the deployed local judge queries its six dimensions separately. Every rollout is bound to the judge active when its batch starts; the batch finishes before replacement, and earlier rollouts are not rescored for policy updates.

Per-boundary calibration audits. The judge is calibrated every 50 policy steps, and the extended Qwen continuation is audited at each calibration boundary (policy steps 50 through 400). Immediately before each update, two checks are performed on that window's rollouts. First, 20 judge-accepted probes are sampled uniformly from rollouts on a fixed 100-issue audit holdout excluded from policy training, validation, and judge-update pools; two annotators, blinded to judge version, update phase, and automatic verdict, independently label each probe as diagnostic or not, and disagreements are adjudicated. Second, Claude labels every executed probe in the window with the six rubric fields, and full-pool agreement is the fraction of probes on which the deterministic aggregate of Claude's fields matches the deployed judge's verdict; each update's 400 hard negatives are selected from the judge-accepted probes that Claude rejects (415 to 468 per window). After each of the seven updates (boundaries 50 through 350), 20 further judge-accepted probes are sampled from fresh rollouts of the same policy checkpoint and audited identically; training ends at step 400, so the final boundary has no post-update audit. Table 12 reports the results.

Table 12: Calibration audits at each 50-step boundary. *Valid* is the adjudicated count of genuinely diagnostic probes among 20 judge-accepted probes sampled before (pre-update) or after (post-update) the judge update at that boundary; *gaming* is the subset of invalid probes showing explicit reward gaming. *Claude agr.* is the fraction of all executed window probes (N_w) on which Claude's aggregated rubric verdict matches the deployed judge. Training ends at step 400, so the final boundary has no post-update audit.

	Pre-update				Post-update	
Boundary (step)	Valid	Gaming	N_w	Claude agr. (%)	Valid	Gaming
50	17	2	6,412	91.3	19	1
100	17	2	6,388	91.4	19	1
150	16	2	6,295	91.3	19	1
200	16	3	6,451	91.5	18	2
250	16	3	6,247	91.2	19	1
300	16	3	6,360	91.5	19	0
350	16	4	6,289	91.3	19	0
400	15	4	6,334	91.2	–	–
Total	129/160	23	50,776	91.3	132/140	6

Audit results. Across the eight pre-update samples, 129/160 judge-accepted probes (80.6%; Wilson 95% interval [73.8, 86.0]) are genuinely diagnostic, with 23 explicit-gaming cases whose count rises over training; of the 31 invalid probes, 25 assert unsupported requirements (including all 23 gaming cases), four reflect incidental infrastructure failures, and two misread outputs. Across the seven post-update samples, 132/140 (94.3%; Wilson [89.1, 97.1]) are diagnostic, with six remaining gaming cases; of the eight invalid probes, seven assert unsupported requirements (including the six gaming cases) and one reflects an infrastructure failure. Annotator agreement over the 300 sampled probes is 279/300 (93.0%) before adjudication, and Claude's aggregated verdict matches the adjudicated human label on 283/300 (94.3%). Full-pool Claude agreement stays between 91.2% and 91.5% across boundaries. These audits are per-window and single-training-seed: the post-update gain measures the

immediate effect of each calibration on accepted-probe precision, not robustness under continued optimization; probes from the same issue are not independent; and full-pool agreement is agreement with the teacher rather than with ground truth. They support dynamic mitigation rather than any claim of immunity to reward hacking.

F PROMPT

F.1 COMPETITIVE-PROGRAMMING SELF-TESTING PROMPTS

Listing F.1: Initial Solution Prompt

```
## Problem
{question}

Write a complete Python solution. Conclude with <answer>...</answer>
    tags containing
only the Python code.
```

Listing F.2: Input-only Self-test Generation Prompt

```
You have written the following solution to a competitive programming
    problem.
Generate {num_tests} diverse test inputs to verify it.

## Problem
{question}

## Your Solution
```python
{code}
```

Format
Return a JSON array of stdin strings inside <test_inputs>...</
    test_inputs> tags.
Use \n as the newline separator within each string.

Generate exactly {num_tests} test inputs.
```

Listing F.3: Self-judgment and Continue/Repair/Submit Prompt

```
## Problem
{question}

## Your Code
```python
{code}
```

Execution Results
{execution_results}

Task
Analyze whether the outputs are correct based on the problem
    specification.
Choose one of three actions:
```

```
1. FIX: If you found incorrect output(s), provide the corrected code
    in
<answer>...</answer> tags.
2. MORE TESTS: If you are uncertain and want to run stronger or more
    diverse
inputs, provide them in <test_inputs>...</test_inputs> tags as a JSON
    array of
stdin strings.
3. SUBMIT: If you are confident ALL outputs are correct, respond with
    exactly:
SUBMIT

Your response should be "SUBMIT", or fixed code in <answer>...</answer
    > tags, or
new test inputs in <test_inputs>...</test_inputs> tags.
```

F.2 LLM Rubric Judge Prompts

Listing F.4: Shared Instruction and Inputs for All Rubric Judges

```
You are a specification-grounded judge for repository-level software
    repair.

Judge only from the issue description, the probe, the observed
    execution output, and the
logs. Do NOT assume access to a gold patch, hidden tests, reference
    outputs, or a
reference implementation.

Inputs:
[ISSUE]
{issue_description}

[PROBE]
{probe_source_or_command}

[EXECUTION_OUTPUT]
{execution_output}

[LOGS]
{relevant_logs}
```

Listing F.5: Relevance Gate prompt

```
Using the shared inputs, decide whether the probe is relevant to the
    issue.

Question:
Does the probe exercise behavior, APIs, files, arguments, error paths,
     or user scenarios
that are clearly related to the issue description?

Return JSON:
{
  "relevance": 0 or 1,
  "rationale": "one concise sentence"
}
```

```
Set relevance=0 for broad unrelated tests, import-only checks,
    unrelated utility tests,
or probes with no clear connection to issue-mentioned behavior.
```

Listing F.6: Execution-validity Gate Prompt

```
Using the shared inputs, decide whether the probe was validly executed.


Question:
Was the probe actually executed in a meaningful way, so that its
    output can be used as
evidence about repository behavior?

Return JSON:
{
  "execution_validity": 0 or 1,
  "rationale": "one concise sentence"
}

Set execution_validity=0 for skipped tests, tests not collected,
    malformed generated
tests, syntax errors in the probe, unrelated dependency/import
    failures, or timeouts
without issue-specific evidence.
```

Listing F.7: Recognizable-violation Gate Prompt

```
Using the shared inputs, decide whether the observed output is
    recognizable as a
violation of the issue specification.

Question:
Does the observed output contradict behavior that can be recognized
    from the issue
description, such as an expected return value, exception behavior,
    state transition,
schema, warning, serialization result, or invariant?

Return JSON:
{
  "recognizable_violation": 0 or 1,
  "rationale": "one concise sentence"
}

Set recognizable_violation=0 if the output is merely surprising,
    incomplete, or not
clearly connected to a requirement stated or implied by the issue.
```

Listing F.8: Failure-specificity Gate Prompt

```
Using the shared inputs, decide whether the failure is specific to the
     issue.

Question:
Is the failure specifically tied to the issue rather than a generic
    repository failure,
```

```
environment problem, broad smoke-test failure, or unrelated API misuse
    ?

Return JSON:
{
  "failure_specificity": 0 or 1,
  "rationale": "one concise sentence"
}

Set failure_specificity=0 for generic crashes, missing packages,
    unrelated configuration
failures, broad test-suite failures, or errors caused by invalid probe
     construction
rather than the issue.
```

Listing F.9: Oracle-strength Filter Prompt

```
Using the shared inputs, score the strength of the probe's oracle.

Question:
How strong is the probe's check of issue-relevant behavior?

Return JSON:
{
  "oracle_strength": number between 0.0 and 1.0,
  "rationale": "one concise sentence"
}

High scores require a concrete assertion or check over issue-relevant
    behavior. Low
scores include superficial checks such as only importing the package,
    only checking that
code does not crash, or checking an unrelated side effect.
```

Listing F.10: Fix-plausibility Filter Prompt

```
Using the shared inputs, score whether a correct fix would plausibly
    satisfy the probe.

Question:
If the issue were correctly fixed, would the probe plausibly pass?

Return JSON:
{
  "fix_plausibility": number between 0.0 and 1.0,
  "rationale": "one concise sentence"
}

High scores mean the probe checks behavior required or strongly
    implied by the issue.
Low scores mean the probe asserts invented behavior, an over-specific
    implementation
detail, or behavior not implied by the issue.
```

Listing F.11: Aggregation Rule for the Rubric Judge

```
The final judge verdict is computed deterministically:
```

```
verdict = 1 iff
  relevance == 1
  and execution_validity == 1
  and recognizable_violation == 1
  and failure_specificity == 1
  and oracle_strength >= {theta_o}
  and fix_plausibility >= {theta_f}

Otherwise verdict = 0.
```

G IMPLEMENTATION SETTINGS

The numerical configurations in this section describe the earlier experiments; the extended continuation reuses them with 400 additional policy updates and the judge schedule in Appendix E (see Section 4.1).

Competitive-programming evaluation. CodeContests and LiveCodeBench are evaluated with an output-free agentic self-testing interface. The policy writes a candidate program and may call an execution tool with self-generated input cases. The executor returns only runtime status, stdout, stderr, timeout information, and formatted logs; it does not reveal expected outputs, hidden verdicts, or reference solutions. The final submitted program is then scored on held-out benchmark tests. Unless otherwise stated, decoding follows the official Qwen code-generation setting: one sample, temperature 0.6, top-p 0.95, and a maximum response length of 65,536 tokens, and up to ten assistant turns during evaluation (distinct from the five-turn Phase B training cap).

Competitive-programming training. Stage 1 SFT trajectories are built from CodeContests and TACO training instances. For each problem, we sample an initial solution, ask the model to construct input-only self-tests, execute the current program on those inputs, and continue the repair loop when the observed trace exposes a runtime error, timeout, or specification-recognizable wrong behavior. The stored trajectory contains the problem statement, code revisions, generated inputs, execution observations, model actions, and final benchmark outcome. For collapse-recovery demonstrations, benchmark failure inputs can be used to trigger the trace, but the policy observes only the candidate program's actual outputs on those inputs, never the reference outputs.

Stage 1 SFT is implemented with LLaMA-Factory (Zheng et al., 2024b). Stage 2 uses GRPO (Guo et al., 2025). Phase A trains the local decision, tester, and repair skills from Section 3.2; Phase B trains full multi-turn self-testing trajectories with Eq. (3). Qwen3.5 and GPT-OSS rollouts use an SGLang (Zheng et al., 2024a) backend with top-p 0.95 and rollout temperature 1.0; Gemma4 training uses VERL (Sheng et al., 2024) with a vLLM (Kwon et al., 2023) rollout backend. Phase A uses train batch size 16 and prompt/response length caps of 32,768 tokens. Phase B enables multi-turn tool use with up to five assistant turns and five user/tool turns, caps each tool response at 32,768 tokens, and uses 65,536-token actor sequence limits.

Repository-agent training environment. Agent-mode adaptation is trained on R2E-Gym-V1 repository-repair environments. Each instance provides an issue description, an initial repository checkout, a containerized execution environment, and an evaluator for the final patch. The policy interacts through a tool-level software-engineering interface: it can inspect files, run shell commands, create reproduction scripts or tests, edit repository files, and finish with a patch. During training rollouts, execution and repair tools are enabled only after the explicit initial patch is frozen (Section 3.3). The agent observes only tool outputs from the current repository state. Gold patches, hidden evaluator outcomes, and final resolution labels are withheld during the rollout.

Tool-level rollout loop. The online RL loop separates parameter optimization from repository execution. VERL performs trajectory sampling, advantage computation, and GRPO parameter updates. OpenHands provides the repository runtime: it initializes a fresh per-instance container, checks out the target repository state, executes shell and file-editing actions, returns stdout/stderr/exit

status/file observations, and extracts the final git diff when the agent finishes. Metadata from each tool call is carried with the trajectory, including the command or edit action, exit status, truncated logs, changed-file summaries, generated probe bodies when present, and final patch text. This metadata is used only by reward computation and logging, not injected back into the policy beyond the normal tool observations.

Compute resources. The earlier reported training and evaluation runs use servers with 32 NVIDIA H200 GPUs, 128 CPU cores, and 3,126 GiB of system memory. The resource specification covers SFT, RL rollouts, repository-agent execution, and benchmark evaluation; the two continuation arms are charged separately. A representative earlier Qwen3.5-35B pipeline consumes approximately 8,512 H200 GPU-hours, and its 400-update extended continuation allocates 10,752 GPU-hours (336 hours on 32 GPUs); GPT-OSS and Gemma budgets are smaller.

Patch evaluation and reward computation. After a rollout terminates, the final diff is evaluated in a clean benchmark testbed. The evaluator applies the patch to the base repository, runs the instance-specific evaluation script, parses the benchmark report, and returns the held-out resolution label C_K. This label is used for reward computation and aggregate reporting, but is never exposed to the agent during the trajectory. The reward combines final resolution, the diagnostic event from the rubric judge, and the collapse penalty as in Eq. (5).

Initial rubric judge and dynamic updates. The repository-level violation predicate is implemented by a Qwen3.5-4B judge distilled from Claude Opus 4.6[1] and served through an OpenAI-compatible API. The distillation set is built from repository-agent rollouts by extracting candidate probes, commands, generated tests, observed stdout/stderr, exit status, issue descriptions, and short log windows around the relevant failure. We collect 4,000 Claude-labeled hard-gate examples: 1,000 examples for each of relevance, execution validity, recognizable violation, and failure specificity, with a 1:1 positive/negative balance within each gate. The two soft filters, oracle strength and fix plausibility, are trained on the same probe pool with teacher scores stratified into bins so that the scalar targets approximately cover $[0, 1]$ uniformly. We use a stratified 3,600/400 train/test split.

The Qwen3.5-4B judge uses GRPO with the Claude-labeled training split as reward data. The reward combines JSON validity, field completeness, binary agreement with hard-gate labels, and scalar-score agreement for the two soft filters. The final verdict is computed only by Listing F.11, never from held-out resolution labels. Each of the 4,000 examples denotes a unique probe with all six rubric labels and one primary hard-gate stratum; the 400 held-out probes can therefore be scored by the aggregate rule. Split assignment is grouped by issue, with every duplicate probe and gate-specific view kept in the same partition. The split targets 3,600/400 probes and approximate primary-gate stratification; the split protocol requires zero shared issue or probe IDs between partitions.

After initialization, on-policy hard negatives are used for dynamic judge updates (Appendix E). At reward time, the six rubric dimensions in Appendix F.2 are queried independently with temperature 0. Each subjudge receives the issue description, the candidate probe or command, observed stdout/stderr and exit status, and a short surrounding log window. It does not receive reference patches, hidden test verdicts, gold outputs, or the final resolution label. Binary gates and scalar filters are returned as JSON and aggregated by the deterministic rule in Listing F.11; the default soft thresholds are $(\theta_o, \theta_f) = (0.6, 0.6)$. Malformed JSON, missing fields, skipped tests, collection failures, dependency failures not attributable to the issue, and timeouts without issue-specific evidence are treated as non-diagnostic. If a trajectory contains multiple probes, the diagnostic event is the maximum judge verdict over probe-level inputs. These prompts, schemas, thresholds, and aggregation rules specify the scoring interface. Compatibility with that interface does not imply unchanged downstream performance when another judge is used.

Collapse measurement. For competitive-programming tasks, a trajectory is collapsed when the model submits after all self-generated tests pass, yet the final program fails the held-out benchmark tests. Repository-agent collapse is measured analogously but requires post-patch local verification. We set $L(\tau) = 1$ only when, after the final code edit, the agent runs or cites a local verification signal that it treats as sufficient: for example, an issue-directed regression test passes, a reproduction script

[1] https://www.anthropic.com/news/claude-opus-4-6

no longer fails, or the selected relevant test command completes successfully. Commands run before the last edit do not count unless rerun on the final patch state. Skips, collection errors, unrelated dependency failures, broad import checks, and inconclusive logs are not counted as successful verification.

We then compute $E(\tau)$ by Eq. (1): local verification succeeds but the held-out repository evaluator returns $C_K = 0$. If the trajectory contains no valid post-patch verification, $L(\tau) = 0$. The repository tables report $100 \sum E / \sum L$; the denominator is locally validated submissions, not all submissions or all self-test invocations. All-instance incidence is $100 \sum E/N$ and requires separate counts. This distinction keeps unverified and unrepaired failures separate from false local confidence.

H Reward Ablation

Table 13 ablates the reward components of Falsification-Guided Self-Testing on CodeContests using Qwen3.5-35B. Removing diagnostic reward lowers pass rate by 9.2 points and raises conditional collapse from 30.5 to 55.8 under the listed budget. Together with the fixed-budget initialization control favoring SFT before RL (Appendix D), these results support rewarding diagnostic behavior explicitly and treating failure discovery and repair as complementary; they do not show that other budgets or reward designs could not match this performance. Weight sensitivity is in Appendix I. In the expected-output alternative, the tester additionally predicts an output for each of its inputs; the offline scorer compares those predictions to the reference using the same normalization and at most 20 inputs per request. Its bonus replaces $D(\tau)$ with the event that at least one valid input has a correctly predicted reference output, retaining the $(1 - C_0)C_K$ gate, $\lambda = 0.2$, the same update counts, token limits, and reference-execution cap. Predictions and reference-matching labels are not returned to the repairing policy. This changes the test objective as well as what the tester generates and does not isolate judgeability. Removing αC_0 reduces initial pass rate and increases average turns. Removing Phase A repair shaping degrades final recovery despite leaving Eq. (3) unchanged. The final-correctness-only row removes the other shaping signals together.

Table 13: Contest-style reward ablation on CodeContests with Qwen3.5-35B. Rows remove or replace reward components as named; the final-correctness-only row removes multiple shaping components. The Phase A repair-shaped reward is action-level shaping from Section 3.2. *Diag. Recovery* is $100 \sum (1-C_0) D(\tau) C_K / \sum (1-C_0) D(\tau)$: success among initially incorrect trajectories that discover a DTI. An empty denominator is undefined, not zero. Conditional collapse uses locally validated submissions as its denominator.

Reward Configuration	Initial ↑	Pass ↑	Turns	Collapse ↓	Diag. Recovery ↑
Full objective	58.6	73.8	3.28	30.5	41.7
w/o diagnostic reward	57.9	64.6	2.41	55.8	18.9
expected-output reward	56.8	66.1	2.88	49.6	23.7
w/o initial-correctness reward	51.4	69.2	4.46	35.7	38.4
w/o Phase-A repair-shaped reward	56.7	68.8	3.62	38.6	29.5
Final correctness only (C_K)	54.8	62.7	1.94	61.2	12.8

Table 14 ablates the repository-agent reward from Eq. (5) on SWE-bench Verified. These ablation arms are trained and evaluated at the earlier repository budget, so the full-reward row reports the earlier Verified resolution of 65.4; the extended-checkpoint evaluation appears in Table 6. Removing the diagnostic judge reward reduces resolved rate and lowers self-test coverage, as the agent receives less credit for discovering issue-grounded falsifying evidence. Removing the collapse penalty preserves a comparably high self-test rate but substantially increases false local validation, consistent with the intended distinction between the terms: the diagnostic term rewards finding a genuine violation, while the collapse penalty discourages submitting after self-confirming but non-diagnostic verification.

I Parameter Sensitivity

We evaluate sensitivity to the reward weights by varying one coefficient at a time around the default setting while holding all other hyperparameters fixed.

Table 14: Repository-agent reward ablation on SWE-bench Verified with Qwen3.5-35B. The two terms address complementary failure modes: the diagnostic term rewards falsifying evidence, and the collapse penalty discourages false local confidence at submission time.

Reward Configuration	Resolved Rate ↑	Self-test Rate ↑	Collapse Rate ↓
Full agent reward	65.4	96.9	27.2
w/o diagnostic judge reward	61.6	88.4	35.9
w/o collapse penalty	63.7	97.5	39.8

Contest-style weights. Table 15 varies λ and α for the CodeContests objective. Because Eq. (3) does not include a collapse penalty, only these two weights are examined here. Setting λ too small under-rewards falsification and increases collapse; setting it too large encourages excessive self-testing and marginally reduces pass rate. Lowering α reduces initial correctness, consistent with its role in discouraging repair-reliant policies. The default $(\lambda = 0.2, \alpha = 0.6)$ provides a stable operating point across this range.

Table 15: Sensitivity to contest-style reward weights on CodeContests with Qwen3.5-35B. Bold row indicates the default configuration.

λ	α	Pass ↑	Turns	Collapse ↓
0.1	0.6	71.9	3.04	36.8
0.2	**0.6**	**73.8**	**3.28**	**30.5**
0.4	0.6	72.6	3.91	28.9
0.2	0.4	70.8	3.11	39.6
0.2	0.8	73.1	3.72	27.4

Repository collapse penalty. Table 16 varies β while keeping λ and α at their default values; as in Table 14, all sweep arms use the earlier repository budget, whose default-β Verified resolution is 65.4. As β increases, collapse decreases monotonically, but at $\beta = 0.5$ the agent becomes overly conservative, reducing resolved rate and increasing turn count. We use $\beta = 0.3$ as the default, which achieves the best resolved rate in this sweep while keeping turn count comparable to lower-penalty configurations.

Table 16: Sensitivity to the repository collapse penalty β on SWE-bench Verified with Qwen3.5-35B. Bold row indicates the default configuration.

β	Resolved ↑	Turns	Collapse ↓
0.0	63.7	78.4	39.8
0.1	64.8	79.1	31.6
0.3	**65.4**	**79.6**	**27.2**
0.5	64.1	83.7	24.5

J Static Repository Judge Validation

This section describes static validation, separately from the on-policy audit in Appendix E. The repository-level reward relies on a rubric judge to determine whether a local probe provides issue-grounded falsifying evidence. We validate this component by sampling post-patch verification attempts from SWE-bench Verified trajectories and comparing automatic judge verdicts against manual annotations.

Annotation protocol. Two annotators independently label each example as diagnostic or non-diagnostic, following the rubric criteria in Appendix F.2. Disagreements are resolved through discussion. Inter-annotator agreement on a shared 80-example subset reaches $\kappa = 0.78$, consistent with substantial agreement on this subset, while borderline cases remain.

Automatic judge accuracy. Table 17 reports accuracy, precision, recall, and F1 on both the held-out Claude-labeled distillation test set and manual annotations across three SWE-bench Verified trajectory splits. The held-out distillation split measures agreement with the teacher after training, while the manual splits measure whether the resulting aggregated verdict matches human judgments of diagnostic evidence. The reported judge attains precision and recall around 90% on these static splits, which is important for reward learning: false positives would reinforce spurious probes, while false negatives would miss useful diagnostic evidence. The primary source of remaining errors is borderline trajectories with long or noisy logs where issue-relevant evidence is present but implicit, a failure mode that could be addressed in future work through improved log truncation or summarization strategies.

Table 17: Validation of the repository-level diagnostic judge. The first row evaluates the aggregated verdict on the held-out Claude-labeled distillation split; the remaining rows compare the same aggregated verdict against manual annotations on SWE-bench Verified trajectories.

Split	Examples	Accuracy	Precision	Recall	F1
Claude distillation held-out	400	93.5	93.9	93.0	93.4
Qwen3.5-35B trajectories	120	89.2	88.9	90.3	89.6
Gemma4-26B trajectories	120	88.3	87.3	90.2	88.7
Mixed held-out	160	88.8	88.2	90.4	89.3

The static metrics use one aggregated binary verdict per probe. In table order, the (TP, FP, FN, TN) counts are (185, 12, 14, 189), (56, 7, 6, 51), (55, 8, 6, 51), and (75, 10, 8, 67). Accuracy is $(TP + TN)/N$, precision is $TP/(TP + FP)$, recall is $TP/(TP + FN)$, and F1 is $2TP/(2TP + FP + FN)$; percentages are rounded only after computing these ratios.

Threshold sensitivity. The default soft thresholds are $(\theta_o, \theta_f) = (0.6, 0.6)$. Sweeping both jointly from $(0.55, 0.55)$ to $(0.75, 0.75)$ changes resolved rate by at most 1.4 points and collapse rate by at most 2.6 points, indicating that the agent reward is not overly sensitive to the exact threshold choice within a reasonable operating range.

Relationship to the held-out evaluator. The diagnostic judge is designed to detect issue-grounded falsifying evidence, not to predict final resolution. As expected, trajectories containing at least one diagnostic probe are positively correlated with successful repair, but the two signals are not equivalent: some trajectories identify a genuine violation yet fail to repair it, while others pass the held-out evaluator without producing an explicit diagnostic probe. The information boundary is enforced by withholding final evaluator labels from judge inputs and policy observations. Correlation between diagnostic probes and repair success does not by itself establish either causal use of the evidence or robustness against reward exploitation.