TEACHING AGENTS TO CODE RELIABLY

Anonymous authors
Paper under double-blind review

ABSTRACT

Autonomous coding agents solve repository issues by reading code, running commands, editing files, and submitting patches. Extra inference-time compute yields gains only when it produces a useful repair and supplies reliable evidence for choosing one. Three behaviors decide both, and we argue they are teachable rather than byproducts of scale, so a policy can carry them instead of a scaffold. Location diversity remains narrow, since attempts return to the same site and extra samples add no coverage. Edit diversity is left unexploited, since methodologies that differ resolve complementary issues no single run reaches. Verification misleads, since a test the agent writes for its own patch accepts many incorrect ones. Directing search by execution feedback and scoring each patch against its own reverted tree resolves 52.8% of SWE-bench Verified using 48.1% of the agent-steps an eight-sample baseline spends. Training moves these behaviors into the policy. On the 270 issues held out from SFT and RL training, weighted supervised fine-tuning raises pass@1 from 31.9% to 35.2% and pass@8 from 46.7% to 51.1%. A reinforcement objective then trains the verifier against gold-labeled repairs and incorrect variants, crediting the assertions that detect them. It raises pass@1 to 43.0% and pass@8 to 60.7%, lifts verifier precision from 26.8% to 41.7%, and more than halves false acceptance. Resolution improves on two of three out-of-distribution suites and verifier precision on all three, and the gains hold at 7B, 14B, and 30B against published coder baselines.

1 INTRODUCTION

Autonomous coding agents resolve repository issues through code inspection, editing, and test execution (Jimenez et al., 2024; Yang et al., 2024). Repeated attempts and verification can improve resolution by increasing inference-time computation (Brown et al., 2024; Snell et al., 2024; Han et al., 2026). Their benefit depends on whether additional attempts produce correct repairs and whether the agent can identify them. Attempts can revisit the same edit location or repeat an incorrect repair. A test generated by the agent can share the patch's mistaken interpretation of the issue and accept incorrect behavior (Ahmed et al., 2025). More candidates do not guarantee a correct selected patch.

We distinguish candidate coverage from selection quality. Coverage is the fraction of issues for which the candidate set contains at least one correct repair. Selection quality is the fraction of those issues for which the agent returns a correct candidate. Oracle pass@8 measures coverage across eight attempts, whereas the accuracy of the selected patch depends on both quantities. Verification also guides the search by determining whether the agent revises a patch, explores another location, or stops. The verifier therefore affects both the candidates generated and the patch returned.

We hypothesize that training can improve repair search and test discrimination under a fixed backbone and scaffold. Our diagnostic study examines edit locations, repair approaches, and verification decisions on SWE-bench Verified. Repeated self-verification produces few changes to the submitted patches, while different problem-solving preambles produce complementary repairs. Proactive localization visits more locations than reactive search but resolves fewer issues (Section 2.1). These observations motivate training the agent to use execution feedback when choosing its next edit and to evaluate tests by their ability to detect incorrect repairs. Execution provides evidence beyond the agent's judgment of its own output (Huang et al., 2024).

Our approach combines a test-time diagnostic with two training stages. The diagnostic pairs exploration of edit locations and repair approaches with differential verification, which runs a reproduction

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.

on the patched and reverted repository. Weighted supervised fine-tuning uses the resulting trajectories, with weights determined by repair-quality tier, exploration score, and agent-step count. Reinforcement learning then evaluates generated tests against oracle-rejected patch variants and sibling repairs. On the 270 SWE-bench Verified issues held out from SFT and RL training, the two training stages raise greedy pass@1 from 31.9% to 43.0% (Table 7). Across eight attempts, oracle coverage rises from 46.7% to 60.7% and selected-patch accuracy from 33.5% to 50.5%, with selection efficiency increasing from 0.72 to 0.83 (Table 3). These measurements separate improvements in candidate coverage from improvements in selection. On the external benchmarks, RL raises pass@1 over SFT from 15.7% to 20.2% on SWE-bench Pro and from 24.2% to 31.8% on SWE-PolyBench, while DeepSWE full resolution remains zero at this scale, though its partial-credit measures rise (Table 6).

2 METHODOLOGY

This section describes the test-time search procedures, weighted trajectory supervision, and reinforcement learning objective. Figure 1 illustrates exploration and verification within one repair trajectory.

2.1 TEST-TIME SCALING METHODS ON CODING AGENTS

The diagnostic uses all 500 SWE-bench Verified issues to analyze repair and verification failures. We select 230 issues from this population for training dataset curation and hold the remaining 270 out of SFT and RL training. All search procedures run on the same backbone and scaffold. Table 1 reports resolution, agent-steps, attempts per instance, and diversity. Attempt limits and stopping rules differ by procedure, so each comparison reflects an accuracy and cost trade-off.

We compare single attempts, repeated self-verification, eight repair preambles, proactive localization, and reactive bans on failed edits or locations. The preambles reach an oracle union of 305/500 issues, while repeated self-verification leaves 78 to 85% of audit passes without a code change. Appendix A.1 describes the procedures and scoring rules.

Differential-verification search pairs up to eight locations and repair approaches. It runs each reproduction on the patched and reverted trees, checks repository tests and changed-line execution, and uses the resulting score to stop or prune search. A reproduction that passes on both trees receives zero credit. Search uses repository execution rather than hidden grader feedback. It resolves 52.8% of issues against 46.8% for the eight-sample control, using 48.1% of its agent-steps (Table 1). Appendix B.11 reports component ablations.

2.2 SUPERVISED FINE-TUNING ON WEIGHTED AGENT TRAJECTORIES

We train the policy to imitate repair trajectories weighted by outcome quality, exploration, and agent-step count. The training interface matches the inference interface. Some examples rely on verifier outcomes rather than gold labels, so the corpus can contain incorrect repairs.

Corpus construction. We curate the SFT corpus from the 230 issues selected after the all-500 failure analysis in Section 2.1. Trajectories from the remaining 270 issues are excluded from SFT and RL training. The corpus includes the six search procedures in Table 1 and the methodology-diversity runs, generated with Qwen3-Coder-30B-A3B-Instruct under mini-SWE-agent. Each example records the conversation, final patch, edit locations, agent-step count, verifier outcome, available gold label, and behavior tags. Trajectories are ranked within each `instance_id` by the weight defined below, with ties resolved by fewer steps. The retained corpus contains 5,061 examples from 230 issues.

The corpus includes contrastive-edit examples pairing a correct repair with an incorrect candidate at the same location, and reproduction examples distinguishing buggy from fixed behavior. Contrastive examples receive additional weight when the incorrect candidate passed the verifier. These examples are included in the 5,061 total.

Quality tiers. Each example receives the weight of the first tier it matches. Tier $T1$ has weight 1.0 and requires a gold-resolved patch that was selected or received a verifier score of at least 1.0. Tier $T2$ has weight 0.6 and requires either a verifier score of at least 1.0 or a gold-resolved patch with a

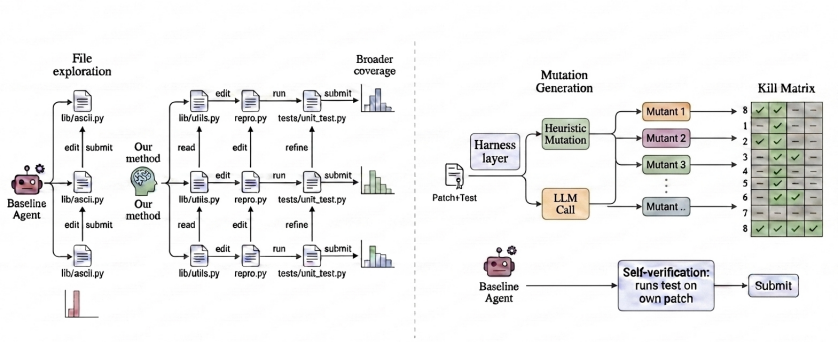

Figure 1: Two behaviors, one reward. *Left, exploration.* The baseline resubmits the same site (`lib/ascii.py`), so coverage stays narrow, while parallel chains that read first each reach the true defect (`lib/utils.py`), write a reproduction, and submit. *Right, verification.* The harness creates incorrect patch variants and records which assertions detect each variant. The gate opens only when the test kills every variant, so the reward pays for discrimination.

Experiment	Accuracy /500	Accuracy rate	Compute total steps	Compute att./inst	Compute early-stop	Compute steps/solve	Diversity loc.	Diversity edit
Self-Verification	184	36.8%	189,371	3.99	0 / 500	1,029	1.10	1.31
Single-Axis Diagnosis	189	37.8%	**41,332**	1.00	–	**219**	–	–
Proactive Location Diversity	210	42.0%	138,424	4.00	0 / 500	659	**2.46**	**3.26**
Blind Retry (baseline)	213	42.6%	139,632	–	–	656	–	–
Reactive Edit Diversity	217	43.4%	131,417	3.67	65 / 500	606	1.84	2.79
Reactive Location Diversity	225	45.0%	131,680	3.66	66 / 500	585	1.86	2.73
Independent Retry (baseline)	221	44.2%	130,564	–	–	591	–	–
Sample Scaling	234	46.8%	275,943	8.00	0 / 500	1,179	∼2.0	–
Differential Verification	**264**	**52.8%**	132,697	3.12	**102 / 500**	**503**	1.75	–

Table 1: Efficiency across the test-time procedures on the 500 issues, all-attempts basis, so steps/solve equals total steps over bugs resolved. Dark yellow marks the best value in a column. The highlighted row is our combined procedure, which is strongest on accuracy at near the lowest cost. Single-axis diagnosis is the one-attempt reference, cheapest overall though not the lowest resolution. Methodology diversity is reported in the text and not here, because its 61.0% is an oracle union over eight preambles rather than a rate any single deployable procedure reaches. Blind Retry, Independent Retry, and Sample Scaling are baselines in italics. A dash marks not applicable.

verifier pass. Tier $T3$ has weight 0.25 and requires a non-empty patch with a weaker positive verifier signal. All remaining examples receive weight 0. Only $T1$ requires a gold label.

Exploration and efficiency weights. We multiply the tier weight by exploration and efficiency factors.

$$w = \underbrace{w_{\text{tier}}}_{\text{quality}} \cdot \underbrace{\big(1+\lambda\,\tilde{\nu}\big)}_{\text{exploration}} \cdot \underbrace{\Big(1+\beta\big[1-\min\big(1,\tfrac{\text{steps}}{S_{\text{ref}}}\big)\big]\Big)}_{\text{efficiency}}, \qquad \lambda = 1.0,\ \beta = 0.5,\ S_{\text{ref}} = 120. \tag{1}$$

The effective novelty score $\tilde{\nu}$ has behavior-dependent floors of 0.6 for post-ban recovery and self-correction, 0.5 for localization-win and synthesis-only solves, and 0.4 for methodology-unique solves. The efficiency factor decreases from 1.5 at zero steps to 1.0 at $S_{\text{ref}} = 120$ steps and remains 1.0 for longer trajectories. These factors adjust the contribution of each example to supervised training.

2.3 Reinforcement Learning for Discriminative Verification

The SFT objective imitates trajectories without ever rewarding a test for rejecting an incorrect repair (Shum et al., 2025). We add a reinforcement objective that evaluates generated tests against incorrect programs, which measures which incorrect behaviors a test detects.

Generated objects. For each training issue, the policy produces a patch P and a test T containing assertions $\{a_1, \ldots, a_n\}$. We draw k rollouts per issue. The other $k-1$ patches in the group are

sibling repairs, which provide additional candidates for test evaluation. A gold execution oracle $\mathcal{G}$ determines whether a patch satisfies the training issue's evaluation tests.

Oracle access and rollout feedback. The policy observes repository files, command results, and the `pytest` output of T on P. Revision rounds return surviving patch variants as counterexamples that the policy can use to revise its patch or test. The policy does not receive gold tests, gold verdicts, assertion-credit scores, or group advantages. Oracle labels are used by the scoring procedure described below. The completed rollout receives a scalar reward, from which we compute trajectory and assertion-token advantages for the training update.

Mutation evaluation. The harness generates variants from the current patch using deterministic operators and a frozen model distinct from the policy. The operators cover boundary, value, logic, control-flow, and effect faults. Model-generated variants are produced during the run rather than retrieved from a cache. A line trace checks that the test executes the patched lines. A variant counts as detected when an assertion fails. Uncaught errors receive no detection credit.

The scoring set S contains oracle-rejected variants and oracle-rejected sibling repairs. Candidates that pass the gold tests are excluded from this negative set. Passing the gold tests does not establish semantic equivalence, but it prevents those candidates from being scored as known incorrect repairs. We record the detected subset as $\mathrm{Kill}(T)$. Appendix A.6 gives a worked example.

The repair loop allows up to three revision rounds and applies a penalty for additional rounds. Exploration terms measure group diversity across edited files and assertion targets, coverage of fault classes, and repetition of test templates. These terms supplement the patch-correctness and test-discrimination signals.

Post-rollout reward. We score each completed rollout with

$$R_{\text{traj}} \;=\; w_c\,\textsc{Correct} \;+\; w_v\,\textsc{Verif} \;+\; w_e\,\textsc{Edit} \;+\; w_p\,\textsc{Process}, \tag{2}$$

We use $w_c = 3.0$, $w_v = 1.0$, $w_e = 0.4$, and $w_p = 0.2$. The correctness term is 1 when P satisfies $\mathcal{G}$ and 0 otherwise. The verification term measures detection of the scored negatives and agreement with the oracle.

$$\textsc{Verif} \;=\; \underbrace{\tfrac{|\mathrm{Kill}(T)|}{|S|}}_{\text{dense slope}} \;+\; \underbrace{\lambda_{\text{fp}}\,\mathbf{1}[\mathrm{Kill}(T) = S]}_{\text{fixpoint bonus}} \;+\; \underbrace{h(T)}_{\text{honesty}} \tag{3}$$

The detection fraction provides partial credit, and $\lambda_{\text{fp}} = 0.5$ adds a bonus when the test detects every variant in S. The agreement term $h(T)$ is -0.5 for false acceptance, -0.25 for false rejection, and 0 when the test verdict agrees with the oracle. The resulting verification term lies in $[-0.5, 1.5]$. The edit-placement and reproduce-first terms lie in $[0, 1]$ and contribute only when $\textsc{Verif} > 0$ or $\textsc{Correct} = 1$.

The weighted auxiliary terms have a combined range of at most 2.6, below the correctness weight of 3.0. Every correct rollout therefore receives a higher scalar reward than every incorrect rollout. The auxiliary terms determine the ordering within each correctness class. This ordering concerns the scalar reward, not the sign of every group-relative advantage.

Detecting all scored variants does not establish correctness beyond that set. For a fixed test and M independent draws from a fixed mutant distribution with no survivors, the one-sided 95% upper confidence bound on the survival probability is $\rho \le 1 - 0.05^{1/M}$. Table 17 reports training completion and held-out detection. Appendix A.4 specifies empty negative sets, invalid executions, and other boundary cases.

Assertion credit. We optimize with Group Relative Policy Optimization (GRPO). Rewards normalized within each issue's rollout group give the trajectory advantage A_{traj}. The kill matrix $K_{m,i}$ records whether assertion i detects mutant m. We allocate credit to detecting assertions using

$$\tilde{c}_i \;=\; \sum_{m:\,K_{m,i}=1} \frac{\omega_{\text{cls}(m)}}{\kappa_m}, \qquad c_i \;=\; \frac{\tilde{c}_i}{\sum_j \tilde{c}_j} \tag{4}$$

Here, κ_m is the number of assertions that detect mutant m, and $\omega_{\text{cls}(m)}$ is its fault-class weight. Each detected mutant divides one class-weighted unit among its detecting assertions. Normalization gives c_i. We set $s_t = c_{i(t)}$ for tokens in assertion $i(t)$ and $s_t = 0$ elsewhere. The token advantage is

$$A_t \;=\; A_{\text{traj}} \;+\; \beta\, G\, A^{+}_{\text{traj}}\, s_t, \qquad A^{+}_{\text{traj}} = \max(A_{\text{traj}}, 0), \quad G \in \{0, 1\} \tag{5}$$

The gate $G = 1$ when the patch resolves the issue or the test detects every scored mutant. The assertion bonus is applied when this gate is satisfied and $A_{\text{traj}} > 0$. Complete detection can therefore enable assertion credit for an incorrect patch. The bonus rewards detection of scored negatives and does not certify the accompanying patch. The trajectory-level agreement term penalizes false rejection, but the unsigned assertion credit contains no penalty for rejecting correct alternatives. Patch tokens retain the trajectory advantage. Setting $\beta = 0$ removes the assertion bonus.

The unnormalized credit $\tilde{c}_i$ is the Shapley allocation of a weighted coverage game in which an assertion set receives the total weight of the mutants it detects. This interpretation describes the credit allocation and provides no convergence guarantee. Appendix A.4 discusses assertion execution and token-span mapping. We use GRPO group size $k = 6$, PPO clip 0.2, and no KL penalty. Appendix B.5 lists the training settings.

3 Evaluation

3.1 Evaluation Protocol

Populations and exposure. Section 2.1 analyzes all 500 SWE-bench Verified issues. We curate SFT and RL data from 230 of them and evaluate the trained policies on the remaining 270, of which 180 come from training repositories and 90 from repositories excluded from training. Held-out means excluded from SFT and RL training. All 270 appeared in the diagnostic, so this split measures held-out issues within a benchmark used for method development rather than an out-of-distribution evaluation. Transfer uses DeepSWE (113 tasks), SWE-PolyBench, SWE-bench Lite, and SWE-bench Pro (731 issues). The 30B study evaluates all four after SFT and all but Lite after RL, while the 7B and 14B runs also evaluate Lite (Appendix A.8).

Decoding and aggregation. The policies share the backbone, scaffold, tools, context limits, and attempt budget. Pass@1 uses the greedy first attempt, and eight-attempt evaluation adds seven samples at $T = 0.7$, so the samples are not identically distributed. This budget differs from the eight methodology-conditioned chains and the candidate grid of Section 2.1. The configuration specifies three seeds.

Candidate populations and metrics. Coverage and selection use each policy's own eight-candidate sets. Verifier discrimination instead uses one frozen bank of labeled repairs per held-out issue, shared across policies, combining natural incorrect repairs from several generators with correct alternatives, with labels withheld from the test-generating policy. Cross-error-source evaluation separates familiar operators, held-out operators, an excluded semantic-mutant generator, and natural incorrect repairs. Repair correctness is defined by the benchmark grader, so resolution and verifier labels measure agreement with its tests rather than semantic correctness. Pass@8 is oracle coverage, Selected@8 is the fraction whose selected candidate is correct, and selection efficiency is Selected@8/pass@8. Taking a correct repair as the positive class, precision is $TP/(TP + FP)$, the false-positive rate is $FP/(FP + TN)$, correct acceptance is $TP/(TP + FN)$, and balanced accuracy averages correct acceptance and incorrect rejection. DeepSWE verifier labels use fail-to-pass satisfaction. We report $\eta_k = U^{\text{loc}}_k/k$ as location diversity per attempt, and agent-steps count agent interactions rather than tokens or execution time. Appendix B.9 defines acceptance coverage, correct-alternative acceptance, and the zero-survivor audit.

3.2 Repair Accuracy

On the Verified split held out from training, SFT increases pass@1 from 31.9% to 35.2% and RL from SFT raises it to 43.0% (Table 5). Applying the same RL objective to the base policy yields 35.9% (Table 7), so the objective gains 4.0 points from the base policy and 7.8 from the supervised one.

3.3 Candidate Coverage and Selection

SFT increases oracle pass@8 from 46.7% to 51.1%, while RL raises it to 60.7%. Selected@8 increases from 33.5% to 38.5% after SFT and to 50.5% after RL. Selection efficiency rises from 0.72 to 0.75 and then to 0.83 (Table 3). The gains therefore involve both the availability of correct repairs and the fraction selected by the verifier. The coverage half traces to the exploratory behavior the corpus weighting teaches. Trained to spread attempts across locations and vary the edit, the policy reaches 1.98 distinct localizations per issue against 1.41 for the base model, and the share of samples contributing a new fault location rises from 0.18 to 0.25 at eight attempts, so more of the budget lands on ground no earlier sample covered.

Behavioral changes under supervision. Supervision raises pass@1 on every suite that resolves any task, by 3.3 points on Verified and 3.8, 8.4, and 13.4 on SWE-PolyBench, Lite, and Pro (Table 2). The trajectory measures move in the same direction on every suite, with more files read before the first edit, smaller patches, fewer test-file edits, and fewer steps to first submission. Reproduce-first is the one measure where base leads, 94.4% against 92.1% on Verified, while rising on all four transfer suites. This is the behavior the weighting targets. A policy that commits to one site keeps editing it, so a wrong first localization is unrecoverable, and Section 2.1 showed that breadth alone loses to blind retry. The weighting rewards moving on once execution rules a site out, so the added breadth lands where feedback directs it rather than across arbitrary sites. At matched corpus and budget, removing the exploration factor lowers distinct localizations from 1.98 to 1.74 and pass@1 from 35.2% to 34.8% on Verified, and from 4.13 to 3.70 and 24.2% to 23.2% on SWE-PolyBench (Table B.10). Staging the terms raises localization and resolution together, from 1.55 and 33.7% under uniform weighting to 1.90 and 34.8% with exploration, while an interface-compliance control reaches 1.45 and 32.6%, below uniform on both (Table 19). Which trajectory measure carries the effect is not separated.

Steps to first submission decrease from 62.5 to 56.0 (Table 2), and η_8 rises from 0.34 to 0.61 on DeepSWE. Location diversity does not measure candidate correctness, so it complements rather than replaces the coverage analysis.

3.4 Verifier Discrimination on Fixed Candidates

On one frozen candidate bank shared across policies, so that differences reflect the verifier and not an easier candidate population, the RL verifier rejects 87.4% of incorrect repairs against 73.0% for SFT, while correct-patch acceptance and balanced accuracy rise with it, so both sides of the decision improve rather than the verifier becoming uniformly stricter. Correctness-only RL captures roughly a third of that gain (Table 4). The per-assertion credit pushes the test toward assertions that each catch something the others miss, since a detected variant divides one unit among its detectors and a redundant assertion collects less than a new one. Two mutant sources keep that pressure from collapsing onto one distribution, since an assertion set tuned to operator shapes leaves model-generated variants alive. Training on operators alone loses pass@1, precision, and zero-survivor rate together (Table 9), and rejection degrades gradually off the training distribution (Table 16).

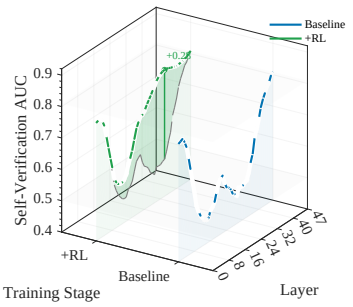

Figure 2: Layer-wise self-verification AUC across training stages. The post-RL representation has higher AUC for distinguishing correct and incorrect patches, with the largest differences in the mid-to-late layers (shaded).

The verification code itself changes. The median test carries no assertions before training and 11 after, length falls from 42 to 33 lines, and it covers 3 of 5 fault classes rather than none (Table 7). A test with no assertions cannot reject any variant.

Holding the base repair generator fixed, replacing its verifier with the trained verifier raises resolution from 31.9% to 36.5% (Table 15). The estimated interaction between changing the generator and changing the verifier is -0.1 percentage points, with a 95% confidence interval of $[-2.6, 2.4]$. This experiment detects a verifier contribution without establishing a nonzero interaction.

Table 2: Base and SFT results across five evaluation suites. Behavioral measures summarize the evaluated trajectories, diversity measures compare eight attempts per issue, and downward arrows mark metrics where smaller is preferred. Behavioral changes are descriptive rather than independent measures of correctness.

	SWE-bench Verified		DeepSWE		SWE-PolyBench		SWE-bench Lite		SWE-bench Pro	
Metric	Base	SFT	Base	SFT	Base	SFT	Base	SFT	Base	SFT
Resolution rate										
pass@1	31.9	**35.2**	0.0	0.0	20.4	**24.2**	28.3	**36.7**	2.3	**15.7**
pass@8	46.7	**51.1**	0.0	0.0	24.6	**30.2**	30.7	**39.7**	7.4	**19.6**
Exploration										
Files read before first edit	1.87	**2.19**	1.26	**2.12**	2.69	**3.91**	2.61	**3.89**	4.71	**6.81**
Read-to-edit ratio	3.44	**7.49**	9.14	**13.7**	6.21	**8.19**	7.24	**9.50**	5.56	**7.61**
Search commands / trajectory	28.9	**33.5**	48.3	**52.8**	25.9	**30.1**	28.2	**33.8**	14.5	**18.8**
Reproduction & testing										
Reproduce-first rate	**94.4**	92.1	28.7	**31.9**	23.3	**27.1**	15.7	**16.9**	7.1	**12.5**
Test-execution rate	92.1	92.1	24.4	**27.8**	86.2	**88.2**	96.3	**98.2**	31.1	**37.8**
Test-after-fix rate	91.9	**93.1**	23.5	**25.1**	84.5	**87.6**	91.3	**93.3**	11.1	**18.5**
Self-correction										
Self-correction rate	77.9	**81.2**	17.3	**21.8**	44.8	**45.7**	47.0	**53.1**	13.7	**20.1**
Edit discipline										
Patch size, lines changed ↓	4	**2**	4	**2**	4	**2**	4	**2**	71	**60**
Source files touched	1	1	1	1	1	1	1	1	1	1
Test-file-edit rate ↓	30.2	**18.4**	13.7	**8.80**	11.2	**7.4**	17.3	**12.1**	9.3	**5.5**
Efficiency										
Steps to first submission ↓	62.5	**56.0**	84.9	**76.5**	54.2	**46.5**	61.5	**55.5**	29.5	**21.5**
Format/context-overflow rate ↓	20.3	**10.4**	30.8	**18.6**	27.6	**16.3**	22.7	**10.8**	39.4	**25.7**
Solution diversity (across 8 samples)										
Distinct localizations	1.41	**1.98**	2.74	**4.86**	3.33	**4.13**	2.72	**3.81**	3.77	**4.51**
Distinct patch signatures	5.77	**6.10**	4.26	**5.11**	3.12	**3.89**	3.21	**5.62**	6.37	**7.54**

Format/context-overflow rate is the fraction of rollouts with at least one recoverable protocol or context event. An event can occur in a rollout that resolves the issue.

Policy	Oracle@8	Selected@8	Majority vote	Random pick	Sel. eff.
Base	46.7	33.5	30.0	26.0	0.72
SFT	51.1	38.5	35.0	30.2	0.75
+RL	60.7	50.5	44.0	36.0	0.83

Table 3: Coverage and selection on the Verified split held out from training using the existing eight-candidate sets. Oracle@8 is the fraction of issues with a correct candidate. Selected@8 is the fraction for which the verifier returns a correct candidate. Selection efficiency is their ratio. No additional repairs are generated for this analysis.

Figure 2 reports a layer-wise probe comparison. The post-RL curve has higher AUC, with its largest differences in the mid-to-late layers. This association does not establish a causal contribution to repair accuracy.

3.5 Reward Controls

The RL controls start from one SFT checkpoint and match training issues, update counts, and rollout budgets. Correctness-only GRPO reaches 39.6% pass@1 and 28.5% verifier precision. Adding scalar verification reward raises these values to 40.7% and 38.4%. Assertion credit raises them to 43.0% and 41.7% (Table 5b). Both the verification reward and assertion credit contribute to the reported gain.

Uniform assertion credit yields 38.0% precision and shuffling it at matched total mass yields 31.5%, so placement affects discrimination beyond the amount assigned. Removing mutation evaluation reduces pass@1 from 43.0% to 38.1%, the largest reduction among the component removals in Table 8. Allowing uncaught errors to count as detections reduces it to 39.6%. These removals measure contributions within the full configuration and need not have additive effects. Ungating the shaping terms costs 1.9 points, removing the repair loop costs 3.0 while leaving precision at 40.1%, and removing the exploration terms costs 1.5 (Table 8).

Group size raises both metrics, from 41.5% pass@1 and 39.0% precision at k=4 to 43.0% and 41.7% at the trained k=6 and 43.3% and 42.6% at k=16. Lowering the correctness weight raises precision to 43.2% while pass@1 falls to 38.9% (Table 10), so the two respond differently to the same reward change.

Verifier policy	Wrong rej. (TNR)	Correct acc. (TPR)	Bal. acc.	Accept. cov.	Sel. acc.	FPR ↓
Base	70.0	68.0	69.0	66.0	52.0	30.0
SFT	73.0	72.0	72.5	70.0	58.0	27.0
Correctness-only RL	78.0	74.0	76.0	73.0	63.0	22.0
Full (+RL)	87.4	80.0	83.7	78.0	72.0	12.6

Table 4: Verifier discrimination on a frozen bank of labeled repairs shared across policies. The bank contains natural incorrect repairs from multiple generators and correct alternatives. Labels are not exposed to the test-generating policy. Wrong rejection and correct acceptance measure the two sides of discrimination. Metric denominators are defined in Section 3.1 and Appendix B.9.

Table 5: Training exposure and reward controls. Panel (a) separates the 230 training issues, 180 test issues from training repositories, 90 test issues from repositories excluded from training, and SWE-bench Pro. Panel (b) compares objectives from one SFT checkpoint at matched training issues and update and rollout budgets. Δ is the pass@1 difference from the full objective in percentage points.

(a) Resolution by training exposure

		pass@1			pass@8		
Evaluation subset	N	**Base**	**SFT**	**+RL**	**Base**	**SFT**	**+RL**
Train set, Verified (reference)	230	33.9	37.4	45.7	49.1	53.5	63.5
Test issues, training repositories	180	33.3	36.7	44.4	48.9	53.3	62.8
Test issues, nontraining repositories	90	28.9	32.2	40.0	42.2	46.7	56.7
Frozen external (SWE-bench Pro)	731	2.3	15.7	20.2	7.4	19.6	22.8
All 270 held-out test (Verified)		31.9	35.2	43.0	46.7	51.1	60.7

(b) Reinforcement-learning controls (from one shared SFT checkpoint)

Configuration	pass@1	pass@8	Verif. prec.	Correct-alt acc.	Δ vs full
SFT initialization (no RL)	35.2	51.1	26.8	60.0	−7.8
Correctness-only GRPO	39.6	56.7	28.5	64.0	−3.4
Correctness + scalar VERIF (β=0)	40.7	58.5	38.4	72.0	−2.3
Full objective (+ token credit)	43.0	60.7	41.7	78.0	–
Matched uniform assertion credit	41.5	59.6	38.0	73.0	−1.5
Shuffled-credit control (matched mass)	40.0	57.8	31.5	64.0	−3.0
Prior mutation-RL control	40.7	58.1	35.5	60.0	−2.3

3.6 TRANSFER AND LIMITATIONS

RL improves pass@1 over SFT by 8.3 percentage points on train-exposed issues, 7.7 on test issues from training repositories, and 7.8 on repositories excluded from training, reaching 45.7%, 44.4%, and 40.0% respectively (Table 5a). The gain is near-constant across the exposure gradient while the absolute level falls, the pattern expected if the objective transfers rather than recalling training fixes. On SWE-PolyBench, pass@1 increases from 24.2% to 31.8%. On SWE-bench Pro, it increases from 15.7% to 20.2% (Table 6). These evaluations support transfer beyond the training issues. They do not exclude pretraining exposure to public repositories.

DeepSWE full resolution remains zero at this model scale, but its internal measures improve. SFT raises the tasks with at least one satisfied fail-to-pass test from 17 to 21 of 113 and the mean best-of-eight satisfied fraction from 4.0% to 4.8% (Table 13), which is partial test satisfaction rather than completed repair. Verifier precision uses a different positive criterion here (Section 3.1).

The signed assertion-credit ablation raises correct-alternative acceptance from 78.0% to 84.0%, cross-generator discrimination from 74.0% to 78.0%, and pass@1 from 43.0% to 43.3% (Table 18). This result supports adding an assertion-level penalty for false rejection. The main results use unsigned credit, so the signed variant's gains apply to this ablation.

Verification remains sensitive to the source of incorrect repairs. The RL verifier rejects 91.0% of familiar-operator mutants, 84.0% of held-out operators, 79.0% of variants from an excluded generator, and 74.0% of natural incorrect repairs from independent generators. The latter exceeds SFT by 12 percentage points, while correct-alternative acceptance remains 78.0% (Table 16). Complete detection of a finite mutant set does not establish correctness outside that set.

The 7B and 14B experiments gain from both stages. Verified pass@8 rises 21.9 → 30.4 → 33.3% at 7B and 27.0 → 36.7 → 42.6% at 14B, and SWE-PolyBench rises 8.0 → 17.8% and 19.8 → 37.4% (Table 14). Published coder results in Appendix A.5 provide context but use different scaffolds,

	DeepSWE				SWE-PolyBench				SWE-bench Pro			
Metric	Base	**+RL**	SFT	**+RL**	Base	**+RL**	SFT	**+RL**	Base	**+RL**	SFT	**+RL**
pass@1	0.0	0.0	0.0	0.0	20.4	23.8	24.2	31.8	2.3	15.5	15.7	20.2
pass@8	0.0	0.0	0.0	0.0	24.6	30.2	30.2	41.6	7.4	19.4	19.6	22.8
Verifier precision	21.1	33.7	23.1	38.6	13.6	22.8	17.5	26.8	15.5	24.5	21.0	24.7
False-positive rate ↓	23.0	14.6	21.1	11.1	19.0	14.6	18.1	11.3	21.7	14.5	20.9	12.7
Fixpoint (zero-survivor) rate	3.0	3.9	5.11	8.3	0.4	2.1	3.87	7.31	1.90	4.51	2.81	4.19

Table 6: Transfer evaluation from base and SFT initializations. Each initialization is paired with its RL-trained policy. DeepSWE verifier precision uses a fail-to-pass criterion rather than full task resolution.

subsets, or selection protocols. They do not constitute matched comparisons. Agent-step savings also exclude differences in token lengths and test-execution costs.

4 Related Work

Repository search and feedback. SWE-agent and OpenHands provide interfaces for repository inspection, editing, and execution (Yang et al., 2024; Wang et al., 2024). Agentless organizes repair around localization (Xia et al., 2024), and inference-time scaling studies examine additional attempts (Brown et al., 2024; Snell et al., 2024). Our diagnostic holds backbone and scaffold fixed and compares how location search and verification affect repair. Generated tests also support candidate selection (Chen et al., 2023), and we measure candidate coverage and selected accuracy as distinct outcomes.

Training repair and test generation. CodeRL trains code generation from execution feedback (Le et al., 2022), SWE-Gym studies training repository agents and verifiers (Pan et al., 2024), and CURE co-trains a coder and unit tester with reinforcement learning (Wang et al., 2025b). Our policy generates both the patch and its test, and the reward combines patch correctness with detection of oracle-rejected variants. SWT-Bench evaluates tests for repository bug fixes (Mündler et al., 2024), a related setting for assessing tests beyond whether they execute.

Mutation rewards and credit assignment. MIST-RL rewards each test for detecting mutants earlier tests missed and penalizes redundancy (Zhu et al., 2026). Ockhamareto is the closest comparator for our token-level credit, mapping mutation contributions to test-block tokens under a Pareto gate on suite effectiveness and size (Huang et al., 2026). Our credit divides each detected mutant's weight among its detecting assertions and applies a positive-advantage bonus gated by repair correctness or complete detection, with the trajectory reward prioritizing repair correctness. These differences concern the reward and credit rule rather than introducing mutation testing or token-level attribution.

Test and patch revision. CoHarden generates a reproduction before editing and uses mutant-based feedback to revise tests and fixes (Tan et al., 2026), the closest comparison to our revision loop. We also train the policy from those signals and evaluate the resulting verifier on frozen candidates, correct alternatives, and incorrect repairs from independent generators.

5 Conclusion

We hypothesized that reliable repair rests on three learnable behaviors, locating the fault, varying the edit, and verifying which candidate to return. The first two determine candidate coverage and the third selection quality, neither compensating for the other, and the swap estimates their contributions as additive. Weighted supervision and mutation-based reinforcement learning improve both, with gains on SWE-bench Pro and SWE-PolyBench and, on DeepSWE, partial credit without full resolution. The constraint is general, since a proposer scoring its own work cannot be wrong, so negatives must be manufactured rather than collected.

Where the verifier is weakest indicates what to improve. Rejection falls from 91.0% on familiar operators to 74.0% on repairs from independent generators, so a wider negative distribution during training is the first direction. The second is the signed credit rule, which charges an assertion for rejecting a correct alternative and raises correct-alternative acceptance from 78.0% to 84.0% in our ablation, though not yet under the full evaluation. Folding the verifier into the search loop, so that it steers exploration rather than filtering afterwards, would reach the coverage limit selection cannot.

AI USE STATEMENT

Generative AI was used only to improve the grammar, clarity, and concision of author-written text, and coding assistants were used for routine tasks such as plotting, table generation, and code refactoring. AI was not used to develop the research ideas, design the methodology or experiments, or generate or interpret experimental results. The authors reviewed and verified all AI-assisted content and take full responsibility for the paper.

ETHICS STATEMENT

This work uses publicly available models and benchmarks and involves no human subjects, personally identifiable information, or new data collection.

REPRODUCIBILITY STATEMENT

Section 2 defines the method, and Section 3.1 specifies the evaluation protocol. Appendices A.4 and B.5 report implementation details and training hyperparameters. Search rules, prompts, and experiment configurations appear in Appendices A.1, B.13, and B.15. Appendices A.3 and B report ablations.

REFERENCES

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Investigating test overfitting on SWE-bench. *arXiv preprint arXiv:2511.16858*, 2025.

Pepe Alonso, Sergio Yovine, and Victor A. Braberman. TDAD: Test-driven agentic development – reducing code regressions in AI coding agents via graph-based impact analysis. *arXiv preprint arXiv:2603.17973*, 2026.

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

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code generation with generated tests. In *International Conference on Learning Representations (ICLR)*, 2023.

Zhilong Chen, Chengzong Zhao, Boyuan Chen, Dayi Lin, Yihao Chen, Arthur Leung, Gopi Krishnan Rajbahadur, Gustavo Oliva, Haoxiang Zhang, Aaditya Bhatia, Chong Chun Yong, and Ahmed E. Hassan. RepoForge: Training a sota fast-thinking SWE agent with an end-to-end data curation pipeline synergizing SFT and RL at scale. *arXiv preprint arXiv:2508.01550*, 2025.

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

Dong Huang, Mark Harman, Jie M. Zhang, Zhijiang Guo, Mingzhe Du, and See Kiong Ng. Ockhamareto: Pareto-gated segment-level credit assignment for concise unit-test generation with reinforcement learning. *arXiv preprint arXiv:2608.24473*, 2026.

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. In *International Conference on Learning Representations (ICLR)*, 2024.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. CodeRL: Mastering code generation through pretrained models and deep reinforcement learning. *arXiv preprint arXiv:2207.01780*, 2022.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. Lingma SWE-GPT: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*, 2024.

Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. SoRFT: Issue resolving with subtask-oriented reinforced fine-tuning. *arXiv preprint arXiv:2502.20127*, 2025.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and validating real-world bug-fixes with code agents. *arXiv preprint arXiv:2406.12952*, 2024.

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.

Mohit Raghavendra, Anisha Gunjal, Bing Liu, and Yunzhong He. Agentic rubrics as contextual verifiers for SWE agents. *arXiv preprint arXiv:2601.04171*, 2026.

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

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

Yuhao Tan, Zhibang Yang, Fangkai Yang, Yuan Yao, Yu Kang, Lu Wang, Pu Zhao, Xin Zhang, Xiaoxing Ma, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. Beyond fail-to-pass: Iterative hardening of co-generated bug reproduction tests and fixes. *arXiv preprint arXiv:2607.19843*, 2026.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. SWE-Dev: Building software engineering agents with training and inference scaling. *arXiv preprint arXiv:2506.07636*, 2025a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, 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. *arXiv preprint arXiv:2506.03136*, 2025b.

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

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 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.

Sicheng Zhu, Jiajun Wang, Jiawei Ai, and Xin Li. MIST-RL: Mutation-based incremental suite testing via reinforcement learning. *arXiv preprint arXiv:2603.01409*, 2026.

A APPENDIX

This appendix provides reward ablations, implementation details, comparator studies, and a worked reward example. It also reports transfer and model-scale results and documents the prompts, search logic, and experiment configurations.

A.1 TEST-TIME SEARCH PROCEDURES

Self-verification. The agent submits a patch, then runs up to three audit passes that read the repository, run tests, and revise, with no grader feedback (prompt in Appendix B.12). It resolves $184/500$ (36.8%) at $189{,}371$ agent-steps ($1{,}029$ per solve), yet 78 to 85% of audit passes change no code and across four attempts it reaches only 1.10 distinct locations and 1.31 distinct edits. It neither escapes its wrong first guess nor stops paying to re-examine it, taking both the lowest accuracy and the highest cost.

Single-axis diagnosis. One attempt per issue, no retry and no diversity mechanism, fixing the single-sample reference the other procedures answer to. It resolves $189/500$ (37.8%) at $41{,}332$ agent-steps, the cheapest run in the study at 219 steps per solve.

Methodology diversity. Running each issue under eight problem-solving preambles (Appendix B.13) solves overlapping but distinct subsets, with combined oracle coverage $305/500$ (61.0%), above any single preamble. This label-using union is not a deployable rule, yet it marks the headroom. The agent already produces a correct fix for far more issues than one strategy surfaces, so selection limits performance more than generation.

Directed exploration. Up-front localization pins one attempt to each of 2.46 ranked sites yet resolves only $210/500$ (42.0%), below blind retries ($213/500$), so broad coverage alone does not help. Spending attempts on the right sites does, and that needs execution feedback. Banning a confirmed-wrong edit signature and stopping on a strong positive resolves $217/500$ (43.4%) at $131{,}417$ steps, reaching 1.84 locations and 2.79 edits with early stops on 65 issues.

Reactive location diversity. Banning the whole location, its files and functions, resolves $225/500$ (45.0%) at 585 steps per solve, under the ban and early-stop rules of Appendix B.14. Location diversity pays only when earned reactively, and reactive location-banning (225) exceeds reactive edit-banning (217), blind retries (213), and proactive site-planning (210), in that order.

Differential-verification search. The combined procedure builds up to eight candidate cells, each pairing one localization with one repair approach. It scores every candidate by executing the agent's own reproduction twice, once on the tree carrying the patch and once on that same tree with the patch reverted. A failure on the reverted tree together with a pass on the patched tree shows the patch changed the behavior the reproduction measures, which is weaker than showing the patch correct elsewhere. The score itself rises with how much independent evidence stands behind that verdict, and it decides whether search continues. A confirmed differential scores 1.2 when two further checks agree, the repository's existing tests still pass and the reproduction touches the changed lines, and 1.0 when either is missing, which reads a regression failure as pre-existing rather than caused by the patch. A patch whose reproduction passes while the reverted run returns no verdict scores 0.3, since the pass is real but nothing corroborates it. No usable signal or a broken harness scores 0. A vacuous differential, where the reproduction passes on both trees, means it never exercised the fault, so the candidate takes a hard 0 and the location is pruned. Any score of at least 1.0 is strong enough to stop the search. Every decision uses the agent's own reproduction and the repository tests, never the hidden grader.

Differential-verification search resolves $264/500$ (52.8%) against $234/500$ (46.8%) for the eight-sample control, at $132{,}697$ agent-steps, 48.1% of the control's $275{,}943$, and 3.12 attempts per issue. That is 503 steps per solve against $1{,}179$. Early stopping fires on about 102 issues and pruning drops 126 localizations across 73 issues, so the gains come from spending compute where feedback says it will pay. A component ablation (Appendix B.11) isolates each part. Dropping the two-tree differential does the most damage ($52.8 \rightarrow 47.4\%$, near the eight-sample baseline), while early stopping costs compute alone (steps/solve $503 \rightarrow 728$ at unchanged resolution).

A.2 Reinforcement learning results by starting point

The reinforcement objective is applied twice, once to the base policy and once to the supervised policy, so that the contribution of the initialization can be read off directly. Both starting points improve on every metric in Table 7, which shows the objective does not depend on supervision to function. The size of the gain does depend on it. Pass@1 rises by 4.0 points from the base policy and by 7.8 from the supervised one, and verifier precision by 9.6 against 14.9 points, so the same reward is worth roughly twice as much once the policy already searches widely. The reason is visible in how GRPO forms its advantage, since rollouts that repeat the same edit share the same outcome and leave the group-relative advantage flat, while a wider candidate set puts correct and incorrect patches in the same group and gives the verifier term something to separate. Table 4 scores the resulting verifiers on one shared bank of labeled repairs, which removes the possibility that the improvement reflects an easier candidate population rather than a better verifier.

A.3 Reinforcement Learning Ablations

Each ablation removes or varies one component of the objective or training configuration described in Section 2.3. Unless otherwise noted, runs start from the supervised policy and are evaluated on SWE-bench Verified. We report the issue resolution rate (pass@1) and verifier precision against gold correctness labels. Table 9 varies the adversary that generates the semantic mutants, and Table 10 sweeps the reward weights and the group size.

A.4 RL objective implementation details and boundary cases

Two implementation notes accompany the per-assertion credit of Eq. 4. Mapping $s_t = c_i$ onto every token of assertion i scales its total bonus by the span length L_i, so a per-token form $s_t = c_i/L_i$ removes this length effect. And ordinary execution halts a test at its first failing assertion, so reconstructing the full matrix K needs isolated, independently executable test nodes. The Shapley reading of $\tilde{c}_i$ carries no convergence claim.

We define the scored quantities only when their inputs exist and fix the empty cases explicitly. With no valid negatives ($|S| = 0$) the kill fraction is undefined and the fixpoint bonus does not fire, so we record insufficient negatives and give no verifier credit. With no assertions or no detected mutants we zero the local credit c_i and log the event. Infrastructure or import errors count as invalid execution and earn no behavioral-kill credit, so a broken harness earns no reward. The honesty term penalizes a test that fails on a correct repair, so an always-failing test cannot look discriminative. We report and control the valid mutant count and duplicate rate, since zero-survivor rates are otherwise incomparable. The gate G is an OR, so the local bonus can fire on a resolved or complete rollout, correctness still enters only through Correct, and we never read a full kill as correctness assurance.

A.5 Comparator Studies

Table 11 lists published results with their backbones, scaffolds, and evaluation settings. The rows differ in evaluated subsets and inference protocols. Oracle pass@8 measures whether a correct candidate exists, whereas best-of-16 reports a selected candidate under a different budget. These values provide context and are not a matched ranking.

A.6 A worked example of the mutation-verifier reward

The reward of Section 2.3 is clearest on a single instance. The instance shows how the reward spreads credit across fault classes and how it rewards a group of parallel attempts for covering different edit locations.

The loop on one instance. The example composites two real rollouts on SWE-bench Verified, `matplotlib__matplotlib-21568` and `django__django-16950`. The task, reproduction, and assertion design (stages 1, 2, 4), the mutant set and kills (stages 3, 5), and the group statistics (stage 6) all come from those runs.

	from Base		from SFT	
Metric	pre-RL	**+RL**	pre-RL	**+RL**
Resolution rate (SWE-bench Verified)				
pass@1	31.9	35.9	35.2	43.0
pass@8	46.7	53.7	51.1	60.7
Verifier quality (vs. gold, held-out)				
Verifier precision	23.5	33.1	26.8	41.7
False-positive rate ↓	30.0	19.0	27.0	12.6
Fixpoint (zero-survivor) rate	1.5	6.3	2.8	15.6
Verification-code discipline				
Assertions / test (median)	0	**5**	0	**11**
Test length, lines ↓	42	**39**	42	**33**
Fault classes covered / test	0/5	**3/5**	0/5	**3/5**

Table 7: Base and SFT policies before and after RL on the 270 Verified issues held out from training. Pass@1 measures greedy repair accuracy and pass@8 measures oracle coverage. Verifier metrics use oracle correctness labels. Assertion counts, test length, and fault-class coverage describe the generated tests.

Configuration	pass@1	Verifier prec.	Δ vs. full
Full objective	43.0	41.7	–
$\beta = 0$ (flat scalar, no step-credit)	40.7	38.4	−2.3
no mutation gauntlet (self-consistency only)	38.1	29.5	−4.9
credit crash-kills (no assertion gate)	39.6	33.0	−3.4
ungated E, P shaping terms	41.1	37.9	−1.9
no in-episode repair loop	40.0	40.1	−3.0
no exploration terms	41.5	40.9	−1.5

Table 8: Ablations of the reinforcement objective on SWE-bench Verified, each removing one component the methodology argues is load-bearing. Δ is the change in resolution against the full objective.

1. **The task.** An over-restrictive rule rejects an input it should accept, treating a legal separator as illegal. A correct fix broadens the rule to admit the character while still rejecting malformed inputs.
2. **Reproduce before editing.** The agent writes a minimal script that triggers the reported failure, localizes the defect to the validation routine, and applies a small targeted change. Reproducing first ensures the later test observes the real defect instead of a patch artifact.
3. **Adversarial mutation.** From the model's own patch, a fix in one file with eleven changed lines all exercised by the test, the adversary produces eight behavior-changing variants of the fixed region (Table 12). Each is a plausible but subtly wrong rewrite meant to defeat a shallow test, spread across five fault classes so the test must probe several input regimes.
4. **The discriminative test.** A shallow check that only confirms the reported input no longer errors catches at most one variant and earns near-zero reward. The objective instead drives the policy to assert concrete required outputs on inputs chosen to separate each fault class, so every wrong variant fails an assertion.
5. **The mutation gauntlet.** Each variant runs against the test and counts as eliminated only when an assertion fails, so a crash gives no credit. The loop repeats until no variant survives or the budget is spent. Here the test reaches the fixpoint across all five fault classes (Table 12), so assertion credit spreads across classes and the fixpoint gates the correctness reward.
6. **Edit-location diversity.** The four parallel attempts stay spread across locations on `django__django-16950`. Three edited the consensus forms module (`forms/models.py`), including the selected verified attempt, and the fourth edited an upstream model-field module (`fields/__init__.py`). The edit term credits both a plausible or validated location and an edit distinct from the group, so the objective rewards covering the space and staying diverse.

A.7 DeepSWE partial-credit breakdown

DeepSWE's binary grader resolves zero tasks for either policy, so we score partial credit against its fail-to-pass tests.

Semantic-mutant adversary	pass@1	Verifier prec.	Fixpoint rate
Weak (same-size, untuned)	41.1	37.5	11.8
Same-family (policy's sibling)	41.9	39.8	13.9
Strong (frozen, ours)	43.0	41.7	15.6
Deterministic mutants only	40.4	35.0	9.5

Table 9: Robustness to the adversary used to generate semantic mutants. All values are percentages. Every semantic-mutant configuration beats the deterministic-only baseline on all three metrics, so the benefit extends beyond the strong adversary, which leads by 2.6, 6.7, and 6.1 points over deterministic mutants alone.

	Reward weights $w_c : w_v$		GRPO group size k		
Metric	**High w_c**	**Low w_c**	k=4	k=6	k=16
pass@1	43.0	38.9	41.5	43.0	43.3
Verifier prec.	41.7	43.2	39.0	41.7	42.6

Table 10: Sensitivity to the relative correctness and verifier reward weights ($w_c : w_v$) and the GRPO group size k, the number of rollouts drawn per training instance. All values are percentages. The trained setting of Section 2.3 is w_c:w_v = 3.0:1.0 at k=6, tinted in both blocks because it is the shared reference point. Lower correctness weighting increases verifier precision by 1.5 percentage points but reduces pass@1 by 4.1. Group size pays off early. Moving from k=4 to k=6 adds 1.5 pass@1 and 2.7 precision, while going on to k=16 adds a further 0.3 and 0.9 for more than twice the rollouts per instance, which is why we train at k=6.

A.8 Model-scale trend

We repeat the full pipeline on 7B and 14B backbones (Table 14).

B Rigor and reproducibility

This section provides training settings, metric definitions, and additional controls for the main results. It examines generator-verifier swaps, error-source transfer, mutation audits, and ablations of credit assignment, trajectory weighting, and differential verification.

Verified evaluations use the 270 issues held out from SFT and RL training unless a different population is stated. The test-time diagnostic and differential-verification ablation use all 500 issues, including the 270 evaluation issues. Section 3.1 defines the populations, decoding, and candidate-based metrics.

B.1 Protocol for the main-text rigor results

We record the protocol behind Table 5 here.

Exposure split. Table 5a separates training issues, test issues from training repositories, and test issues from repositories excluded from training. Section 3.1 defines training exposure.

Reward controls. The RL configurations in Table 5b use one SFT checkpoint and matched training issues and update and rollout budgets. The shuffled-credit control preserves total credit mass and permutes its assignment across assertion tokens.

B.2 Generator and verifier swap

This experiment separates the two things training changed. Repair generation and candidate selection are held in fixed roles, and each is swapped independently between the base and trained policies, so the four cells of Table 15 span every combination at one attempt budget. Holding the generator at base and replacing only the verifier raises end-to-end resolution from 31.9% to 36.5% and selection accuracy from 52.0% to 66.0%, which shows the verifier changes which patch is returned rather than which patches exist. Holding the verifier at base and replacing only the generator reaches 38.5%, so both roles contribute on their own. The interaction term is the diagonal contrast, $(43.0 - 38.5) - (36.5 - 31.9) = -0.1$ points with a 95% interval of $[-2.6, 2.4]$, so the two contributions add rather than multiply and neither is a precondition for the other paying off.

System	Backbone	Scaffold	Setting	Res. (%)
7B backbones				
SWE-Gym (Pan et al., 2024)	Qwen2.5-Coder-7B	OpenHands	Verified pass@1	10.6
SWE-Gym (Pan et al., 2024)	Qwen2.5-Coder-7B	OpenHands	Lite pass@1	10.0
SWE-agent-LM-7B (Yang et al., 2025)	Qwen2.5-Coder-7B	SWE-agent	Verified pass@1	15.2
SWE-agent-LM-7B (Yang et al., 2025)	Qwen2.5-Coder-7B	SWE-agent	Lite pass@1	11.7
Lingma SWE-GPT (Ma et al., 2024)	Qwen2.5-Coder-7B	SWESynInfer	Verified pass@1	18.2
Lingma SWE-GPT (Ma et al., 2024)	Qwen2.5-Coder-7B	SWESynInfer	Lite pass@1	12.0
R2E-Gym (Jain et al., 2025)	Qwen2.5-Coder-7B	R2E-Gym	Verified pass@1	19.0
R2E-Gym (Jain et al., 2025)	Qwen2.5-Coder-7B	R2E-Gym	Lite pass@1	11.0
SoRFT (Ma et al., 2025)	Qwen2.5-Coder-7B	Agentless	Verified pass@1	21.4
SoRFT (Ma et al., 2025)	Qwen2.5-Coder-7B	Agentless	Lite pass@1	14.0
SWE-Dev (Wang et al., 2025a)	Qwen2.5-Coder-7B	OpenHands	Verified pass@1	23.4
Ours (+RL)	Qwen2.5-Coder-7B	mini-SWE-agent	Verified pass@1	25.8
Ours (+RL)	Qwen2.5-Coder-7B	mini-SWE-agent	Lite pass@1	15.2
14B backbones				
RepoForge (base) (Chen et al., 2025)	Qwen2.5-Coder-14B	OpenHands	Verified pass@1	4.8
SWE-Gym (Pan et al., 2024)	Qwen2.5-Coder-14B	OpenHands	Verified pass@1	16.4
SWE-Gym (Pan et al., 2024)	Qwen2.5-Coder-14B	OpenHands	Lite pass@1	12.7
R2E-Gym (Jain et al., 2025)	Qwen2.5-Coder-14B	R2E-Gym	Verified pass@1	26.8
R2E-Gym (Jain et al., 2025)	Qwen2.5-Coder-14B	R2E-Gym	Lite pass@1	20.7
Ours (+RL)	Qwen2.5-Coder-14B	mini-SWE-agent	Verified pass@1	31.2
Ours (+RL)	Qwen2.5-Coder-14B	mini-SWE-agent	Lite pass@1	24.1
30B backbone (Qwen3-Coder-30B-A3B)				
TDAD (Alonso et al., 2026)	Qwen3-Coder-30B-A3B	quantized	Verified pass@1 (100)	31.0
Agentic Rubrics (Raghavendra et al., 2026)	Qwen3-Coder-30B-A3B	SWE-Agent	Verified best-of-16	54.2
Base	Qwen3-Coder-30B-A3B	mini-SWE-agent	Verified pass@1	31.9
Base	Qwen3-Coder-30B-A3B	mini-SWE-agent	Verified pass@8	46.7
Supervised (SFT)	Qwen3-Coder-30B-A3B	mini-SWE-agent	Verified pass@1	35.2
Supervised (SFT)	Qwen3-Coder-30B-A3B	mini-SWE-agent	Verified pass@8	51.1
Ours (+RL)	Qwen3-Coder-30B-A3B	mini-SWE-agent	Verified pass@1	43.0
Ours (+RL)	Qwen3-Coder-30B-A3B	mini-SWE-agent	Verified pass@8	60.7

Table 11: Published coding-agent results and our results at 7B, 14B, and 30B. Each row gives the backbone, scaffold, and evaluation setting. Dataset subsets, sampling budgets, and selection procedures differ across systems, so the reported values do not establish a matched comparison.

Table 12: The eight adversarial variants generated for a real fixed patch (`matplotlib__matplotlib-21568`), by changed line and fault class, with the measured elimination outcome. Every variant is a distinct, behaviour-changing rewrite of the patched region. The harness records each variant's outcome and fault class but not which individual assertion eliminated it, so we report the per-variant verdict.

Variant	Fault class	Changed line	Outcome
V1	logic	595	eliminated (assertion)
V2	value	603	eliminated (assertion)
V3	control	605	eliminated (assertion)
V4	boundary	603	eliminated (assertion)
V5	effect	607	eliminated (assertion)
V6	boundary	597	eliminated (assertion)
V7	value	607	eliminated (assertion)
V8	control	600	eliminated (assertion)
Survivors 0/8 at fixpoint, from 8 assertion-eliminations and 0 crash-eliminations.			

B.3 Cross-error-source transfer

A verifier trained against one family of incorrect programs may only have learned that family. Table 16 moves the error source progressively away from training while holding the evaluation bank and the issues fixed, so any decline isolates distribution shift in the negatives rather than a change in difficulty. Rejection falls monotonically for every policy, and for the RL verifier it runs 91.0% on familiar operators, 84.0% on held-out operators, 79.0% on variants from an excluded generator, and 74.0% on natural incorrect repairs written by independent systems. The decline is gradual rather than a collapse, and the margin over SFT stays near constant at 11 to 12 points across all four sources, which indicates the objective taught discrimination that outlives its training operators. The last row is the one that matters for deployment, since those repairs were produced the way a real candidate would

Metric	Base	SFT model
Tasks evaluated	113	113
Fully resolved (all fail-to-pass, no regression)	0	0
Tasks with ≥ 1 fail-to-pass test satisfied	17	21
Tasks clearing $> 50\%$ of fail-to-pass tests	4	5
Mean fail-to-pass pass fraction (best of 8)	4.0%	4.8%
Distribution of best-of-8 fail-to-pass pass fraction (tasks)		
0%	96	92
$(0, 25]\%$	10	12
$(25, 50]\%$	3	4
$(50, 75]\%$	4	5
$(75, 100)\%$	0	0
Strongest tasks: fail-to-pass tests satisfied (best of 8)		
`httpx-deterministic-cookie-store`	78/115	86/115
`happy-dom-deterministic-intersectionobserver`	8/14	10/14
`optique-conditional-option-dependencies`	20/36	20/36
`testem-per-launcher-reports`	30/65	36/65
`true-myth-iterable-collection-combinators`	**33/96**	31/96
`geo-shapeindex-serialization`	7/24	7/24
`adaptix-name-mapping-aliases`	8/44	11/44
`mashumaro-flattened-dataclass-fields`	11/66	12/66

Table 13: DeepSWE evaluation at pass@8 over all 113 out-of-distribution tasks, comparing the base policy with our supervised fine-tuned model. The strict binary criterion resolves no task for either model, but the fine-tuned model makes greater partial progress. Dark yellow marks the fine-tuned model's wins, and the one base win is in bold.

	7B			14B		
Benchmark (pass@8, %)	Base	SFT	+RL	Base	SFT	+RL
SWE-bench Lite	16.0	22.0	24.7	23.0	30.3	34.7
SWE-PolyBench	8.0	14.4	17.8	19.8	28.8	37.4
SWE-bench Verified	21.9	30.4	33.3	27.0	36.7	42.6

Table 14: Effect of model scale. Pass@8 resolution (%) for the base, supervised (SFT), and reinforcement (+RL) policies at 7B and 14B on three SWE-bench-family benchmarks. Both the Base→SFT gain and the further SFT→+RL gain hold at both scales and do not diminish with capacity, so the taught search and verification behavior is complementary to raw model scale.

be, and correct-alternative acceptance holds at 78.0% there, so the verifier is not buying rejection by refusing everything.

B.4 Trajectory Statistics

Table 2 reports the behavioral inventory for base and SFT policies. The reproduce-first rate decreases from 94.4% to 92.1% on Verified and increases on the four transfer suites. The other behavioral changes describe the generated trajectories rather than establishing separate causal effects on resolution.

B.5 Reproducibility

We report the training hyperparameters used in the experiments. The backbone is Qwen3-Coder-30B-A3B-Instruct and the mutation labels come from a frozen Claude Opus-4.8 distinct from the policy. Context is 40k prompt and 65k response. We optimize with AdamW at learning rate 1×10^{-6} and weight decay 0, train batch 4, mini-batch 4, GRPO group k=6, PPO clip 0.2, and no KL penalty. The reward coefficients are w_c=3.0, w_v=1.0, w_e=0.4, w_p=0.2 with fixpoint bonus λ_{fp}=0.5. The gauntlet draws 8 mutants per candidate with a retry cap of 3, and the training configuration specifies 3 seeds.

Repair generator	Verifier	Sel. acc.	End-to-end resolve	Note
Base	Base	52.0	31.9	starting system
Base	Trained	66.0	36.5	verifier effect, fixed generator
Trained	Base	58.0	38.5	generation effect, fixed verifier
Trained	**Trained**	72.0	43.0	combined system
Interaction Δ_{int} (95% CI)		0.0 $[-2.5, 2.5]$	-0.1 $[-2.6, 2.4]$	CI includes 0

Table 15: Generator-verifier swap, same role interface and budget in every cell. The Base/Base and Trained/-Trained resolve cells are the measured held-out Verified pass@1 (31.9, 43.0). Holding the generator fixed and swapping in the trained verifier alone lifts resolution $31.9 \rightarrow 36.5$ ($+4.6$) and selection accuracy $52.0 \rightarrow 66.0$, so verification changes repair decisions independently of generation. The interaction term is in the last row.

Error source (held-out issues)	Base rej.	SFT rej.	+RL rej.	+RL correct-alt acc.
Familiar operators	72.0	80.0	91.0	80.0
Held-out fault operators / classes	64.0	73.0	84.0	78.0
Semantic mutants, excluded generator	58.0	68.0	79.0	76.0
Natural wrong repairs, independent gens	51.0	62.0	74.0	78.0

Table 16: Rejection rate by error source on a bank frozen across checkpoints, with correct-alternative acceptance in the last column. The familiar-operator +RL rejection (91.0) tracks the measured wrong-rejection margin (TNR 87.4, Table 4). Rejection falls as the source moves off the training distribution ($91.0 \rightarrow 74.0$), staying well above SFT everywhere, while correct-alt acceptance holds between 76 and 80. The decisive row is natural wrong repairs, where a $+12$ gain over SFT there indicates general verification rather than specialization to the mutation distribution.

B.6 Zero-survivor audit

Train-time completion is not the fixpoint rate of Table 7. Completion counts training instances, where the policy wrote the test against the instance in front of it. The fixpoint rate counts held-out instances. Held-out detection measures discrimination on issues excluded from training, including issues from training repositories and repositories excluded from training.

B.7 Assertion-credit ablation

The unsigned kill credit of Eq. 4 pays an assertion only for detecting a wrong variant. The signed two-sided rule also charges it for rejecting a gold-labeled correct alternative (Table 18).

The signed variant is evaluated in this ablation. All other results use unsigned assertion credit. The improvements in this table do not establish the signed variant's performance on the transfer benchmarks.

B.8 Weighted supervised fine-tuning ablation

Table 19 asks whether weighting the corpus matters or whether any fine-tuning on the same trajectories would do. All variants see the same data under the same token budget, so only the weight assigned to each example changes. Resolution and localization diversity rise together as terms are added, from 33.7% and 1.55 under uniform weights to 35.2% and 1.98 with the full weighting, and the exploration term is where the diversity gain concentrates. The interface-compliance control is the informative row. It repairs formatting and tool-call handling without exposing any task solution, and it reaches 32.6% with 1.45 distinct localizations, below uniform weighting on both, so the gain is not the model merely learning to survive the scaffold. Because the corpus is fixed across rows, these differences are attributable to the weighting rule rather than to data selection.

B.9 Operational metric definitions

Section 3.1 defines repair accuracy, candidate coverage, selection efficiency, and patch-level verifier metrics. The generated eight-candidate sets and frozen verifier banks are different populations. Acceptance coverage is the fraction of issues with at least one accepted candidate. Selection accuracy is the fraction whose selected candidate is correct. Correct-alternative acceptance is the fraction of gold-labeled correct alternatives accepted by the test. Cross-generator rejection is measured on natural incorrect repairs from independent generators, as in Table 16.

Adversary	Train-time completion	Held-out detection	Survival ρ (95% UB)	Audit M
Weak (same-size, untuned)	99.0	71.0	≤ 0.089	32
Same-family (policy sibling)	95.0	78.0	≤ 0.089	32
Strong (frozen, ours)	88.0	85.0	≤ 0.089	32
Deterministic mutants only	100	66.0	≤ 0.046	64

Table 17: Zero-survivor behavior split into train-time completion (fraction of training instances the frozen test drives to zero survivors against that adversary) and detection on a fresh mutant set generated *after* the test is frozen. Training against the strong frozen adversary lowers train-time completion, because its mutants are harder, yet raises held-out detection (66.0 $\rightarrow$ 85.0 vs deterministic-only), the crossover that motivates using a strong adversary. This mirrors the measured fixpoint-rate gain (2.8 $\rightarrow$ 15.6, Table 7). The one-sided bound of Section 2.3 gives 0.089 at M=32 and 0.046 at M=64.

Local-credit scheme	Wrong rej.	Correct-alt acc.	Cross-gen discr.	pass@1
Unsigned kill credit (main objective)	87.4	78.0	74.0	43.0
Uniform token bonus	79.0	72.0	66.0	41.5
Shuffled-credit control	72.0	63.0	55.0	40.0
Signed two-sided credit	88.0	84.0	78.0	43.3

Table 18: Assertion-credit ablation, all four schemes trained and measured under the same protocol. The unsigned row is the main objective used everywhere else in the paper (wrong rejection 87.4 from FPR 12.6, pass@1 43.0) and is tinted for that reason, not because it is the column winner. The signed two-sided rule, which charges an assertion for rejecting a correct alternative, gains most on the side unsigned credit is blind to. Correct-alt acceptance rises 78.0 $\rightarrow$ 84.0, with cross-generator discrimination 74.0 $\rightarrow$ 78.0 and pass@1 +0.3, consistent with penalizing over-rejection. The uniform and shuffled controls confirm the effect requires per-assertion attribution. The evaluation bank includes both correct alternatives and natural wrong repairs.

For a candidate set $\mathcal{C}_k$, coverage is $C_k = \Pr(\exists P \in \mathcal{C}_k : \mathcal{G}(P) = 1)$. If A_k is the probability that the selector returns a correct member of that set, selection efficiency is $Q_k = A_k/C_k$ for $C_k > 0$. The zero-survivor rate is the fraction of instances whose tests detect every scored mutant at the stated audit size. Distinct localizations count unique edited locations across attempts, with empty patches contributing no location.

B.10 Supervised-fine-tuning weighting ablation

Table B.10 isolates the exploration factor of Eq. 1 by setting $\lambda = 0$ while leaving the quality tiers and the efficiency factor in place. It also reports plain rejection-sampling fine-tuning, which keeps only gold-resolved trajectories and is the standard alternative to a weighted corpus. Rejection sampling already improves on uniform weighting, so part of the gain is simply preferring successful trajectories. Turning the exploration factor on adds the rest, raising distinct localizations from 1.74 to 1.98 on Verified and from 3.70 to 4.13 on SWE-PolyBench with matching resolution gains on both suites. Reporting two suites matters here, because a diversity gain confined to the training benchmark would suggest the weighting had fit the corpus rather than taught a search behavior.

The λ=0 row removes only the exploration factor of Eq. 1, isolating its effect on diversity.

B.11 Differential-verification component ablation

We remove one component at a time from full differential-verification search (DVS, Section 2.1) and vary the cell count, on the 500 issues, test-time only.

B.12 Self-Verification Hook

After the agent submits a candidate patch, we inject the prompt below to run an adversarial audit over the agent's *own* patch, applied up to three times per chain and lightly edited here for space.

Self-Verification Hook

You have submitted a candidate fix. Assume it is wrong until you prove otherwise. For this pass you are a hostile, meticulous reviewer of your own work, and your only job is to find a concrete reason the patch fails. A pass that concludes the patch is fine is acceptable only after you have genuinely tried to break it from several angles and failed. The single most common way a wrong patch survives review is to read the

SFT variant (matched data, budget)	pass@1	pass@8	Distinct loc.
Uniform SFT (same successes)	33.7	48.5	1.55
Quality-only weighting	34.1	49.6	1.72
Quality + exploration	34.8	50.7	1.90
Quality + expl. + efficiency (full)	35.2	51.1	1.98
Interface-compliance control	32.6	47.4	1.45

Table 19: Weighted-SFT ablation with an interface-compliance control that fixes format and tool-call handling without exposing task solutions. The full-weighting row is the measured SFT policy (pass@1 35.2, pass@8 51.1, distinct localizations 1.98, Table 2). The exploration term is where distinct localizations rise (1.72 → 1.90), which shows the weighting teaches search behavior rather than tool-use competence.

	SWE-bench Verified			SWE-PolyBench		
Weighting scheme	**p@1**	**p@8**	**dist. loc.**	**p@1**	**p@8**	**dist. loc.**
Uniform weights	33.7	48.5	1.55	21.6	26.9	3.42
T1-only (gold-resolved; plain RFT)	34.1	49.3	1.63	22.5	28.0	3.55
Full, λ=0 (no exploration)	34.8	50.4	1.74	23.2	28.9	3.70
Full weights (ours)	35.2	51.1	1.98	24.2	30.2	4.13

Table 20: SFT weighting ablation (no RL), same corpus and token budget. Plain rejection-sampling fine-tuning (T1-only) already beats uniform weighting, and the exploration factor adds distinct localizations beyond it on both suites, with matching pass@8 gains.

expected answer off what the current code returns, so never do that. Re-establish what correct means from the issue's stated requirement alone.

1. **Re-establish ground truth from the requirement.** Restate the issue in your own words, naming the exact inputs, the exact expected outputs, and the conditions under which each applies. For the failing scenario, derive the correct result directly from the issue text before you look again at what your patch produces. List the source functions the requirement implicates. If your change touches none of them, you almost certainly fixed the wrong place and should relocate before going further.
2. **Re-audit for root cause.** Re-read every region you edited with fresh eyes. For each edit, ask whether it removes the underlying cause or only masks a downstream symptom. Swallowing an exception, clamping a value into range, and special-casing the issue's literal input are all symptom fixes, and you must reject them. Trace the data and control flow by hand, confirm the fix holds for the whole class of inputs the requirement covers, not the reported one alone, and confirm it leaves previously correct inputs unchanged.
3. **Test adversarially.** Run the issue's FAIL_TO_PASS tests first, then go past them. Write a throwaway script that constructs the expected result you derived in step 1 and asserts your code matches it, independent of the shipped tests. Run the repository's existing tests for every file you touched, since a newly failing one is a regression you introduced. Delete all scratch scripts afterward so they never leak into the diff.
4. **Enumerate edge cases, where the hidden tests live.** Passing the visible tests is necessary but far from sufficient. Work through empty, None, missing, and unset inputs, then zero, negative, boundary, and just-past-boundary values, then malformed or unexpected types, then unicode and whitespace, then duplicate, single-element, and out-of-order collections, then the error path itself, then idempotency under repeated application, and finally interaction with adjacent options. For each, decide the correct behavior from the requirement and confirm the code does exactly that.
5. **Guard regressions and completeness.** Confirm you did not rename, weaken, or delete any existing test. Remove debug prints and unrelated refactors so the diff stays minimal. A diff that is only whitespace, comments, or import churn cannot satisfy a behavioral requirement, so treat that as a failure. Confirm the diff modifies the exact source region the requirement implicates.

If any step surfaces a problem, fix it at the root cause, re-run the relevant tests, and keep auditing from the top. Call `submit` only when every step passes under genuinely hostile scrutiny. End every message, and every message after it, with exactly one tool call.

B.13 The Eight Methodology Preambles

One of the preambles below, prepended to the base system prompt, conditions each of the eight parallel chains in the methodology-diversity arm, so the chains differ by conditioning rather than by temperature. We give the eight in canonical order, lightly edited for space.

Configuration	Resolved /500	Rate	Steps/solve	Early-stop /500	Loc. pruned
Full DVS (8 cells)	264	52.8%	503	102	126
− early stopping	264	52.8%	728	0	126
− location pruning	256	51.2%	598	99	0
− reverted-tree run (patched only)	237	47.4%	665	118	0
− regression / coverage checks	261	52.2%	511	106	129
4 cells (vs 8)	246	49.2%	452	77	94

Table 21: Differential-verification component ablation (test-time only). Section 2.1 reads the two-tree differential and early-stopping rows. Location pruning contributes 1.6 points and halving the cell count costs 3.6 points.

(1) `baseline`, pragmatic control

Coding approach. Pragmatic baseline, the experimental control.

You are the control chain. Solve the issue with competent, unconstrained engineering judgment and adopt no special methodology. The other seven chains layer discipline on top of a run like this one, and the comparison is only meaningful if this chain stays neutral, so do not import their ideas. Work the way a strong engineer works a normal ticket, moving quickly but never carelessly.

1. **Restate the requirement.** Put the required behavior in your own words, derived from the issue text, not the current code. Name the inputs, the expected outputs, and the conditions that select between them.
2. **Locate the code.** Search for the symbols, function names, and error strings named in the issue, and read the files the search returns. Follow the real call path, not a guess from names.
3. **Form and confirm a hypothesis.** State a concrete root-cause hypothesis about which line or function produces the wrong result, then confirm it by reading the actual code path, not by assuming.
4. **Make the smallest correct change.** Fix the underlying cause with the least code that does the job. Touch no unrelated code and never edit a test file.
5. **Verify and refine.** Run `run_tests`. On failure, read the output, revise the hypothesis if needed, and repeat. Do not patch over the symptom.
6. **Review the diff.** Skim for stray debug code and sanity-check one or two edge cases the requirement implies.

Be direct and economical throughout. End every message with exactly one tool call, and call `submit` once the tests pass and the change looks clean.

(2) `tdd`, strict Test-Driven Development

Coding approach. Strict TDD, red then green then refactor.

Drive the entire fix from tests, in tight red, green, refactor cycles. You may not write production code except to make a currently failing test pass, so the test always comes first and the bar decides when you are done.

1. **Understand the contract.** Express the required behavior as concrete assertions derived from the issue, never from what the code currently returns. Each assertion should pin a specific input to its specific expected output.
2. **Red.** Identify or write the test that captures the bug. Run it and confirm it fails, and confirm it fails for the right reason, not an unrelated import or fixture error.
3. **Green.** Write the minimal production code that turns the bar green. Resist premature generalization and add nothing the current failing test does not demand.
4. **Re-run.** Confirm the new test passes and that nothing else broke. Treat any newly red test as part of this cycle, not a later cleanup.
5. **Refactor.** Clean up only the code you just introduced, and stay green the whole time.
6. **Expand coverage.** For each edge case the requirement implies, add an assertion that starts red, then make it green. Continue until the tests pin the behavior the requirement describes.

Test after every change and let the bar, not intuition, decide when you are finished. Do not modify existing tests. End every message with exactly one tool call, and call `submit` only when all relevant tests are green.

(3) `bdd`, Behavior-Driven Development

Coding approach. Behavior-Driven Development, Given/When/Then.

Frame the whole fix around externally observable behavior, expressed as concrete scenarios, then make each scenario hold. You care about what the system does from the outside, not about implementation details, so every scenario names a state, an action, and an outcome you could check by running the code.

1. **Specify behavior.** Translate the issue into two to four scenarios in Given, When, Then form, where Given is the initial state or inputs, When is the action or call, and Then is the expected observable outcome. Cover the primary success scenario and the notable alternative and error scenarios the issue implies, such as bad input, the boundary, and the feature disabled. Derive every Then from the requirement, not from current behavior.
2. **Ground each scenario in code.** Identify the function, endpoint, or CLI path that realizes each When, and the concrete value or state that realizes each Then. Use `search` and `read_file` to connect scenarios to specific call sites so you know exactly where behavior is produced.
3. **Make scenarios executable.** Rely on `run_tests` for the issue's own tests. For behavior they do not cover, write a small `run_bash` script that sets up the Given, performs the When, and asserts the Then, then note which currently fail.
4. **Implement scenario by scenario.** Make the smallest change that turns one failing scenario green without breaking the others, and prefer changes expressed in terms of the observable contract.
5. **Verify all scenarios**, including the alternative and error ones. A patch that satisfies the happy path but mishandles an error scenario is incomplete by this methodology's standard.
6. **Re-read and confirm.** Check each Given, When, Then against the final behavior and confirm it truly holds.

Think continuously in terms of observable outcomes. Do not edit existing tests, and clean up any scratch scripts. End every message with exactly one tool call, and call `submit` when every scenario holds.

(4) `agile`, incremental delivery

Coding approach. Agile incremental delivery, smallest shippable slices, fast feedback.

Deliver the fix as a sequence of tiny, independently verifiable increments, getting feedback after each, not attempting one large change you cannot easily debug when it goes wrong. The tight feedback loop and the willingness to adapt the plan are the essence of this approach.

1. **Slice the work.** Decompose the fix into the smallest increments that each leave the code in a working, checkable state. Order them by value and by risk, doing the change that most reduces uncertainty first. A typical order is to reproduce the bug with a minimal script, then apply the core minimal fix, then harden for edge cases. Each slice should be small enough that if it is wrong you immediately know which change caused it.
2. **Implement one slice.** Make the minimal edit for the current slice only. Avoid bundling unrelated changes into a slice, since that destroys the fast-feedback property.
3. **Get feedback immediately.** After each slice run `run_tests` or a focused `run_bash` check, and treat the result as the signal for whether to proceed, adjust, or revert this slice.
4. **Iterate and adapt.** Advance to the next slice only when the current one is verified. If a slice reveals the plan was wrong, re-plan the remaining slices then and there.
5. **Integrate.** After the final slice, run the full relevant test set to confirm the increments compose correctly, and review the cumulative diff to ensure it stayed minimal and coherent.

Favor many small verified steps over one big leap. Do not edit existing tests, and clean up scratch artifacts. End every message with exactly one tool call, and call `submit` when the increments are complete and the relevant tests pass.

(5) `tdd_agile`, TDD × Agile hybrid

Coding approach. TDD crossed with Agile, test-first in small increments.

Combine red, green, refactor discipline with incremental delivery, so each Agile increment is itself a complete TDD micro-cycle. This is the explicit hybrid under test, so honor both disciplines fully.

1. **Plan increments.** Break the fix into the smallest value-ordered, risk-ordered slices, exactly as in Agile, and sketch the sequence before you start coding.

2. **Run a full TDD cycle per increment.** Write or identify a test asserting that increment's expected behavior, derived from the requirement, and confirm it fails for the right reason. Make the minimal production change that passes it and nothing more. Re-run to confirm the new test is green and that earlier increments' tests still pass, so slices do not regress as you stack them. Tidy only what you just added, staying green.
3. **Adapt between increments.** If an increment's red test or its result reveals the plan was wrong, re-plan the remaining increments before continuing.
4. **Integrate.** After the final increment, run the full relevant suite. Together the increments' tests should pin the primary behavior and the edge cases the issue implies.

Two hard rules always hold. Never write production code without a failing test demanding it, and never begin a new increment before the current one is green. This yields tight correctness and adaptability at once. Do not edit existing tests, and clean up scratch scripts. End every message with exactly one tool call, and call `submit` only when every increment is green and the relevant tests pass.

(6) `root_cause`, rigorous root-cause analysis

Coding approach. Fix the disease, never the symptom.

Refuse to apply any edit until you can name the precise underlying cause and explain the complete causal chain from that cause to the observed symptom. Most bad patches treat symptoms, and you will not.

1. **Reproduce deterministically.** Use `run_bash` to write and run a minimal script that triggers the exact failure described in the issue. If you cannot reproduce it reliably you do not yet understand it, so keep investigating until you can summon the failure on demand.
2. **Trace backward.** Starting from the symptom, an exception, a wrong return value, or a corrupted state, follow the call path upstream with `search` and `read_file` on each frame until you reach the precise line where the program's state first becomes wrong. Carefully separate the proximate trigger, where it blows up, from the true origin, where the wrongness is introduced.
3. **Name the cause.** State it explicitly and specifically as a falsifiable claim about concrete lines. If your statement is vague, such as something about None handling, you have not found it yet, so keep digging.
4. **Confirm by prediction.** Before changing anything, predict from your hypothesis exactly which inputs should fail and which should pass, then verify that prediction with `run_bash`. If reality disagrees, the hypothesis is wrong, so revise it and repeat.
5. **Fix at the origin.** Change the code at the true cause so the entire class of failures resolves, not just the one input in the issue. Explicitly forbid yourself the symptom patches, namely catching and ignoring exceptions, clamping or defaulting outputs to mask a bad computation, and special-casing the failing example.
6. **Verify the class is fixed.** Confirm the reproduction now passes, that related inputs in the same class pass, and that `run_tests` is green. The fix should generalize because it targets the cause.

Do not edit existing tests, and clean up your reproduction scripts before submitting. End every message with exactly one tool call, and call `submit` only when the root cause is provably resolved.

(7) `defensive`, defensive programming

Coding approach. Robust against the inputs you were not given.

Assume the fix will be exercised by inputs and conditions well beyond the issue's example, because the hidden tests probe exactly those. Make the change correct for the reported case and resilient across the input space, without over-engineering.

1. **Fix the reported issue first**, at its root cause, so the failing case passes. Establish that baseline before you begin hardening.
2. **Enumerate adversarial conditions** for the changed code and verify behavior for each with `run_bash` or `run_tests`. Cover empty, None, missing, and unset values, then zero, negative, minimum, maximum, and just-past-boundary values, then malformed or unexpected types and very large or deeply nested inputs, then unicode, encoding, and whitespace, then duplicates and single-element and out-of-order collections, and finally repeated invocation and the error path itself.
3. **Decide the correct response** for each condition, guided by the requirement and the project's existing conventions. Choose whether to raise a specific, well-messaged error, return a documented sentinel, clamp, or no-op, then make the code do that deliberately, not crash opaquely or silently corrupt state.

4. **Place guards at the right layer.** Validate at the boundary or entry point, not by scattering ad-hoc checks throughout, and keep the guards minimal, consistent with the surrounding style, and non-redundant.
5. **Re-verify valid inputs.** Confirm the original tests still pass and that your hardening did not change behavior for legitimate inputs, since a defensive check that rejects a valid case is a new bug.

Robustness is the goal, but stay minimal and in-style and do not gold-plate. Do not edit existing tests, and clean up scratch scripts. End every message with exactly one tool call, and call `submit` when the fix is correct and hardened.

(8) `exploratory`, exploration-first

Coding approach. Build a complete mental model before editing.

Fixing the wrong place, a localization error, is a leading cause of failure on real codebases. Counter it by investing heavily in understanding before you change a single line.

1. **Survey broadly.** Map the repository, its top-level layout, the package the issue concerns, and how the relevant module connects to the rest of the system. Read the README and docs sections that describe the contract you are about to touch so you know the intended design.
2. **Search widely.** For every symbol, error message, configuration key, and concept named in the issue, `search` the codebase and `read_file` the hits. Include the tests, to learn the intended contract and examples of correct usage, and the call sites, to learn how the code is actually used. Cast a wide net before narrowing, and note related code you did not expect.
3. **Synthesize a model.** Write down how the feature is supposed to work end to end, where the current behavior diverges, and the two or three most plausible locations for the true fix, ranked with the reasoning for each. Commit to a location only once this model feels solid.
4. **Confirm the location.** Reproduce the bug and watch it flow through the code you suspect, so you have direct evidence the fix belongs there and not somewhere upstream or downstream.
5. **Fix surgically.** Make a minimal, in-context change at the well-understood location, matching the surrounding conventions.
6. **Verify broadly.** Run `run_tests` plus checks for the edge cases your exploration surfaced, since deep understanding usually reveals edge cases a narrow approach would miss.

Depth of understanding first, surgical change second. Do not edit existing tests, and clean up scratch artifacts. End every message with exactly one tool call, and call `submit` once the well-understood fix is verified.

B.14 Reactive Location-Ban Logic

The reactive arms walk the candidate grid sequentially, grouping cells by location in confidence order, and let the verifier steer subsequent attempts.

Ban, freeze, early-stop. Three rules govern each cell. *Ban on confirmed fail*. The walk bans a location proven wrong, skips its remaining approaches, and moves to a new site. *Freeze site, vary edit*. A weak pass means the site is plausibly right and the edit is wrong, so the walk keeps the location and varies the approach. *Early stop*. The walk stops on the first strong pass.

The walk enforces bans by *filtering*, holding banned location keys in a set and skipping any cell whose location is in that set.

Deprioritize trigger (condensed, `driver.py` 175 to 193)

```
def location_wrong(ver: dict) -¿ bool:
# Deprioritize trigger = the location is LIKELY wrong. The strongest signal for a wrong
# *location* is the reverted-differential going vacuous: reverting the edit did NOT move the
# symptom, so this reproduction did not exercise the fault (edit off-path, or repro too weak).
# A red repro/regression means the EDIT is wrong, NOT the location, so banning on those is a
# false ban. We therefore deprioritize ONLY on a confirmed-vacuous differential; edit-quality
# failures fall through to freeze-site/vary-edit, and a deprioritized site can be revisited.
# 'inconclusive'/missing differential is treated as NOT-wrong (never ban on uncertainty).
return ver.get("differential") == "vacuous"
```

A second guard gates the ban on the edit being real and on-plan, since a cell that drifted off its assigned file never tested that location and must not ban it. The localization-ban configuration uses the explicit "look elsewhere" prompt below.

Location-ban injection

Hard constraint. The source locations below are ruled out by the current reproduction.

This is a fresh, independent attempt, but not the first. Earlier attempts edited the source locations listed below, and we ran their patches. The issue's reproduction together with the repository's existing tests demonstrated by execution that a fix at those locations does not resolve the issue. Treat them as ruled out, not merely unpromising.

{body}

Your single hard constraint is where you fix the bug. Make your change in a different location than every entry above, meaning a different source file or a different function. You are free in how you fix it.

1. **Understand the failure from the requirement.** Reproduce the issue and derive the correct behavior from the issue text, not from the current code. You need a clear picture of the intended result before you choose a new site.
2. **Trace the fault to a new location.** The measured failure mode is fixation, where engineers keep editing the first place they looked. You have executed evidence those places are wrong, so walk the fault's call path outward toward the entry point and inward toward the helpers. Choose a genuinely different site, such as a deeper function that produces the wrong value, an intermediate transformer, the public entry point, or a helper that supplies a wrong precondition.
3. **Fix at the new location and verify hard.** Implement the minimal correct fix for the whole class of inputs, re-run your reproduction and the repository's tests, and probe the edge cases the requirement implies. Never edit a test file.

Escape hatch, to be used sparingly. If after honest investigation you are convinced the fix must live at a banned location, you may return there, but you must explain why the prior attempt failed and how your change differs. Your strong default is to look somewhere new first.

B.15 Experiment Configurations

The three meanings of k=8. All three test-time axes use k=8 and mean something different by it. For *pass@8*, eight attempts run with sample 1 greedy and samples 2 to 8 at T=0.7. The samples are independent but not identically distributed, since the greedy sample comes from a different distribution than the temperature ones, and diversity comes only from temperature. For the *eight methodologies*, eight parallel chains each take a distinct methodology preamble (Appendix B.13), so diversity comes from conditioning and the chains explore different solution axes. For the *coverage-space arms*, eight candidate cells per instance differ in how diversity is enforced and how candidates are verified, which isolates grid conditioning from the reactive verifier.