BEYOND FAIL-TO-PASS: ITERATIVE HARDENING OF CO-GENERATED BUG REPRODUCTION TESTS AND FIXES

Anonymous authors
Paper under double-blind review

ABSTRACT

Bug reproduction tests (BRTs) turn bug reports into executable signals for automated program repair, but the usual fail-to-pass (F→P) criterion does not establish whether a test supports correct repair. A test may fail on buggy code and pass on the golden fix while also accepting plausible-but-incorrect patches. We introduce Mutation Patch Evaluation (MPE) to distinguish rigorous tests from such lax tests, and show empirically that downstream repair gains concentrate in the rigorous tier. We also identify error coupling in co-generation, where a generated test accepts an incorrect generated patch. Based on these findings, we propose COHARDEN, a co-generation framework for iterative test-and-fix refinement. An initial test-only round targets the observable bug symptom before committing to a fix. Subsequent hardening rounds use mutated variants of the current fix to strengthen the test and refine the patch. On SWE-bench Verified, COHARDEN reaches 69.4% Resolved, improving on the strongest fix-only and co-generation baselines by 9.6 and 7.9 percentage points, respectively. Further experiments show gains across LLM backbones and benchmarks. Code is available at `https://anonymous.4open.science/r/COHARDEN-9753`.

1 INTRODUCTION

Automated Program Repair (APR) has advanced rapidly with Large Language Models (LLMs), and recent agentic systems can generate increasingly plausible fixes for real-world software bugs (Xia et al., 2023; Yang et al., 2024; Zhang et al., 2024; Xia et al., 2025). However, a plausible fix is not necessarily correct. When APR is driven directly from natural-language issue reports, the task remains fundamentally underconstrained: bug reports are often incomplete, many candidate patches appear superficially reasonable, and passing the existing test suite does not guarantee that the reported bug has actually been fixed (Qi et al., 2015; Wang et al., 2026). This problem is compounded by the fact that, when an issue is first reported, the existing test suite often does not yet contain a reproduction test for that bug (Kang et al., 2023; Mündler et al., 2024; Khatib et al., 2026). Bug Reproduction Tests (BRTs) help close this gap. A BRT is generated from the natural-language issue description and is designed to expose the reported bug. By turning the issue report into an executable, bug-specific signal, BRTs can validate candidate patches, constrain the downstream repair space, and filter out plausible-but-incorrect fixes (Nashid et al., 2026; Fei et al., 2026; Mündler et al., 2024). This role has made BRT generation increasingly important in APR, and has also motivated recent systems to co-generate tests and fixes in iterative loops, where generated tests guide repair and generated fixes in turn shape tests (Cheng et al., 2026; Li et al., 2026b; 2025).

The dominant criterion for evaluating generated BRTs is the Fail-to-Pass (F→P) property: a test must fail on the buggy version and pass on the fixed one. While useful, F→P only checks whether a test detects the reported bug under one reference fix; it does not assess whether the test sufficiently constrains repair by ruling out plausible but incorrect fixes. Figure 1 illustrates this with `sympy-21612`, where a SymPy printer omits brackets around `1/X` denominators. Three reproduction tests all detect the buggy behavior but differ sharply in repair guidance. A *lax* test satisfies F→P yet accepts both the golden fix and an incorrect simplify-away fix. A *rigorous* test accepts the golden fix while rejecting the incorrect one. A *misaligned* test instead accepts the incorrect fix and rejects the golden fix.

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.

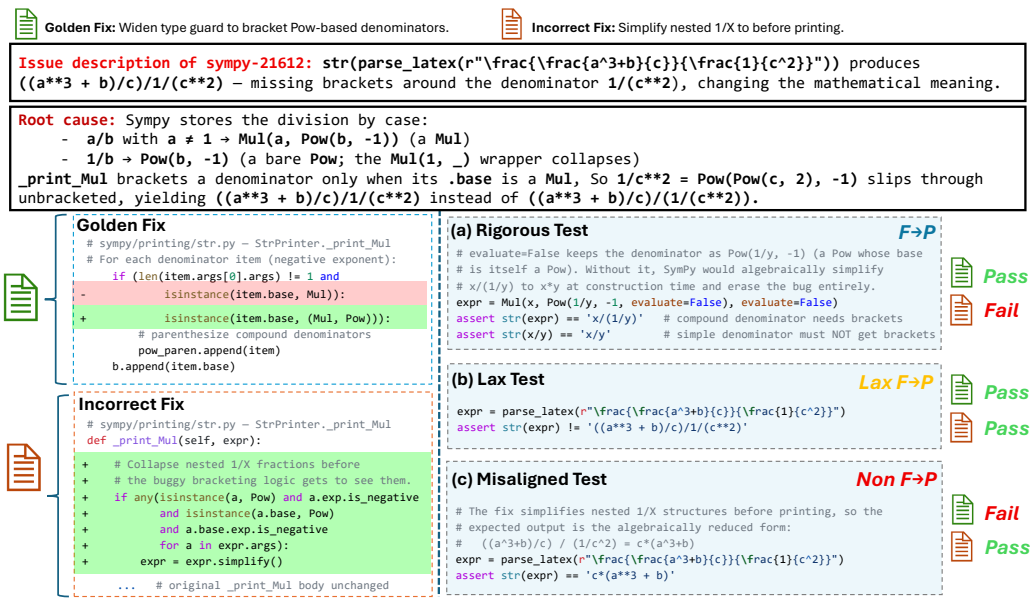

Figure 1: Three reproduction tests for SymPy issue `sympy-21612` from SWT-bench (Mündler et al., 2024). The printer omits brackets around `1/X` denominators, changing the mathematical meaning. The golden fix widens the type check; the incorrect fix simplifies away the triggering input without repairing that check. All three tests fail on the buggy code. (a) The rigorous test checks exact output on two probes, accepting the golden fix and rejecting the incorrect one. (b) The lax test only requires output to differ from the buggy string, accepting both fixes. (c) The misaligned test requires the simplified form `c*(a**3+b)`, accepting the incorrect fix and rejecting the golden one.

Current BRT evaluation does not distinguish these cases: both rigorous and lax tests satisfy F→P, although only the rigorous test properly constrains the repair space.

Current approaches miss this repair-constraining dimension. Methods that generate BRTs separately (Nashid et al., 2026; Fei et al., 2026) treat the task largely as F→P optimization, while cogeneration methods (Li et al., 2026b; 2025) evolve tests and fixes jointly but still do not distinguish lax from rigorous tests. Post-hoc variant analysis (Li et al., 2026a) similarly uses fix variants to expose inadequate tests, but it requires gold fixes and is aimed at benchmark analysis rather than in-loop repair. As a result, current methods cannot tell whether a generated BRT merely reproduces the symptom or truly constrains repair toward the correct fix.

To study this, we introduce *Mutation Patch Evaluation* (MPE), which evaluates a BRT against plausible-but-incorrect fixes and classifies it as *Rigorous*, *Lax*, or *Misaligned*. Using MPE, we analyze generated BRTs in two settings (Section 2): *BRT injection*, where a separately generated BRT guides a fix agent, and *cogeneration*, where one agent jointly produces the fix and BRT. We find that the benefit of F→P-passing BRTs comes entirely from the Rigorous tier ($\Delta = +8.5$ Resolved), while Lax tests provide no gain ($\Delta = 0.0$). We also find that cogeneration often produces Lax tests together with correlated fix errors, with a joint-failure rate $1.87\times$ the independent prediction.

Building on these findings, we propose CoHarden (Section 3), a two-phase cogeneration loop. *Lax-init* generates a reproduction test t_0 before any fix attempt, reducing early Misaligned tests and test–fix coupling. *Hardening* then iteratively refines the (t_k, c_k) pair against a per-round mutant pool, using a *Temporal Matrix* that converts MPE's within-pool signal into a reference-free in-loop signal. On the SWE-bench ∩ SWT-bench Verified subset, CoHarden reaches 69.4% Resolved, outperforming vanilla cogeneration on the same backbone (OpenHands + GPT-5-mini) by $+8.0$ points. In summary, we make three contributions:

- We introduce *Mutation Patch Evaluation* (MPE), a finer-grained evaluation framework that partitions BRTs into a three-way taxonomy (Rigorous, Lax, Misaligned), exposing a quality dimension orthogonal to F→P.

- Using MPE, we conduct an empirical study (Section 2) that identifies a research gap between automatic BRT generation and downstream BRT utility: F→P-passing BRTs concentrate their fix-side benefit in the Rigorous tier; cogeneration further produces Lax tests in its own outputs, and its Misaligned tests strongly correlate with incorrect fixes.
- We design CoHarden, a two-phase cogeneration loop (lax-init, hardening) that turns MPE into a reference-free in-loop signal via a *Temporal Matrix* (with the previous-round test in place of t_{golden}); the test is hardened against a per-round mutant pool while the agent updates the fix in lockstep.

2 Empirical Study

Bug-reproducing tests (BRTs) are intended to expose a reported bug and verify whether a candidate fix resolves it, but F→P alone does not reveal how effectively they constrain repair. Figure 1 shows a failure case where a lax test satisfies F→P yet still accepts an incorrect fix. We introduce *Mutation Patch Evaluation* (MPE, Section 2.1), which partitions generated BRTs into Rigorous, Lax, and Misaligned, and use it to measure their effect in two settings: a *standalone setting* (Section 2.2), where the BRT is provided to a fix agent as guidance, and a *cogeneration setting* (Cheng et al., 2026; Li et al., 2026b; 2025) (Section 2.3), where one agent jointly produces the fix and the BRT.

2.1 Mutation Patch Evaluation (MPE)

We follow the standard SWT-Bench setup (Mündler et al., 2024), where each instance provides a buggy program c_{buggy}, a golden fix c_{golden}, and a golden test t_{golden} as the human bug specification. Under this setup, a generated reproduction test t_{gen} is typically judged by F→P: it fails on c_{buggy} and passes on c_{golden}. However, F→P only checks rejection of c_{buggy} and acceptance of c_{golden}; it does not test whether t_{gen} rejects plausible but incorrect fixes. Inspired by classical mutation testing (DeMillo et al., 1978; Jia & Harman, 2011), we therefore evaluate t_{gen} on a pool of *semantic* mutations of c_{golden} to formalize the three-way classification into Rigorous, Lax, and Misaligned.

Mutation patch pool. We construct a *mutation patch pool* $\mathcal{C} = \{c'_1, c'_2, \ldots, c'_N\}$ that approximates the space of plausible-but-incorrect repairs. Each c' is generated by an LLM under one of five *semantic operators*: *vanilla*, *symptom suppression*, *incomplete fix*, *input-specific shortcut*, and *behavior substitution*. These operators target common ways an incorrect fix can pass a reproduction test. In Figure 1, the Incorrect Fix simplifies the input expression, matching the behavior-substitution pattern, and the Lax test in panel (b) accepts it because the test only asserts that the output differs from the known-buggy string. Section B.1 gives operator definitions, sampling budget, and full prompts.

Confusion matrix. Running each $c' \in \mathcal{C}$ against both t_{gen} and t_{golden} yields a 2×2 confusion matrix.

	c' *passes* t_{golden}	c' *fails* t_{golden}
c' *passes* t_{gen}	α (agree-pass)	β (**Laxity cell**)
c' *fails* t_{gen}	γ (**Rigor cell**)	δ (agree-fail)

The diagonal cells (α, δ) collect the mutation patches on which the generated and golden tests agree, while the two off-diagonal cells expose failure modes F→P conflates. Cell β counts mutation patches that pass t_{gen} but fail t_{golden}: the test detects the bug, yet still admits incorrect repairs and so under-constrains the repair space. Cell γ counts patches that fail t_{gen} but pass t_{golden}: the test rejects behaviorally valid repairs and imposes inconsistent constraints. We summarize each test by β/total, the share of mutation patches on which t_{gen} is laxer than t_{golden}.

Three-way partition. Combined with the F→P status of t_{gen} on c_{golden}, the 2×2 matrix yields three categories: **Rigorous** (F→P with below-median β/total), where the test detects the bug and constrains repairs consistently with the golden test; **Lax** (F→P with above-median β/total), where the test detects the bug but still admits many plausible yet incorrect repairs; and **Misaligned** (non-F→P), where the test rejects c_{golden} and therefore conflicts with the golden correctness specification.

2.2 Bug Fixing with BRT Injection

We first study how BRTs from BRT-only methods affect downstream bug fixing. These methods treat BRT generation as a standalone task and produce a BRT independently of fix generation. We use four representative BRT-only methods: LogicStar (LogicStar AI, 2025), e-Otter++ (Ahmed et al., 2026), AssertFlip (Khatib et al., 2026), and OpenHands in BRT-only mode (Wang et al., 2025).

Table 1: Bug fixing utility of generated BRTs on SWT-Bench. Δ is the absolute change in RESOLVED. The performance gain mainly comes from the generated F→P BRTs that are consistent with the golden tests (i.e., Rigorous), while the non-F→P BRTs (i.e., Misaligned) hurt the fixing performance.

Test class	# instances	FIX-ONLY	with-BRT	Δ
Rigorous	552	67.3%	75.8%	+8.5
Lax	552	69.5%	69.5%	+0.0
Misaligned	474	44.5%	40.9%	−3.6

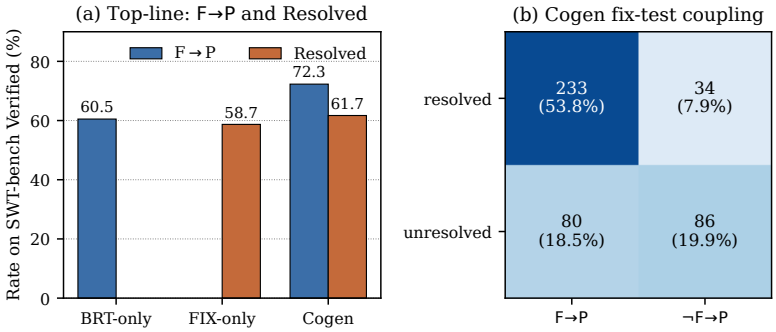

Figure 2: Cogeneration vs. single-task baselines on SWT-Bench. **(a)** COGEN (joint fix+test in one trajectory) exceeds each single-task baseline (BRT-ONLY on F→P, FIX-ONLY on RESOLVED), but its fix-side gain is modest. **(b)** The Lax pattern of Section 2.2 surfaces inside COGEN's own output: 80 of the 166 unresolved instances pair an F→P-passing test with an incorrect fix, and the joint-failure cell is 1.87× the rate predicted under independent fix and test errors.

Setup. We compare three configurations of the same underlying agent (OpenHands + GPT-5-mini) on SWT-Bench (Mündler et al., 2024): COGEN jointly produces a fix and a BRT in one trajectory; BRT-ONLY produces only the BRT (evaluated by F→P against the golden fix); and FIX-ONLY produces only the fix (evaluated against the golden test).

BRTs that satisfy F→P help the fix agent on average, but the gain concentrates in the rigorous subset. On the 1,104 F→P pairs, adding BRTs lifts Resolved from 68.4% to 72.6% ($\Delta = +4.3$), while on the 474 non-F→P pairs it drops from 44.5% to 40.9% ($\Delta = -3.6$), validating F→P as a coarse filter. However, within the F→P pool, applying MPE (Section 2.1) reveals that the entire fixing gain comes from the rigorous half ($\Delta = +8.5$), whereas the lax half contributes none ($\Delta = 0.0$) (Table 1). The same pattern holds within each individual method; detailed results are provided in Section B.2.

Finding 1

F→P agrees with downstream bug fixing utility as a coarse binary filter. Within the F→P pool, however, the fixing gain concentrates in the rigorous half: F→P alone is too coarse and collapses a within-pool quality dimension that materially determines BRT utility.

2.3 BUG FIXING WITH COGENERATION

In practice, human developers typically resolve a bug by writing the fix and the reproduction test together rather than as two separate tasks. *Cogeneration* (Cheng et al., 2026) methods mirror this workflow by jointly producing the fix and the BRT in a single trajectory. We study, under this setting, whether the cogenerated BRT actually contributes to downstream bug fixing.

Setup. We compare three configurations of the same underlying agent (OpenHands + GPT-5-mini) on SWT-Bench (Mündler et al., 2024): COGEN jointly produces a fix and a BRT in one trajectory (Cheng et al., 2026); BRT-ONLY produces only the BRT, evaluated by F→P against the golden fix; and FIX-ONLY produces only the fix, evaluated against the golden test.

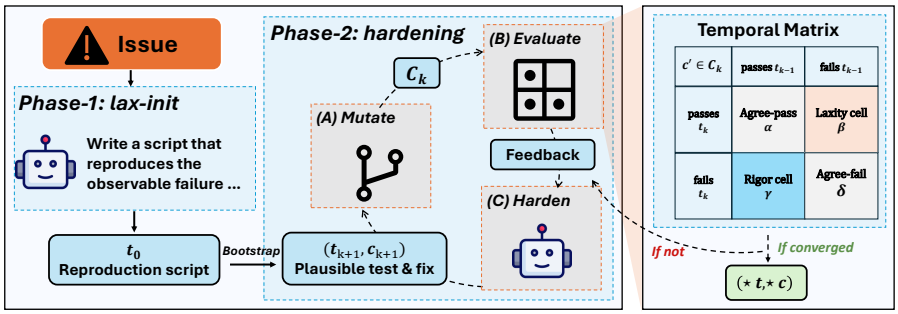

Figure 3: **CoHarden pipeline. Phase 1 (Lax-init):** given the issue, the agent writes a reproduction test t_0 targeting the observable failure without source edits. **Phase 2 (Hardening):** bootstrapping t_0 into a plausible pair (t_1, c_1), the loop iterates three steps per round k: **(A) Mutate** produces a mutant pool $\mathcal{C}_k$ from c_k via four targeted operators; **(B) Evaluate** runs each $c' \in \mathcal{C}_k$ against t_k and t_{k-1} to fill the Temporal Matrix with cells α (agree-pass), β (Laxity), γ (Rigor), δ (agree-fail); **(C) Harden** feeds the Temporal Matrix back to the agent, which strengthens the test (and updates the fix) into the next pair (t_{k+1}, c_{k+1}). The loop terminates when no Laxity cell remains, returning final $(\star t, \star c)$.

Cogeneration beats both specialists, but the fix-side gain is marginal. Figure 2(a) shows that CoGen exceeds each single-task baseline: test F→P 72.3% (+11.8 over BRT-only) and Resolved 61.7% (+3.0 over Fix-only).

The Lax pattern also appears in the cogeneration setting. For 80 (48.2%) of CoGen's 166 unresolved instances, the cogenerated test F→P-passes on the golden fix (Figure 2(b), bottom-left cell) yet the cogenerated fix is wrong. In these trajectories, the agent stopped after observing its own test pass on its own fix, so the cogenerated test passes on *both* the wrong fix and the golden fix, exhibiting the Lax pattern of Section 2.2. A Lax test lets an incorrect fix satisfy the in-trajectory F→P signal, triggering early stopping while the bug remains unfixed.

CoGen's fix and test errors are correlated. An ideal joint formulation would derive the fix and test from the issue specification independently, so that errors on each would be uncorrelated. Figure 2(b) shows that CoGen's joint-failure rate is 19.9% (1.87× the 10.6% predicted under independence), with $P(\text{fix wrong} \mid \text{test wrong}) = 71.7\%$ and $P(\text{test wrong} \mid \text{fix wrong}) = 51.8\%$. This suggests that the cogenerated fix and test share a common (mis)interpretation of the intended behavior.

Finding 2

Cogeneration's joint fix-side gain over a fix specialist is marginal. The Lax pattern from Section 2.2 resurfaces within CoGen's own output, and its fix and test errors are correlated rather than independent: F→P between two artifacts of one trajectory cannot serve as an independent correctness check.

3 CoHarden: Iterative-Hardening Cogeneration

We design CoHarden (Figure 3), a two-phase cogeneration loop, to mitigate two failure modes identified in Section 2: Lax tests that admit incorrect fixes and Misaligned tests that reject even the golden fix. *Phase 1: Lax-init* (Section 3.1) writes a reproduction test t_0 before any fix attempt to avoid an early Misaligned t_0. *Phase 2: Hardening* (Section 3.2) bootstraps t_0 into a (t_1, c_1) pair and iteratively refines it through rounds of *Mutate*, *Evaluate*, and *Harden*, with a Temporal Matrix flagging per-round Lax regression. We detail Phase 1 in Section 3.1 and Phase 2 in Section 3.2.

3.1 Lax-init: produce the test before the fix

Misaligned tests harm the fix process. Section 2.2 shows that injecting a non-F→P (Misaligned) BRT into the fix agent lowers Resolved below the no-BRT baseline: the wrong test misleads the agent

rather than helping it. The same coupling appears inside cogeneration, where the cogenerated fix tends to be wrong when the cogenerated test is wrong (Section 2.3).

To avoid Misaligned t_0, CoHarden mirrors how human developers begin debugging: write a short script that captures the observable failure (printed output, raised exception, return-value discrepancy) before reasoning about the implementation. The agent therefore emits only a reproduction test t_0 and is forbidden from modifying any source file; the prompt steers t_0 toward the most observable witness of the bug and away from internal hooks or strict structural assertions. Writing the test first also weakens the cogeneration coupling of Section 2.3, since t_0 has no current fix to anchor against.

3.2 Hardening: cogenerate and tighten the (test, fix) pair

After Phase 0 produces t_0, Hardening iteratively refines a (t_k, c_k) pair over rounds $k = 1, \ldots, K$. At each round, the agent debugs from the previous round to locate the root cause and jointly emit an F→P-passing pair (t_k, c_k): t_k fails on the buggy code with c_k reverted, and passes on c_k. Joint generation gives the test concrete internal behaviors to check, and Section 2.3 provides empirical support that cogeneration raises test F→P above the BRT-only baseline.

The F→P gate is binary, however, and cannot tell Rigorous from Lax, so passing it is not by itself a sign of progress. CoHarden therefore equips each round with a reference-free within-pool signal derived from MPE, and uses that signal both to render feedback to the agent and to discard rounds that weaken the test rather than tighten it. In Section 2.1, the mutant pool is generated from c_{golden} and the confusion matrix grades the test against t_{golden}. Both golden artifacts are unavailable inside the loop, so we substitute the agent's round-k fix c_k for c_{golden} and the previous-round test t_{k-1} for t_{golden}. At each round k, after the agent emits an F→P-passing (t_k, c_k), the loop applies the four targeted operators of Section 2.1 to c_k to produce a mutant pool $\mathcal{C}_k$, and running each $c' \in \mathcal{C}_k$ against both t_k and t_{k-1} yields the *Temporal Matrix*:

	c' *passes* t_{k-1}	c' *fails* t_{k-1}
c' *passes* t_k	α (agree-pass)	β (**Laxity cell**)
c' *fails* t_k	γ (**Rigor cell**)	δ (agree-fail)

The matrix mirrors MPE's confusion matrix with t_{k-1} in place of t_{golden}: β is the *Laxity* cell (per-round regression of t_k relative to t_{k-1}) and γ the *Rigor* cell (per-round improvement). CoHarden uses the Laxity cell to discard non-tightening rounds. When $\beta > 0$ at round k, the loop terminates and returns the previous-round pair (t_{k-1}, c_{k-1}). The loop thus converges when no Laxity cell remains, matching the early-round concentration of hardening's benefit observed in Section 4.3. Each round renders operator-wise counts and samples of t_k-passing mutants to the agent; the next-round prompt instructs it to harden the test against these mutants while updating the fix in lockstep, and the per-round F→P gate enforces the joint update. Section D.2 states full Algorithm 1.

4 Experiments

Setup. We evaluate on the 433-instance SWE-bench Verified ∩ SWT-bench Verified intersection using Resolved (fix passes the SWE-bench harness) and F→P (test fails on buggy code and passes on the golden fix). We compare fix-only and cogeneration baselines. CoHarden uses OpenHands with GPT-5-mini, $K = 5$ hardening rounds, and four mutation operators with $N = 3$ samples each. Full setup details and supplementary evaluations are in Sections C and E.

4.1 CoHarden Improves Fix Correctness over Existing Pipelines

CoHarden achieves 69.4% Resolved (Table 2), exceeding SWE-Agent, the strongest fix-only baseline, by 9.6 points and InfCode, the strongest cogeneration baseline, by 7.9 points. The gain over vanilla OpenHands + CoGen is 8.0 points. In a separate round-matched experiment, Ralph-Loop reaches 63.8% Resolved at $K = 5$, versus 69.4% for CoHarden (Section E.5 and table 15). This compares complete refinement procedures, including their initialization, rather than isolating mutation feedback. CoHarden costs $0.84 per instance in the final run, including $0.079 for mutation generation; equal round budgets do not imply equal expenditure.

The R3 checkpoint reaches 68.3% Resolved at an accumulated cost of $0.696 per instance, compared with 69.4% and $0.834 at R5 (Table 16). Mean wall-clock time is 22.0 minutes and the 95th percentile

is 49.2 minutes (Table 17). Mutation generation accounts for 9.5% of API cost, not the complete overhead of mutant execution or infrastructure.

Table 2: Resolved, F→P, and average per-instance cost on the 433-instance SWE-bench Verified ∩ SWT-bench Verified. All methods share the same GPT-5-mini backbone. "–" denotes unreported values. COHARDEN outperforms fix-only and cogeneration baselines on Resolved, suggesting that iterative hardening improves fix quality beyond simply increasing F→P coverage.

Method	Resolved (%)	Test F→P (%)	Avg. cost (USD/inst)
FIX-ONLY *(no reproduction test)*			
SWE-Agent (Yang et al., 2024)	59.8	–	0.10
mini SWE-Agent (Yang et al., 2024)	55.3	–	0.17
Agentless (Xia et al., 2025)	46.5	–	0.24
OpenHands (Wang et al., 2025)	58.9	–	0.39
Cogeneration *(fix and BRT produced jointly)*			
InfCode (Li et al., 2025)	61.5	69.6	0.77
Agent-CoEvo (Li et al., 2026b)	58.2	64.9	0.88
OpenHands + COGEN (Wang et al., 2025)	61.4	72.7	0.56
COHARDEN (Ours)	**69.4**	**78.9**	0.84

4.2 COHARDEN GENERALIZES ACROSS LLM BACKBONES AND BENCHMARKS

Table 3: Cogeneration methods across LLM backbones (GPT-5.4 and Claude Opus 4.5) on the 433-instance SWE-bench Verified ∩ SWT-bench Verified: InfCode (Li et al., 2025), Agent-CoEvo (Li et al., 2026b), and OpenHands + COGEN (Wang et al., 2025). COHARDEN leads under both backbones, with up to +4.0 Resolved gain over the strongest baseline.

	GPT-5.4			Claude Opus 4.5		
Method	Resolved (%)	F→P (%)	Cost ($)	Resolved (%)	F→P (%)	Cost ($)
InfCode	73.2	76.6	7.25	74.4	71.3	26.91
Agent-CoEvo	73.5	75.6	8.79	74.7	71.8	28.43
OpenHands + COGEN	70.0	75.3	4.26	70.8	71.5	13.99
COHARDEN (Ours)	**76.2**	**76.7**	6.69	**78.7**	**79.0**	18.07

COHARDEN leads the cross-backbone comparison (Table 3), with Resolved gains of 2.7 and 4.0 points over the strongest baseline. On SWE-bench Lite (Table 4), its 57.6% exceeds InfCode by 3.4 points. The cross-backbone table also reports average per-instance costs. Section F.1 summarizes the benchmark and backbone distinctions.

Table 4: Cogeneration methods on the 276-instance SWE-bench Lite ∩ SWT-bench Lite with GPT-5-mini. Baselines: OpenHands + COGEN (Cheng et al., 2026), Agent-CoEvo (Li et al., 2026b), and InfCode (Li et al., 2025). COHARDEN achieves the highest Resolved (57.6%) and Test F→P (64.1%), showing transfer to a different benchmark.

Method	Resolved	Test F→P
OpenHands + COGEN	52.0	62.3
Agent-CoEvo	51.0	53.6
InfCode	54.2	63.7
COHARDEN	**57.6**	**64.1**

Table 5: Phase ablation on 433 Verified instances. Each variant removes lax-init, hardening, or both from the full COHARDEN pipeline. Lax-init and hardening each address a distinct cogeneration weakness; either alone recovers little of the full gain. Three-seed results on 60 instances are reported separately (Table 13).

Variant	Resolved (%)	F→P (%)
COHARDEN	**69.4**	78.9
w/o lax-init	62.4	70.2
w/o hardening	61.0	**82.6**
w/o both	61.4	72.7

4.3 Effects of Lax-init and Hardening

In the phase ablation (Table 5), adding lax-init improves F→P by 8.7 points both with and without hardening. Resolved improves by 7.0 points when hardening is present, but changes from 61.4% to 61.0% without it. Thus the evidence supports complementary roles, not an independent repair gain from lax-init alone. Figure 4 shows Resolved reaching 68.3% by R3 and 69.4% at R5 as F→P declines, so most gains occur early without increasing F→P coverage. The separate three-seed, 60-instance ablation also reports higher mean Resolved for CoHarden than vanilla CoGen ($68.1 \pm 0.7\%$ versus $63.8 \pm 1.7\%$; Table 13). Its mean ± SD summaries (Section E.3) are not full-benchmark confidence intervals; Section F.2 distinguishes the two evaluations.

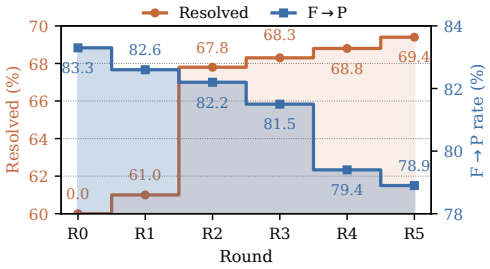

Figure 4: Per-round evaluation of Resolved and F→P on 433 Verified instances. The orange Resolved curve uses the left axis; the blue F→P curve uses the right axis. Resolved rises steeply in the first three rounds and saturates after R3; F→P declines steadily as hardening tightens the test. Resolved reaches 68.3% at R3 and 69.4% at R5. R0 is test-only; its displayed zero is not a repair result.

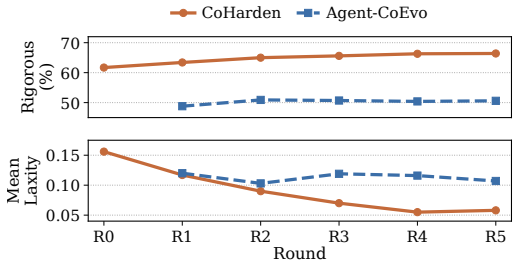

Figure 5: MPE trajectories for CoHarden and Agent-CoEvo on 433 Verified instances, scored with the same mutant pools across methods and rounds. The upper panel shows the share classified as Rigorous; the lower shows mean Laxity β/total, the fraction of mutants accepted by the generated test but rejected by the golden test. Higher Rigorous share and lower Laxity indicate stronger discrimination under this pool, not guaranteed correctness.

4.4 Test Quality and Sensitivity

Under a shared, method-independent mutant pool (Figure 5), CoHarden's Rigorous share rises from 61.7% to 66.4% and Laxity falls from 0.156 to 0.058; Agent-CoEvo plateaus near 51% Rigorous. Improvement is not monotonic, and Misaligned outputs rise from 18.5% to 24.7%. Full distributions and the separate shared-subset comparison are in Section E.1 and tables 8 and 9. On the latter subset, CoHarden has 68.8% Rigorous tests, versus 49.5–51.5% for the three cogeneration baselines.

Changing the mutant generator flips 19.7–29.7% of Rigorous/Lax labels (Table 10), but the repair-utility gap remains positive at 4.45–5.98 points, versus 8.51 for GPT-5-mini (Table 11). The gap also remains positive at every tested Laxity cutoff in $[0.05, 0.60]$ (Table 12), rather than only at the median split. Full sensitivity sweeps are in Section E.2. The corpus audit (Section E.4 and table 14) supports executability and structural diversity, not semantic realism or invariant labels.

5 Related Work

5.1 Generating and Evaluating Bug Reproduction Tests

Prior work on bug reproduction tests (BRTs) has focused mainly on generating tests from issue descriptions and measuring whether they reproduce the bug. Most methods improve generation success through issue-driven or repository-aware construction (Issue2Test (Nashid et al., 2026), Otter (Ahmed et al., 2025)), retrieval and execution feedback (Echo (Fei et al., 2026)), model specialization (SWE-tester (Soni et al., 2026)), or test inversion from passing tests (AssertFlip (Khatib et al., 2026)). Related studies ask whether LLM-generated tests improve downstream repair (Zhang et al., 2026) or analyze properties of developer-written bug-reproducing tests (Hora & Fraser, 2026). TestExplora (Liu et al., 2026) instead studies proactive bug discovery through repository-level test

generation. Our work focuses on whether generated BRTs constrain plausible fixes, using the rigorous, lax, and misaligned categories to motivate test refinement during repair.

5.2 Test-Fix Co-Evolution

Another line of work treats tests and fixes as coupled search objects and improves them jointly. Dynamic Cogeneration (Cheng et al., 2026) studies different loop orderings, while Agent-CoEvo (Li et al., 2026b) and InfCode (Li et al., 2025) iteratively refine tests and fixes through mutual or adversarial feedback. CoCoEvo (Li et al., 2026c) and LLMLOOP (Ravi et al., 2025) explore related designs in function-level settings. These works motivate joint refinement, but do not use our distinction between rigorous, lax, and misaligned tests to guide repair. Our work targets the resulting test-fix coupling and repair failures.

5.3 Overfitting and Variant-Based Test Analysis

The broader repair literature has long shown that passing available tests is not enough for correctness. Work on patch plausibility (Qi et al., 2015) and behavioral divergence in SWE-bench-style evaluation (Wang et al., 2026) highlights the limits of weak tests and motivates evaluating how well tests constrain variants. SWE-ABS (Yu et al., 2026) uses adversarial analysis to expose inflated success rates under weak tests, SWE-Bench+ (Aleithan et al., 2024) reveals benchmark blind spots, and PROBE (Li et al., 2026a) uses surviving variants of reference patches to guide test augmentation. These benchmark-strengthening approaches differ from our reference-free repair loop. Earlier work also targets patch quality in automated repair (Soto, 2019); our focus is on distinguishing lax from misaligned BRTs and addressing their different effects on repair.

6 Discussion

Beyond F→P. F→P is a useful but coarse criterion for BRT generation; within the F→P-passing pool, repair gains concentrate in the Rigorous tier (Section 2.2). Offline MPE uses golden artifacts to measure this distinction, whereas the Temporal Matrix provides a reference-free comparison between successive tests inside the repair loop. The shared-pool evaluation tests whether improvements extend beyond each method's own feedback; the sensitivity sweeps test whether the repair-utility difference depends on a single labeling choice. Neither agreement on a finite mutant pool nor improvement relative to an earlier test establishes semantic correctness.

Later-round coupling. The follow-up failure analysis attributes 34 of 139 unresolved cases (24.5%) to later-round test–fix coupling. Another 44 cases have partial fixes despite tests classified as Rigorous by an independent post-hoc check (Section E.6 and table 18). Successive tests and fixes can drift together, leaving errors invisible to a relative comparison. Lax-init targets the initial condition; it does not resolve this later coupling or make a Rigorous test an exhaustive specification.

Limitations. Mutant coverage depends on the generator and chosen operators; runnable, structurally diverse edits need not represent realistic incorrect repairs. The sensitivity analysis concerns relative utility, not invariant labels or correctness. Evaluation is limited to Python repair benchmarks and three repair backbones, with repeated-run analysis confined to a 60-instance subset. Generalization beyond these settings remains untested (Section A).

7 Conclusion

We studied whether generated bug-reproducing tests (BRTs) actually advance downstream bug fixing. To answer this, we introduced *Mutation Patch Evaluation* (MPE), which classifies BRTs as Rigorous, Lax, or Misaligned. Our empirical study revealed two limitations of F→P-only quality control: the fix-side gain concentrates in the Rigorous tier, while cogeneration produces Lax tests with correlated fix errors. Building on these findings, we proposed CoHarden, a two-phase cogeneration loop that uses lax-init to avoid early misalignment and hardening with a Temporal Matrix to discourage Lax acceptance. CoHarden reaches 67.9% Resolved on SWE-bench Verified, outperforming the strongest fix-only and cogeneration baselines by $+8.1$ and $+6.4$ percentage points respectively, with consistent gains across LLM backbones and benchmarks.

AI USE STATEMENT

During the preparation of this manuscript, large language models (LLMs) assisted with manuscript drafting, language refinement, and LaTeX formatting. All LLM-assisted manuscript content was manually reviewed and validated by the authors. The research design, experimental analysis, and scientific conclusions were developed independently by the authors, who made all final scientific decisions. Separately, LLMs serve as experimental components for generating candidate patches, bug reproduction tests, and mutation variants, as described in Sections B.1 and D.2; the model backbones and computational environment are documented in Section D.3. The authors take full responsibility for the final content of this work, including all text, claims, and artifacts produced with AI assistance.

ETHICS STATEMENT

This study uses publicly available open-source software benchmarks and does not involve human participants or sensitive data. Generated code is evaluated in isolated benchmark environments. Generated patches and tests can share errors, so agreement between them does not guarantee correctness or safety. Automated repair tools should therefore be used with independent testing and human review before deployment. This work does not release pretrained models or newly scraped datasets.

REPRODUCIBILITY STATEMENT

The datasets and baseline configurations are described in Sections C.1 and C.2. Mutation operators, implementation details, and the computational environment are documented in Sections B.1, D.2 and D.3, and the full prompts are provided in Section G. Source code and evaluation scripts are available at `https://anonymous.4open.science/r/COHARDEN-9753`. Reproduction depends on access to the commercial LLM APIs used in the experiments.

REFERENCES

Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate SWE patches. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*, pp. 752–771. PMLR, 2025. URL https://proceedings.mlr.press/v267/ahmed25b.html.

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Heterogeneous prompting and execution feedback for SWE issue test generation and selection. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 1250–1261, 2026. doi: 10.1145/3744916.3787837. URL https://doi.org/10.1145/3744916.3787837.

Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. Swe-bench+: Enhanced coding benchmark for llms. *CoRR*, abs/2410.06992, 2024. doi: 10.48550/ARXIV.2410.06992. URL https://doi.org/10.48550/arXiv.2410.06992.

Anthropic. Claude Code. https://www.anthropic.com/claude-code, 2025. Accessed 2025.

Runxiang Cheng, Michele Tufano, José Cambronero, Renyao Wei, Sherry Shi, Grant Uy, Pat Rondon, and Franjo Ivančić. Dynamic cogeneration of bug reproduction test in agentic program repair. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*, pp. 644–654. ACM, 2026. doi: 10.1145/3803437.3805237. URL https://doi.org/10.1145/3803437.3805237.

Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978. doi: 10.1109/C-M.1978.218136. URL https://doi.org/10.1109/C-M.1978.218136.

Zhiwei Fei, Yue Pan, Federica Sarro, Jidong Ge, Marc Liu, Vincent Ng, and He Ye. Echo: Graph-enhanced retrieval and execution feedback for issue reproduction test generation. *CoRR*, abs/2603.07326, 2026. doi: 10.48550/ARXIV.2603.07326. URL https://doi.org/10.48550/arXiv.2603.07326.

GitHub. GitHub Copilot. https://github.com/features/copilot, 2025. Accessed 2025.

André C. Hora and Gordon Fraser. Understanding bug-reproducing tests: A first empirical study. *CoRR*, abs/2602.02965, 2026. doi: 10.48550/ARXIV.2602.02965. URL https://doi.org/10.48550/arXiv.2602.02965.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011. doi: 10.1109/TSE.2010.62. URL https://doi.org/10.1109/TSE.2010.62.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pp. 2312–2323, 2023. doi: 10.1109/ICSE48619.2023.00194. URL https://doi.org/10.1109/ICSE48619.2023.00194.

Lara Khatib, Noble Saji Mathews, and Meiyappan Nagappan. AssertFlip: Reproducing bugs via inversion of LLM-generated passing tests. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 3825–3835, 2026. doi: 10.1145/3744916.3773127. URL https://doi.org/10.1145/3744916.3773127.

Chenglin Li, Yisen Xu, Zehao Wang, Shin Hwei Tan, and Tse-Hsun (Peter) Chen. Probe to generate: Program variant-guided test augmentation for repository-level repair benchmarks. *arXiv preprint arXiv:2604.01518*, 2026a. doi: 10.48550/arXiv.2604.01518. URL https://arxiv.org/abs/2604.01518v2.

KeFan Li, Mengfei Wang, Hengzhi Zhang, Zhichao Li, Yuan Yuan, Mu Li, Xiang Gao, Hailong Sun, Chunming Hu, and Weifeng Lv. Infcode: Adversarial iterative refinement of tests and patches for reliable software issue resolution. *CoRR*, abs/2511.16004, 2025. doi: 10.48550/ARXIV.2511.16004. URL https://doi.org/10.48550/arXiv.2511.16004.

Kefan Li, Yuan Yuan, Mengfei Wang, Shihao Zheng, Wei Wang, Ping Yang, Mu Li, and Weifeng Lv. Beyond fixed tests: Repository-level issue resolution as coevolution of code and behavioral constraints. *arXiv preprint arXiv:2604.04580*, 2026b. doi: 10.48550/arXiv.2604.04580. URL https://arxiv.org/abs/2604.04580.

Kefan Li, Yuan Yuan, Hongyue Yu, Tingyu Guo, and Shijie Cao. CoCoEvo: Co-evolution of programs and test cases to enhance code generation. *IEEE Transactions on Evolutionary Computation*, 30(4): 1420–1434, 2026c. doi: 10.1109/TEVC.2025.3593272. URL https://doi.org/10.1109/TEVC.2025.3593272.

Steven Liu, Jane Luo, Xin Zhang, Aofan Liu, Hao Liu, Jie Wu, Ziyang Huang, Yangyu Huang, Yu Kang, and Scarlett Li. Testexplora: Benchmarking llms for proactive bug discovery via repository-level test generation. *CoRR*, abs/2602.10471, 2026. doi: 10.48550/ARXIV.2602.10471. URL https://doi.org/10.48550/arXiv.2602.10471.

LogicStar AI. LogicStar on test generation benchmark SWT-Bench Verified: Best test generation at 84%. LogicStar AI Blog, September 2025. https://logicstar.ai/blog/logicstar-on-test-generation-benchmark-swt, accessed 2026-04-27.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin T. Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/94f093b41fc2666376fb1f667fe282f3-Abstract-Conference.html.

Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. Issue2Test: Generating reproducing test cases from issue reports. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 754–766, 2026. doi: 10.1145/3744916.3773129. URL https://doi.org/10.1145/3744916.3773129.

Zichao Qi, Fan Long, Sara Achour, and Martin C. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In Michal Young and Tao Xie (eds.), *Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015*, pp. 24–36. ACM, 2015. doi: 10.1145/2771783.2771791. URL https://doi.org/10.1145/2771783.2771791.

Ravin Ravi, Dylan Bradshaw, Stefano Ruberto, Gunel Jahangirova, and Valerio Terragni. LLMLOOP: improving llm-generated code and tests through automated iterative feedback loops. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2025, Auckland, New Zealand, September 7-12, 2025*, pp. 930–934. IEEE, 2025. doi: 10.1109/ICSME64153.2025.00109. URL https://doi.org/10.1109/ICSME64153.2025.00109.

Aditya Bharat Soni, Rajat Ghosh, Vaishnavi Bhargava, Valerie Chen, and Debojyoti Dutta. Swe-tester: Training open-source llms for issue reproduction in real-world repositories. *CoRR*, abs/2601.13713, 2026. doi: 10.48550/ARXIV.2601.13713. URL https://doi.org/10.48550/arXiv.2601.13713.

Mauricio Soto. Improving patch quality by enhancing key components of automatic program repair. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pp. 1230–1233. IEEE, 2019. doi: 10.1109/ASE.2019.00147. URL https://doi.org/10.1109/ASE.2019.00147.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, et al. Openhands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

You Wang, Michael Pradel, and Zhongxin Liu. Are "solved issues" in SWE-bench really solved correctly? an empirical study. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 169–181, 2026. doi: 10.1145/3744916.3764576. URL https://doi.org/10.1145/3744916.3764576.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pp. 1482–1494. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00129. URL https://doi.org/10.1109/ICSE48619.2023.00129.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying llm-based software engineering agents. *Proc. ACM Softw. Eng.*, 2(FSE):801–824, 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

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 Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/5a7c947568c1b1328ccc5230172e1e7c-Abstract-Conference.html.

Boxi Yu, Yang Cao, Yuzhong Zhang, Liting Lin, Junjielong Xu, Zhiqing Zhong, Qinghua Xu, Guancheng Wang, Jialun Cao, Shing-Chi Cheung, Pinjia He, and Lionel C. Briand. SWE-ABS: adversarial benchmark strengthening exposes inflated success rates on test-based benchmark. *CoRR*, abs/2603.00520, 2026. doi: 10.48550/ARXIV.2603.00520. URL https://doi.org/10.48550/arXiv.2603.00520.

Chengming Zhang, Haoye Wang, Chuyang Xu, Jiakun Liu, Kui Liu, and Zhongxin Liu. Can test cases generated by large language models facilitate automated program repair? *Empir. Softw. Eng.*, 31(3):68, 2026. doi: 10.1007/S10664-026-10802-W. URL https://doi.org/10.1007/s10664-026-10802-w.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In Maria Christakis and Michael Pradel (eds.), *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2024, Vienna, Austria, September 16-20, 2024*, pp. 1592–1604. ACM, 2024. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

A LIMITATIONS

LLM-synthesized mutants. MPE and the Temporal Matrix rely on LLM-synthesized mutants of c_{golden} or c_k, so their coverage inherits the generator's notion of plausible-but-incorrect fixes. The four targeted operators address common Lax patterns, but rare or domain-specific patterns may go undetected. The follow-up audit establishes executable yield and changed-line diversity, not semantic realism (Section E.4). Generator changes alter individual labels even when the Rigorous-minus-Lax utility gap remains positive. Thus Rigorous denotes a relative classification under a specified pool and threshold, not a guarantee of test adequacy.

Scope. Our repair experiments cover Python bug-fix benchmarks (SWE-bench Verified and SWE-bench Lite) with GPT-5-mini, GPT-5.4, and Claude Opus 4.5. The alternative mutation generators in the sensitivity study are not additional repair backbones. Generalization to other languages, repair settings, or non-agentic LLM systems remains untested.

Statistical coverage. The primary full-benchmark results are single-run evaluations. The repeatability study reports three stochastic runs on a separate 60-instance subset (Section E.3). It does not provide confidence intervals for the full benchmark or establish that every pairwise difference is statistically significant.

B EMPIRICAL STUDY SETUP

This appendix gives implementation details for the empirical study of Section 2: the mutation patch pool used as the evaluation instrument (Section B.1), and the per-setting setup details for the standalone (Section B.2) and cogeneration (Section B.3) studies.

B.1 MUTATION PATCH EVALUATION: OPERATORS AND PROMPTS

This appendix details the mutation patch pool used as the evaluation instrument in Section 2.2. Given a generated test t_{gen}, we construct a pool $\mathcal{C} = \{c_1, \ldots, c_K\}$ of plausible-but-incorrect repairs, and use the joint pass/fail outcomes of t_{gen} and t_{golden} on $\mathcal{C}$ to populate the 2×2 confusion matrix that classifies t_{gen} via MPE's three-way partition.

Operators. Each mutation patch is generated under one of five *semantic operators*, targeting common ways an incorrect fix can pass a reproduction test. Figure 6 illustrates one concrete realization for each of the four targeted operators.

- **Vanilla.** A free-form repair attempt with no operator-specific guidance, used as a reference distribution of unconstrained mutation patches.
- **Symptom suppression.** Targets tests whose detection signal is an exception or warning. The fix does not address the underlying bug; instead, it suppresses, swallows, or replaces the signal the test inspects. Typical realizations: wrap the buggy region in a `try/except` that swallows the exception; suppress warnings via `warnings.catch_warnings()`; silence error output; catch the bug's exception and re-raise a different (expected) type.
- **Incomplete fix.** Targets tests that exercise only a strict subset of the bug-affected paths (one type, one file, one pipeline stage, one caller). Applying the golden fix only to that subset is enough to pass the test, while the rest of the buggy paths remain unfixed. Typical realizations: when c_{golden} modifies multiple locations, edit only one; when c_{golden} handles multiple conditions or types, add an `if` guard that restricts the fix to one of them; when c_{golden} touches multiple stages of a pipeline, only repair one stage.
- **Input-specific shortcut.** Targets tests that pin a specific input shape, type, or value. A precomputed shortcut or guard for that input lets the test pass while the buggy region of the code remains untouched. Typical realizations: an early return for the test's specific input shape, type, or value; a guard clause that routes the test inputs to a clean code path; a cache or memoization layer that returns a precomputed result; a special-case branch inserted before the buggy region.
- **Behavior substitution.** Targets tests that check only partial output properties (e.g., output differs from the buggy string, type, shape, or finiteness). Replacing the golden fix's core expression or

(a) Symptom suppression

```python
def f(x):
    with warnings.catch_warnings():
        warnings.simplefilter("ignore")
        return buggy_call(x)  # still wrong, just quiet
```

(b) Incomplete fix

```python
def pipeline(x):
    x = parse_fixed(x)            # only parsing is fixed
    x = normalize_buggy(x)        # normalization still buggy
    return serialize_buggy(x)
```

(c) Input-specific shortcut

```python
def f(x):
    if x == TEST_INPUT:           # exactly the failing case
        return PRECOMPUTED_OUTPUT
    return buggy_call(x)          # other inputs still buggy
```

(d) Behavior substitution

```python
def f(a, b):
    # gold_fix uses strict '>' to exclude equality;
    if a >= b:
        return handle(a, b)
    return default()
```

Figure 6: Concrete examples of the four targeted MPE operators. **(a) Symptom suppression**: wrapping the buggy call in a warning-suppression context lets the bug-introduced warning disappear from the test's view. **(b) Incomplete fix**: a multi-stage pipeline is repaired only at the parsing stage, leaving normalization and serialization buggy. **(c) Input-specific shortcut**: an early branch returns a precomputed answer for the test's specific input, bypassing the buggy code path. **(d) Behavior substitution**: a comparison operator is replaced (>= for >), satisfying tests that only check the boundary case qualitatively.

call with a syntactically valid alternative satisfies the partial check while the resulting behavior is still incorrect. Typical realizations: a different (e.g., narrower or platform-dependent) type cast; a different but similar API call; a different comparison operator (e.g., >= for >); a different ordering of operations.

For each instance we sample 56 patches per operator, yielding $|\mathcal{C}| = 280$ mutation patches per instance. Patches that fail to apply or fail to produce a runnable repository are discarded and resampled from the same operator until the budget is exhausted.

Patch generator. Mutation patches are generated using the Agentless pipeline (Xia et al., 2025) backed by GPT-5-mini. We prepend the operator-specific instruction to Agentless's fix-generation prompt and inject three additional context items: the golden fix c_{golden}, the golden test t_{golden}, and the generated test t_{gen}. The operator-specific instruction is the only signal that biases the pipeline toward a particular failure mode.

B.2 TDD-STYLE BRT INJECTION: SETUP DETAILS

We use SWT-Bench (Mündler et al., 2024) as the instance pool. For each of the four BRT generators considered—LogicStar (LogicStar AI, 2025), e-Otter++ (Ahmed et al., 2026), AssertFlip (Khatib et al., 2026), and OpenHands (BRT-ONLY mode) (Wang et al., 2025)—we download each generator's published BRT predictions from the prediction logs linked in their GitHub repositories. We then label the resulting (instance, BRT) pair as F→P or non-F→P according to whether the test fails on the buggy code and passes on the golden fix c_{golden}. Pairs on which the test harness errors out (timeouts, parser failures, non-importable test modules) are excluded; the remaining pairs form the two evaluation pools.

Run protocol. The fix agent is OpenHands (Wang et al., 2025) backed by GPT-5-mini-0807, with one trial per instance and a single shared no-test baseline (the same fix agent run on each instance with no BRT injected).

Per-agent pool sizes. Table 6 reports the per-agent pool sizes. The F→P pool aggregates to $1{,}104$ pairs and the non-F→P pool to 474 pairs. The four agents differ widely in F→P rate: LogicStar covers 87.1% of instances, while AssertFlip covers 45.7%; the non-F→P pool is correspondingly skewed toward the lower-coverage agents. This skew matters when interpreting the per-agent breakdown in Table 7, because F→P rate alone does not predict TDD utility—LogicStar combines the highest F→P rate with the largest TDD gain, whereas AssertFlip combines a much lower F→P rate with zero gain.

Table 6: Per-agent pool sizes for the TDD experiment on SWT-Bench. The F→P pool is the set of instance-agent pairs on which the BRT fails on the buggy code and passes on the golden fix c_{golden}; the non-F→P pool is the complementary set, after excluding pairs on which the test harness errors out.

Agent	F→P pairs	non-F→P pairs	F→P rate
LogicStar	377	53	87.1%
e-Otter++	272	154	62.8%
AssertFlip	198	127	45.7%
OpenHands-GPT5mini	257	140	59.4%
Total	1,104	474	63.7%

Table 7: Per-method breakdown of the RQ1 TDD experiment under the three-way taxonomy of generated BRTs. *Overall* TDD *Resolved* is the fraction of the method's instances on which the OpenHands+GPT-5-mini fix agent, guided by the injected BRT, produces a fix passing the gold tests. *Rigorous* and *Lax* partition the method's F→P pool by the median of β/total ($= 0.202$); *Misaligned* collects its non-F→P pairs. Methods are listed in decreasing order of F→P rate (per Table 6).

Test class	# BRT	FIX-ONLY	TDD	Δ	Δ_R
LogicStar (LogicStar AI, 2025)		*Overall* TDD *Resolved* $= 65.8\%$			
Rigorous	196	61.7%	75.5%	+13.8	+36%
Lax	181	68.9%	69.1%	+0.2	+1%
Misaligned	53	28.3%	18.9%	−9.4	−13%
e-Otter++ (Ahmed et al., 2026)		*Overall* TDD *Resolved* $= 60.3\%$			
Rigorous	156	69.5%	75.9%	+6.4	+21%
Lax	116	69.1%	65.7%	−3.4	−11%
Misaligned	154	45.5%	41.6%	−3.9	−7%
OpenHands (Wang et al., 2025)		*Overall* TDD *Resolved* $= 61.0\%$			
Rigorous	123	71.2%	76.1%	+4.9	+17%
Lax	134	72.5%	74.7%	+2.2	+8%
Misaligned	140	40.0%	40.7%	+0.7	+1%
AssertFlip (Khatib et al., 2026)		*Overall* TDD *Resolved* $= 58.9\%$			
Rigorous	77	71.1%	76.3%	+5.2	+18%
Lax	121	70.0%	66.7%	−3.3	−11%
Misaligned	127	55.1%	42.5%	−12.6	−28%

B.3 COGENERATION SETTING: SETUP DETAILS

Cogeneration instantiation. Our COGEN configuration follows the joint fix-and-test design of Cheng et al. (2026), which has no official SWT-Bench implementation; we implement it by adapting the OpenHands (Wang et al., 2025) prompt template to also emit a reproduction test in the same trajectory. The configuration uses GPT-5-mini-0807 as the underlying language model with one trial per instance on SWT-Bench.

Comparison configurations. In COGEN mode, the agent emits both a fix and a test in a single trajectory. In BRT-ONLY mode, the same agent is asked only for a reproduction test (fix emission disabled). In FIX-ONLY mode, the agent is asked only for a fix (test emission disabled). The three configurations share the same backbone, prompt scaffold, and inference settings; they differ only in the artifacts requested from the agent within the trajectory.

Joint-failure baseline. Let p_{fix} be the marginal rate of unresolved fixes ($166/433$) and p_{test} the marginal rate of non-F→P tests ($120/433$). Under independent fix and test errors, the joint-failure rate is $p_{\text{fix}} \cdot p_{\text{test}} = 10.6\%$; the observed joint-failure rate is $86/433 = 19.9\%$, giving a coupling factor of 1.87.

C EXPERIMENTAL SETUP DETAILS

Benchmark. We use the 433-instance intersection of SWE-bench Verified and SWT-bench Verified, which enables joint evaluation of patch correctness and test effectiveness. To assess cross-benchmark generalization, we additionally evaluate on SWE-bench Lite. Dataset construction, filtering criteria, and statistics are detailed in Section C.1.

Metrics. Resolved measures the fraction of instances where the generated patch passes the SWE-bench evaluation harness. **F→P** measures the fraction of instances where the generated test fails on the buggy program and passes on the corresponding reference fix, following the SWT-bench protocol. Together, these metrics capture complementary aspects of correctness (fix validity vs. test effectiveness).

Baselines. We compare against two families: (i) *fix-only* agents that generate patches without explicit reproduction tests, and (ii) *cogeneration* methods that jointly produce patches and tests in a single trajectory. The full list of baselines and their configurations are provided in Table 2, with detailed descriptions in Section C.2.

Implementation. We implement COHARDEN on top of OpenHands with GPT-5-mini as the backbone LLM. The hardening procedure runs for $K = 5$ rounds. In each round, we construct a mutant pool $\mathcal{C}_k$ using the four targeted mutation operators described in Section 2.1, with $N = 3$ samples per operator (up to $|\mathcal{C}_k| = 12$). All OpenHands-based methods share the same backbone to ensure fair comparison. For non-OpenHands baselines, we report results from their official implementations. Additional implementation details, including prompts and per-round procedures, are provided in Section D.2.

C.1 DATASET DETAILS

SWE-bench. SWE-bench (Jimenez et al., 2024) collects real-world GitHub issues paired with developer-written patches (*golden fixes*) from 12 popular open-source Python repositories (Django, scikit-learn, sympy, matplotlib, etc.). SWE-bench Verified is a 500-instance human-validated subset whose golden fixes are confirmed correct and self-contained. SWE-bench Lite is a separate 300-instance subset curated for faster evaluation.

SWT-bench. SWT-bench (Mündler et al., 2024) re-purposes the same issue–patch instances but targets *test generation*: each instance is evaluated against a golden fail-to-pass test extracted from the original pull request. Because not every SWE-bench instance has a test patch suitable for fail-to-pass evaluation, SWT-bench filters out a small number of incompatible cases from each SWE-bench subset, 67 from Verified and 24 from Lite, yielding 433 and 276 usable instances respectively.

Evaluation sets. Our main experiments (Sections 4.1 and 4.3) use the 433-instance SWE-bench Verified ∩ SWT-bench Verified subset, where every instance possesses both a golden fix and a golden test, enabling joint evaluation of fix correctness (Resolved) and test quality (F→P). The cross-benchmark experiment (Section 4.2) uses the 276-instance SWE-bench Lite ∩ SWT-bench Lite subset.

C.2 BASELINE IMPLEMENTATION DETAILS

We compare COHARDEN against two families of methods. **(i) Fix-only**: agents that produce a patch with no explicit reproduction test, including SWE-Agent and mini SWE-Agent (Yang et al., 2024), Agentless (Xia et al., 2025), and OpenHands (Wang et al., 2025). **(ii) Cogeneration**: methods that jointly produce the fix and the BRT, including InfCode (Li et al., 2025), Agent-CoEvo (Li et al., 2026b), and OpenHands + COGEN (same backbone as COHARDEN). Below we describe each method in detail.

Fix-only. Fix-only methods produce a patch without generating an explicit bug-reproduction test. The fix is evaluated directly against the golden test suite.

- **SWE-Agent / mini SWE-Agent Yang et al. (2024).** SWE-Agent equips an LLM with a custom Agent–Computer Interface (ACI) that exposes repository navigation, file viewing, and editing as LM-friendly commands. The ACI abstractions (e.g., a scrollable file viewer,

a targeted edit command with lint feedback) substantially improve the agent's ability to resolve software-engineering issues end-to-end. mini SWE-Agent is a simplified variant ($\sim$100 lines) of SWE-Agent from the same team, designed for minimal complexity while retaining competitive performance.

- **Agentless Xia et al. (2025).** Agentless follows a non-agentic, three-phase pipeline: (1) hierarchical fault localization that narrows the search from file to class/function to fine-grained edit location, (2) LLM-based patch generation via sampling, and (3) patch selection using regression tests and LLM-generated reproduction tests. Unlike agent-based methods, the LLM does not decide future actions or wield interactive tools; it is queried in a fixed, structured sequence.
- **OpenHands Wang et al. (2025).** OpenHands is a composable software-agent platform that provides sandboxed runtime environments, an event-stream architecture, and modular agent components for planning, memory, and skill execution. In fix-only mode the agent interactively browses the repository, localizes the bug, and applies a patch without generating a reproduction test.

Cogeneration. Cogeneration baselines produce the fix and the reproduction test within a single trajectory, allowing mutual feedback between the two artifacts.

- **InfCode Li et al. (2025).** InfCode is a multi-agent adversarial framework comprising a Test Patch Generator, a Code Patch Generator, and a Selector agent. The test agent generates tests that expose failures, the code agent refines patches to pass them, and the Selector picks the best fix. This adversarial co-refinement iterates inside a containerized repository environment. Results are obtained from the official open-source implementation.
- **Agent-CoEvo Li et al. (2026b).** Agent-CoEvo frames issue resolution as coevolution: candidate code patches and test patches are jointly explored by multiple agents, with each side mutually evaluating and semantically recombining the other's candidates across iterations. Tests are treated as dynamic behavioral constraints that evolve alongside the code rather than fixed oracles. Since no official implementation is available, we reproduce it on top of the InfCode codebase from the same authors.
- **OpenHands + COGEN.** We implement the joint fix-and-test workflow studied by Cheng et al. (2026) on OpenHands. Their original study uses Google's Passerine system; our OpenHands implementation is a separate adaptation. Our agent produces both artifacts within one trajectory and uses the co-generated test as an in-trajectory F→P gate.

D METHOD IMPLEMENTATION

D.1 PLUGGABLECRITIC FOR FINE-GRAINED LOOP CONTROL

Background. We build on three mechanisms that today's LLM-agent stacks expose in isolation. A *Ralph-style loop* is the simplest possible autonomous-agent driver: a coding tool such as Claude Code (Anthropic, 2025), GitHub Copilot (GitHub, 2025), or OpenHands (Wang et al., 2025) is re-invoked on the same task until it declares completion, in spirit

```
while :; do cat PROMPT.md | claude-code --continue; done
```

A *skill* is a self-contained, reusable, pluggable unit of domain knowledge that recent SDKs (Claude Code Skills, GitHub Copilot's skill packs, the OpenHands skill catalog) standardize: a skill bundles prompts, reference documents, and small tools that the agent can read or invoke during a turn. A *critic* is an agent-internal scoring component that inspects what the agent has done and decides whether the submission is acceptable or whether the agent should retry.

Where each primitive falls short. Composing these three primitives to produce a domain-controlled loop runs into four gaps. **1.** Loops driven by Ralph-style scripts (or any external orchestrator) only see what the agent prints to `stdout` and the process exit code; they cannot inspect the agent's trajectory (which tools were called, what intermediate artifacts were produced, whether the agent actually ran the test) and so cannot make behavior-level decisions. **2.** Skills are *passive* from the

loop's perspective: the agent decides when, or whether, to consult them, and there is no surface to enforce that a skill's protocol was actually followed before the loop accepts a submission. **3.** There is no community-standard verify-and-feedback interface; each project hand-rolls its own retry harness, verification logic gets entangled with task logic, and good verifiers cannot be packaged and shared the way skills can. **4.** Retry feedback in script-driven loops is largely static ("try again, more carefully"); there is no clean way to push trajectory-derived dynamic content (which files were touched, which mutated patches survived, what the per-operator counts are) back into the next turn.

PLUGGABLECRITIC. PLUGGABLECRITIC combines the pluggability of a skill with the trajectory access of a critic, and exposes the combination as a user-defined verify-and-feedback boundary at the agent's `finish` action. On `finish`, the harness invokes the user-supplied critic, which inspects the agent's full event stream (tool calls, observations, messages) plus the working-tree diff and returns a score in $[0, 1]$. If the score falls below a configured threshold, the harness injects the critic's followup prompt as the next user turn and lets the agent retry, up to a configured maximum. This addresses gap 1 (the critic receives the full event stream and the working-tree diff), gap 2 (the critic can require arbitrary post-conditions before the loop accepts), gap 3 (the critic itself ships as a swappable unit, configured per experiment), and gap 4 (the followup prompt is composed by the critic and can carry trajectory-derived evidence).

Instantiation. A user instantiates a PLUGGABLECRITIC in one of two complementary modes. The *subclass mode* requires implementing one method on the abstract base `CriticBase`: `evaluate(events, git_patch) -> CriticResult`, where `CriticResult` carries a score, a message, and a free-form metadata dictionary; `get_followup_prompt(result, iteration) -> str` is optionally overridden to compose dynamic feedback from that metadata (the default returns a generic "please try again" message). The *skill mode*, used by COHARDEN, instead loads the critic declaratively from a skill directory: a `critic.yaml` manifest lists named signals (each typed as `hard_gate` or `soft` with a weight) together with their detector file, and each detector is a small Python module exposing `detect(events, git_patch) -> (triggered, metadata)`. The harness short-circuits on the first triggered hard gate and otherwise sums the soft-signal weights against the threshold; the default followup prompt walks the triggered signals and concatenates each detector's `metadata["followup_text"]`. The `git_patch` argument is the agent's working-tree diff at `finish` time, computed by the harness so detectors do not each have to recompute it. Two implementation choices support COHARDEN specifically: detectors run in-process via `importlib` so they share the SDK's LLM client (variant generation and survivor analysis would otherwise pay a serialization cost per round), and detectors own their feedback text (sampled survivor patches and per-operator counts) so dynamic content flows back to the agent without an SDK-side templating layer. The current implementation is coupled to the OpenHands SDK's event and tool model; the same design admits a small adapter layer for any other skill-based SDK that exposes a `finish`-equivalent boundary.

Reusability and release. The contract is intentionally minimal, so the same harness hosts critics that have nothing to do with COHARDEN: a clean-patch gate (rejects non-test files in test-only phases), a structural-validity gate (rejects unparseable diffs), and the test-only enforcement gate used by the BRT-ONLY baseline of Section 2.3 all ship as separate single-detector skills without code changes to the harness.

D.2 COHARDEN IMPLEMENTATION NOTES

Algorithm 1 states the full procedure. This appendix records implementation choices that support the algorithm but do not affect its algorithmic content. COHARDEN ships as a single-skill instantiation of Section D.1: `critic.yaml` declares one `hard_gate` signal whose detector implements Algorithm 1 end-to-end.

Algorithm 1 CoHarden: iterative-hardening cogeneration.

Require: issue I; max rounds K; mutation operators $\mathcal{O}$; samples per operator N.
Ensure: reproduction test $t^\star$, fix $c^\star$.

▷ *Phase 0 (Round 0): lax-init*
$t_0 \leftarrow \text{AGENT}_{\text{test-only}}(I)$
▷ *Phase 1 (Rounds 1..K): hardening*
$c_0 \leftarrow \emptyset$; feedback $\leftarrow \emptyset$
for $k = 1, \ldots, K$ **do**
  $(t_k, c_k) \leftarrow \text{AGENT}_{\text{joint}}(I,\ t_{k-1},\ c_{k-1},\ \text{feedback})$
  $\mathcal{C}_k \leftarrow \bigcup_{o \in \mathcal{O}} \text{MUTATE}(c_k, o, N)$
  $\text{TM}_k \leftarrow \text{TEMPORALMATRIX}(t_k,\ t_{k-1},\ \mathcal{C}_k)$
  if $k > 1$ **and** $\beta(\text{TM}_k) > 0$ **then**
    return (t_{k-1}, c_{k-1}) ▷ *regression: terminate*
  feedback $\leftarrow \text{RENDER}(\text{TM}_k,\ \mathcal{C}_k)$
return (t_K, c_K)

Convergence. The loop terminates either at the round budget K, returning (t_K, c_K), or upon the first round whose Temporal Matrix has a non-empty Laxity cell ($\beta > 0$), returning the previous-round pair (t_{k-1}, c_{k-1}). The latter signals that the agent can no longer eliminate a Laxity cell against t_{k-1}. We use $K = 5$ as the default in Section 4.

Variant generation budget. At each round the loop calls the patch-mutation LLM with N samples per operator (default $N = 3$, total $|\mathcal{C}_k| = 12$ when all four targeted operators are active) in parallel. Variants that fail to apply, time out, or raise an environment error are excluded from $|\mathcal{C}_k|$ rather than counted as survivors. A round is marked *inconclusive* if zero admissible variants are produced; the regression check is skipped for that round and the inconclusive count is tracked separately.

State persistence across agent invocations. Algorithm 1 treats the loop as a single program, but in the deployed system each iteration of the `for` loop is a separate agent invocation: the agent calls `finish`, the critic evaluates, and on rejection the SDK injects the followup and lets the agent continue. A small JSON state object (t_{prev}, round counter, inconclusive streak, fix history for the previous-round survivor check) is persisted to disk between invocations and reset whenever the benchmark instance changes.

Sampling budget vs. post-hoc evaluation. The post-hoc evaluation in Section B.1 uses a much larger pool (56 patches per operator, $|\mathcal{C}| = 280$) because it is a one-shot offline measurement; the in-loop pool is deliberately small ($N = 3$) because its cost is paid per agent round and the loop amortizes the evaluation over K rounds.

D.3 Computational Resources and Software Environment

Hardware. All experiments were conducted on a single server running **Ubuntu 24.04.1 LTS**, equipped with four Intel Xeon E7-4830 v3 processors (12 cores per socket, 48 physical cores, 96 logical threads at 2.10 GHz) and 503 GB of RAM. No GPU was used: all workloads are LLM API calls, Docker-based test execution, and lightweight scripting.

Software environment. The orchestration codebase is written in **Python 3.13** managed via **uv**. Key dependencies include OpenHands SDK, SWE-agent, Agentless, and the OpenAI Python client (`openai ≥ 2.24`). Each SWE-bench instance runs inside an isolated Docker container with its own Conda environment, following the SWE-bench harness protocol.

LLM API models. All LLM calls are made through commercial APIs. The per-token pricing (per 1 M tokens) is: GPT-5-mini ($0.25 / $2.00 input / output), GPT-5.4 ($2.50 / $15.00), and Claude Opus 4.5 ($5.00 / $25.00). All cost estimates assume **zero prompt-cache hit rate** to ensure a fair, reproducible upper bound.

Table 8: Test-quality trajectories on the 433-instance benchmark. All shares are percentages; Laxity is β/total. The same evaluation pool is used across methods and rounds.

Method	Round	Rigorous	Lax	Misaligned	Mean Laxity
CoHarden	0	61.7	19.8	18.5	0.156
	1	63.4	14.4	22.2	0.117
	2	65.0	11.7	23.3	0.090
	3	65.6	10.4	24.0	0.070
	4	66.3	9.2	24.5	0.055
	5 / final	66.4	8.9	24.7	0.058
Agent-CoEvo	1	48.8	14.2	37.0	0.120
	2	50.9	13.6	35.5	0.103
	3	50.7	14.0	35.3	0.119
	4	50.4	14.5	35.1	0.116
	5 / final	50.6	14.3	35.1	0.107

Table 9: Final-test quality on the shared subset with outputs from all four methods. These percentages describe the common-output subset, not the full 433-instance benchmark. Lower mean Laxity indicates stronger discrimination under the shared mutant pool.

Method	Rigorous (%)	Lax (%)	Misaligned (%)	Mean Laxity
CoHarden	68.8	9.9	21.3	0.058
Agent-CoEvo	51.2	13.9	34.9	0.110
OpenHands + Cogen	51.5	12.7	35.8	0.093
InfCode	49.5	17.1	33.4	0.100

E Additional Evaluation

E.1 Test-quality trajectories and final outputs

The trajectory analysis evaluates successive tests using a shared, method-agnostic mutant pool rather than each method's own in-loop pool. The same pool is reused across compared methods and rounds. Rigorous, Lax, and Misaligned follow the taxonomy in Section 2.1; mean Laxity rate denotes the average of β/total, where lower values indicate fewer mutants accepted by the generated test but rejected by the golden test. This offline diagnostic uses golden artifacts and is distinct from the reference-free Temporal Matrix used during generation.

Table 8 reports the trajectory comparison on all 433 benchmark instances. CoHarden's Rigorous share increases overall from 61.7% to 66.4%, while Laxity decreases from 0.156 to 0.058. The final Laxity value is slightly higher than R4's 0.055, so improvement is not strictly monotonic. Misaligned outputs also increase from 18.5% to 24.7%, showing that greater discrimination among tests does not eliminate semantic drift. Agent-CoEvo's Rigorous share levels off near 51% after R2.

The four-method final-output comparison in Table 9 uses a shared subset for which all four methods' final tests were available. This is a separate evaluation population from the full-benchmark trajectory comparison. Within this subset, CoHarden has the highest Rigorous share and the lowest mean Laxity rate.

E.2 Sensitivity of the MPE classification

We vary the mutant-pool size, omit individual targeted operators, and change the mutant-generating LLM while holding the evaluated tests fixed. Table 10 reports the resulting changes in the Rigorous/Lax assignment. Pool subsampling produces fewer label changes as the retained fraction grows. Omitting one operator produces reported total flip rates between 2.4% and 5.3%. Generator changes have a larger effect, with 19.7% and 29.7% of labels changing under Kimi-K2.6 and DeepSeek-V4-Flash, respectively. These results do not support generator-invariant labels for individual tests.

Table 10: Sensitivity of Rigorous/Lax labels to the mutant pool. Directional rates and totals are percentages; rounded columns need not sum exactly. Larger retained pools produce fewer label changes, whereas changing the generator has a stronger effect.

Change from the original pool	Rigorous→Lax	Lax→Rigorous	Total
Retain 10% of mutants	4.13	4.34	8.47
Retain 25% of mutants	2.58	2.59	5.17
Retain 50% of mutants	1.52	1.32	2.84
Retain 75% of mutants	0.89	0.78	1.67
Drop symptom suppression	1.4	1.1	2.4
Drop incomplete fix	2.0	1.7	3.7
Drop input-specific shortcut	2.8	2.5	5.3
Drop behavior substitution	1.9	1.6	3.5
Use DeepSeek-V4-Flash	18.4	11.3	29.7
Use Kimi-K2.6	12.4	7.3	19.7

Table 11: Downstream repair utility after reclassifying tests with each mutant generator. All changes and gaps are in percentage points. Generator changes reduce, but do not reverse, the reported Rigorous-minus-Lax gap.

Mutant generator	Rigorous Δ	Lax Δ	Gap
GPT-5-mini	+8.51	+0.00	+8.51
DeepSeek-V4-Flash	+7.25	+1.27	+5.98
Kimi-K2.6	+5.49	+1.05	+4.45

We therefore also recompute the downstream repair-utility difference using each generator's own classification. In Table 11, Δ is the change in Resolved from injecting BRTs rather than using the no-BRT baseline. The Rigorous-minus-Lax gap remains positive for all three generators, although it falls from 8.51 percentage points for GPT-5-mini to 5.98 and 4.45 for the alternatives. The evidence supports a persistent relative utility difference, not an absolute claim that every test retains its label or that Lax tests can never help.

Table 12 varies an explicit cutoff τ on Laxity rather than forcing an equal-size split. The utility gap stays positive over $\tau \in [0.05, 0.60]$, including splits ranging from approximately 29% to 81% Rigorous. The cutoff sweep is distinct from the equal-size partition in Section 2.2: at $\tau = 0.202$, it yields 554 Rigorous and 550 Lax pairs. Both analyses support greater repair utility in the Rigorous group.

E.3 Repeated-run ablations

We repeat each of the four phase-ablation variants for three independent stochastic seeds on a fixed 60-instance subset. Table 13 reports the mean and standard deviation for this subset, separately

Table 12: Sensitivity to the explicit Laxity cutoff τ. Counts partition 1,104 F→P pairs. Repair changes and the Rigorous-minus-Lax gap are percentage points. Explicit cutoffs need not produce the equal-size groups used in the primary analysis.

τ	N_{Rigorous}	N_{Lax}	Δ_{Rigorous}	Δ_{Lax}	Gap
0.05	315	789	+8.57	+2.53	+6.04
0.10	403	701	+8.19	+2.00	+6.19
0.15	491	613	+8.55	+0.82	+7.74
0.202	554	550	+8.30	+0.18	+8.12
0.25	610	494	+7.21	+0.61	+6.61
0.30	678	426	+7.08	−0.23	+7.31
0.40	770	334	+5.97	+0.30	+5.67
0.50	844	260	+5.57	+0.00	+5.57
0.60	899	205	+5.12	+0.49	+4.63

Table 13: Three-seed ablation on the fixed 60-instance subset. Values are mean ± standard deviation in percent, not confidence intervals for the full benchmark.

Variant	Resolved (%)	Test F→P (%)
CoHarden	68.1 ± 0.7	73.4 ± 1.6
w/o hardening	64.0 ± 1.9	81.7 ± 3.4
w/o lax-init	63.6 ± 0.7	68.4 ± 4.3
w/o both / vanilla Cogen	63.8 ± 1.7	69.9 ± 2.9

Table 14: Reported structural audit of the canonical mutant corpus. Runnable yield and changed-line Jaccard similarity characterize executability and textual diversity, not semantic realism.

Measure	Value
Usable, runnable / raw attempts	326,041/446,200 (73.1%)
Within-operator near-duplicate redundancy	8.77%
Within-operator mean pairwise Jaccard	0.189
Cross-operator mean pairwise Jaccard	0.130
Cross-operator near-duplicate rate	1.05%

from the single-run, 433-instance evaluation. CoHarden has the highest mean Resolved and lower observed variation than vanilla Cogen. Three runs on a small subset do not establish a general variance reduction or a formal significance result, and their standard deviations are not uncertainty estimates for the full-benchmark scores.

E.4 Mutant-corpus audit

The corpus audit reports 326,041 usable, runnable mutants from 446,200 raw generation attempts (73.1%). For structural comparison, a patch p is represented by the set $L(p)$ of its whitespace-stripped added and removed lines, retaining the addition or deletion sign. Patch similarity is $J(p, q) = |L(p) \cap L(q)|/|L(p) \cup L(q)|$. Pairs with $J \geq 0.90$ are treated as near-duplicates. Within-operator comparisons are made inside instance/operator groups; cross-operator comparisons assess how similar different operators' edits are.

Table 14 indicates substantial executable yield and low cross-operator structural redundancy. These checks do not establish semantic inequivalence, plausibility, or representativeness of real incorrect repairs. In particular, textually different edits can induce the same behavior. The audit measures executability and structural diversity; it does not include a systematic manual assessment of semantic realism.

E.5 Round-matched control and execution costs

Ralph-Loop uses the same $K = 5$ round budget as hardening but supplies a generic "review and improve" instruction instead of mutation feedback. It starts from the vanilla Cogen prompt. Table 15 presents a separate round-budget control experiment; its vanilla Cogen run is distinct from the primary comparison in Table 2. CoHarden reaches 67.9% Resolved, versus 63.8% for Ralph-Loop. Equal round budgets do not establish equal token usage, expenditure, or wall-clock time. Since the initialization also differs, this experiment compares the complete refinement procedures rather than isolating mutation feedback from lax-init.

Table 16 reports accumulated costs at successive round checkpoints from the final 433-instance run. The final total is $0.834 per instance, of which $0.079 (9.5%) is attributed to mutation generation. The same final total is used in Table 2. This percentage covers generation charges, not the complete cost of mutant execution or infrastructure. Checkpoint totals are cumulative and are not summed across rounds.

The final run has a mean wall-clock duration of 22.0 minutes and 56.5 tool calls per instance (Table 17). Tool calls are a proxy for agent activity, not a count of raw LLM API calls. Mutation-generation

Table 15: Separate round-matched self-refinement experiment. Comparisons use the runs within this table; the vanilla COGEN result is not the run in Table 2. Matching the round budget does not imply equal token use or expenditure.

Method	Round budget	Resolved (%)	Test F→P (%)
Vanilla COGEN	1	61.7	72.3
Ralph-Loop	5	63.8	66.7
COHARDEN	5	67.9	75.3

Table 16: Accumulated per-instance API costs at successive round checkpoints in the final 433-instance run. Generation cost is the mutation-LLM component. The R5 total is the final cost, not the sum of the checkpoint values.

Checkpoint	Total (USD/instance)	Mutation generation (USD/instance)
R0	0.115	0.000
R1	0.492	0.049
R2	0.638	0.062
R3	0.696	0.071
R4	0.744	0.079
R5 / final	0.834	0.079

latency is measured per round, whereas the other two columns are per instance. This profile does not separately quantify token totals, mutant-execution overhead, or timeout rates.

Table 17: Execution profile of the final 433-instance run. Wall-clock duration and tool calls are per instance; mutation-generation latency is per round. These distributions do not establish matched expenditure for Ralph-Loop.

Statistic	Wall-clock (min)	Tool calls	Mutation latency (s)
Mean	22.0	56.5	66.0
Median	16.0	44	57.9
25th–75th percentile	12.5–27.5	33–70	46.6–73.0
95th percentile	49.2	132.8	111.9
Maximum	76.2	188	431.4

E.6 FAILURE ANALYSIS

The follow-up classifies the 139 unresolved cases from the full-budget 433-instance run into four categories (Table 18). Of these unresolved cases, 34 (24.5%) exhibit later-round test–fix coupling. This is 24.5% of failures, not of all benchmark instances. Another 44 cases have partial fixes despite tests classified as Rigorous by an independent post-hoc check. Rigorous is an operational classification under a finite mutant pool, so these cases do not establish that the test is an exhaustive specification or that all remaining error lies solely in fix generation.

The analysis separates failure to obtain an aligned initial test from later drift. The former limits entry into useful hardening; the latter shows how consecutive-round comparisons can miss shared errors. Lax-init addresses the initial condition, but neither it nor the Temporal Matrix guarantees correctness after joint test and fix updates. These observations support treating mutation-based hardening as a repair aid with identifiable failure modes, rather than a correctness certificate.

Table 18: Reported classification of all 139 unresolved cases. Percentages use the unresolved-case denominator and sum to approximately 100% after rounding. Rigorous/Lax labels are based on a post-hoc evaluation, not a correctness proof.

Failure category	Count	Share of unresolved (%)
Initially misaligned test	56	40.3
Later-round test–fix coupling	34	24.5
Partial fix with a post-hoc Rigorous test	44	31.7
Post-hoc Lax test	5	3.6
Total	139	100.0

F Extended Generalization and Ablation Results

F.1 Generalization across backbones and benchmarks

The cross-backbone scores and average per-instance costs are reported together in Table 3 in the main text. These measurements use GPT-5.4 and Claude Opus 4.5, whereas the primary comparison in Table 2 uses GPT-5-mini. Table 4 reports the SWE-bench Lite comparison.

F.2 Phase ablation and repair dynamics

Table 5 in the main text retains the original 433-instance phase ablation, and Figure 4 uses the original repair-dynamics image. R0 is test-only; its plotted zero does not represent a repair attempt. The stacked panels in Figure 5 separately show the test-quality diagnostic. The repeated-run, 60-instance results in Section E.3 remain a separate evaluation, not uncertainty estimates for the full-benchmark scores.

G Prompts

In this section, we provide the prompts used in the different configurations of our framework.

Mutation Operator Definitions

Four semantic-level mutation operators used to generate adversarial code patches. Each operator instructs the LLM to produce a "plausible but incorrect" fix using a specific strategy. These operators are injected into the sampling prompt via the `{{ operator }}` variable.

Operator 1: Symptom Suppression

The bug still occurs and produces incorrect behavior, but you HIDE the observable signal that the test relies on to detect it. The test checks for a specific error signal (exception, warning, assertion error, wrong error message, etc.). Your patch should suppress or silence that signal without fixing the underlying computation.

Common techniques:

- Wrap buggy code in try/except to swallow exceptions
- Use `warnings.catch_warnings()` to suppress warnings
- Redirect or silence error output
- Catch and re-raise a different (expected) exception type

Operator 2: Incomplete Fix

Take the gold fix and REMOVE part of it. Apply only a SUBSET of the changes — enough for the test to pass, but leaving some of the bug unfixed.

Concrete operations:

- If the gold fix modifies multiple locations, apply changes at only ONE location and leave the others unchanged
- If the gold fix handles multiple conditions/types, add an `if` guard that restricts it to only ONE condition/type
- If the gold fix changes multiple stages of a pipeline, only fix one stage and leave the rest broken

- If the gold fix adds both a computation AND error handling, only add the computation and skip the error handling

Operator 3: Input-Specific Shortcut

You add a SHORTCUT that prevents the test input from ever reaching the buggy code. The bug is not fixed — the buggy code still exists unchanged — but the test input is intercepted and handled separately before it gets there.

Common techniques:

- Add an early return for the specific input shape/type/value
- Add a guard clause that routes test inputs to a clean path
- Add a cache/memoization that returns a pre-computed result
- Insert a special-case branch before the buggy code region

Operator 4: Behavior Substitution

Take the gold fix and REPLACE its core expression or function call with a DIFFERENT one that is syntactically valid and plausible. Keep the same code location but use a different approach.

Key constraints:

- Your patch must touch the SAME lines as the gold fix — do not add code elsewhere
- Replace the key expression/operation with a different but syntactically valid alternative

Concrete operations:

- If the gold fix uses one type cast, use a DIFFERENT type cast (e.g., a narrower or platform-dependent type)
- If the gold fix uses one API/function, use a DIFFERENT API/function that has similar but not identical behavior
- If the gold fix uses one comparison operator, use a DIFFERENT operator (e.g., >= instead of >)
- If the gold fix reorders operations in one way, reorder them in a DIFFERENT way

Mutation Patch Sampling Prompt

We are currently solving the following issue within our repository. Here is the issue text:

```
--- BEGIN ISSUE ---
{{ problem_statement }}
--- END ISSUE ---
```

Below are some code segments, each from a relevant file. One or more of these files may contain bugs.

```
--- BEGIN FILE ---
{{ content }}
--- END FILE ---
{{ operator }}
```

Generate `edit_file` commands to implement your patch.

```
--- Test 1 ---
{{ test_1 }}
--- End Test 1 ---
--- Test 2 ---
{{ test_2 }}
--- End Test 2 ---
```

Two tests exist for this bug. Study both carefully and note their differences in assertions, inputs, and coverage. Your patch should produce DIFFERENT outcomes on these two tests — passing one while failing the other.

```
--- Reference Fix (the correct developer patch) ---
{{ gold_fix }}
--- End Reference Fix ---
```

The above is the correct fix. Use it as a starting point to generate a VARIANT patch — one that partially fixes the bug or fixes it in a subtly different way, so that the two tests above produce different outcomes.

The `edit_file` command takes four arguments:

`edit_file(filename: str, start: int, end: int, content: str) -> None:`

Edit a file. It replaces lines `start` through `end` (inclusive) with the given text `content` in the open file.

Args:

`filename`: The full file name to edit.

`start`: The start line number. Must satisfy `start >= 1`.

`end`: The end line number. Must satisfy `start <= end <= number of lines in the file`.

`content`: The content to replace the lines with.

Please note that THE `edit_file` FUNCTION REQUIRES PROPER INDENTATION. If you would like to add the line ‘ `print(x)`’, you must fully write that out, with all those spaces before the code! Wrap the `edit_file` command in blocks “‘`python...`“‘.

CoHarden Lax-Init Prompt (Round 0)

```
<uploaded_files>
/workspace/{{ workspace_dir_name }}
</uploaded_files>
```

I've uploaded a python code repository in the directory `{{ workspace_dir_name }}`. Consider the following issue description:

```
<issue_description>
{{ instance.problem_statement }}
</issue_description>
```

{% block task %}
Your task is to write a test that reproduces this bug, then fix it.
{% endblock %}

Your fix should resolve the issue with focused changes. Your test should reproduce the issue — it should fail before the fix is applied and pass after. Make sure the test is implementation-agnostic: it should verify the expected behavior from the project's semantics, not details specific to your fix.

After each step you will receive feedback guiding you to the next step. The workflow is:

- **Round 0**: write `reproduction.py` only — no fix yet.
- **Round 1**: find the root cause and the boundary.
- **Round 2+** (Hardening): refine both the fix and the test based on feedback.

Guidelines for Round 0:

The issue description's expected behavior or suggested root cause may not be accurate — **question the issue's assumptions**. The correct behavior depends on the project's semantics, which you must verify yourself.

Start from signals that **cannot be wrong**: an operation that crashes should not crash; a result that is clearly garbage should be reasonable; a value that should never be None should not be None. These are strong witnesses because they reduce the risk of checking the wrong thing.

When the symptom may cross layers, start from the **outermost witness** first:

1. Name the **public operation** described in the issue.
2. Name the **final user-visible observable** you want to check.
3. Ask whether the **internal object is already wrong**, or whether **only the rendered/serialized/printed output is wrong** — this tells you at which layer the bug first appears.

Assert that the return value has the **expected type or structure** when the exact value is clear enough. When the correct output value is uncertain, assert a **consistency property** (e.g., two calls that should produce the same result, or a round-trip that should preserve data). Exit code 0 = correct behavior, non-zero = bug present.

In Round 0, make that final observable your **main witness**: printed output, emitted SQL/JSON, rendered response, warning text, exception text, returned value, or visible side effect. Use internal values only as supporting sanity checks.

CoHarden Lax-Init Prompt (Round 0, continued)

Steps (Round 0):

1. Explore the repo to familiarize yourself with its structure, test framework, and conventions. Read the source files mentioned in the issue.
2. Run `cat /workspace/.skills/cogen-refine/references/prob-minimal-reproduction.md` to learn how to create an effective minimal reproduction.
3. **Boundary localization (REQUIRED)**:
   Run `cat /workspace/.skills/cogen-refine/references/examine-boundaries.md`.
   Then produce three outputs: (a) Boundary chain: list the layers from public API to symptom.
   (b) Chosen boundary: state which boundary your reproduction will exercise. (c) Plausibility check: name 2 candidates for where the gold fix might live.
4. Create `reproduction.py` inside the project directory using the boundary from step 3(b). Start from the user-level operation described in the issue.
5. Execute `python reproduction.py` and confirm it exits non-zero because of the bug.
6. Call `finish` — you will receive feedback guiding you through the remaining steps.

Conventions: Create `reproduction.py` inside `/workspace/{{ workspace_dir_name }}/`. Keep all changes uncommitted throughout the process — the detector evaluates via `git diff`. Use `git stash` / `git stash pop` for F→P verification.

Your thinking should be thorough and so it's fine if it's very long.

CoHarden Bootstrap Prompt (Round 1)

Round 0 complete — your `reproduction.py` correctly detects the bug (exits non-zero on buggy code).

What to do next (Round 1: identify scope, find the boundary, write the fix):

1. **Identify scope (sibling enumeration)** — bugs often manifest at multiple sibling-shaped sites that `reproduction.py` exercises only one of. Before debugging, enumerate the candidate scope. Sibling shapes to consider:
   - Parallel implementations: per-backend, per-renderer, subclass overrides
   - Mirror printers/converters: `str/latex/pretty/repr`, `to_python/from_python`, `encode/decode`
   - Caller direction: who CALLS the buggy function?
   - Companion methods: paired changes (`add` ↔ `clear`, `__mul__` ↔ `__rmul__`)

   For each candidate shape, run `grep -rln '<symbol>' -include='*.py'` and inspect each result. Record the sibling files — they are the SCOPE your fix must address.
2. **Debug to find the root cause**:
   Read `/workspace/.skills/cogen-refine/references/deepen-from-symptom-to-cause.md`. Starting from the symptom your `reproduction.py` exhibits, trace inward along the witness chain to the **boundary** — the first inner layer where the behavior turns from correct to wrong.
3. **Determine expected behavior**: Based on your root cause analysis, form a hypothesis about what the correct behavior should be. Verify by examining the project's specification (existing tests, docstrings, API contracts).
4. **Strengthen `reproduction.py` along the boundary**: state the expected vs actual behavior at the boundary, then add hypothesis checks — assertions on intermediate values along the boundary path. If step 1 surfaced multiple sibling sites, add at least one assertion per sibling.
5. **Write a fix** targeting the boundary — each change should follow from the expected behavior in step 3. Apply the fix at every in-scope sibling that shares the buggy pattern. Run the entire test module to check for regressions: `PYTHONPATH={{ repo_dir }} {{ test_framework }} <test_module>`
6. **Verify F→P**: Revert only your fix (not the test) by running `git stash push <fix_files>`, run `reproduction.py` to confirm it FAILS, then restore with `git stash pop`.
7. Call `finish`.

CoHarden Hardening Feedback Prompt (Round ≥ 2)

`{{ n_total - n_rejected }}/{{ n_total }}` mutants accepted. Vs prev round: `{{ temporal.a_T }}` persistent · `{{ temporal.b_T }}` regressed · `{{ temporal.c_T }}` newly killed ✓.

Why this matters: This mutant slipped through your test, which means your test is narrow on some dimension — it cannot tell your fix apart from a wrong-fix that produces the same symptom. By symmetry, your fix may have the same blind spot. Re-debug from the symptom; then strengthen BOTH `reproduction.py` and the fix.

Top accepted mutant:

```
{{ top_exemplar_diff }}
```

Other surviving operators: `{{ surviving_ops[1:] | join(', ') }}`. Per-mutant diffs + hints in `/workspace/.adversarial_feedback/latest.md`.

Fix-side audit (run before `finish`):

1. **Root-cause depth**: is the function you edited the deepest one returning the wrong value? If the symptom flows from a callee whose value is also wrong, fix the callee — not the symptom site.
2. **Sibling-fix enumeration**: for each symbol your fix touches, run `grep -rln '<symbol>'` and inspect each non-edited file. Apply the fix at every site sharing the buggy pattern, or articulate why each is unaffected.
3. **Scope check**: every file in your diff should appear in the issue's stack trace or be named in the issue text.
4. **Output-shape match**: if the fix changes a returned value, message, or exception, predict the exact output and compare to the issue text's expected output. If your prediction differs, restructure the fix.

Verify before `finish`: (1) Re-run `reproduction.py` → F→P holds. (2) Run existing tests for your edited symbol; any newly-failing test → narrow the fix.

FIX-ONLY Prompt

I have access to a python code repository in the directory `{{ instance.repo_path }}`. You can explore and modify files using the available tools. Consider the following issue description:

```
<issue_description>
{{ instance.problem_statement }}
</issue_description>
```

Can you help me implement the necessary changes to the repository so that the requirements specified in the `<issue_description>` are met? I've already taken care of all changes to any of the test files described in the `<issue_description>`. This means you DON'T have to modify the testing logic or any of the tests in any way! Also the development Python environment is already set up for you (i.e., all dependencies already installed), so you don't need to install other packages. Your task is to make the minimal changes to non-test files in the `{{ instance.repo_path }}` directory to ensure the `<issue_description>` is satisfied.

Follow these phases to resolve the issue:

Phase 1. READING: read the problem and reword it in clearer terms

- 1.1 If there are code or config snippets, express in words any best practices or conventions in them.
- 1.2 Highlight message errors, method names, variables, file names, stack traces, and technical details.
- 1.3 Explain the problem in clear terms.
- 1.4 Enumerate the steps to reproduce the problem.
- 1.5 Highlight any best practices to take into account when testing and fixing the issue.

Phase 2. RUNNING: install and run the tests on the repository

- 2.1 Activate the environment by running `./opt/miniconda3/etc/profile.d/conda.sh ; conda activate testbed`
- 2.2 Follow the readme and install the environment and anything needed.
- 2.3 Iterate and figure out how to run the tests.

Phase 3. EXPLORATION: find the files that are related to the problem and possible solutions

- 3.1 Use `grep` to search for relevant methods, classes, keywords and error messages.
- 3.2 Identify all files related to the problem statement.
- 3.3 Propose the methods and files to fix the issue and explain why.
- 3.4 From the possible file locations, select the most likely location to fix the issue.

Phase 4. TEST CREATION: create a script to reproduce and verify the issue

- 4.1 Look at existing test files in the repository to understand the test format/structure.
- 4.2 Create a minimal reproduction script that reproduces the located issue.
- 4.3 Run the reproduction script to confirm you are reproducing the issue.
- 4.4 Adjust the reproduction script as necessary.

Phase 5. FIX ANALYSIS: state clearly the problem and how to fix it

- 5.1 State clearly what the problem is.
- 5.2 State clearly where the problem is located.
- 5.3 State clearly how the test reproduces the issue.
- 5.4 State clearly the best practices to take into account in the fix.
- 5.5 State clearly how to fix the problem.

Phase 6. FIX IMPLEMENTATION: Edit the source code to implement your chosen solution.

- 6.1 Make minimal, focused changes to fix the issue.

Phase 7. VERIFICATION: Test your implementation thoroughly.

- 7.1 Run your reproduction script to verify the fix works.
- 7.2 Add edge cases to your test script to ensure comprehensive coverage.
- 7.3 Run existing tests related to the modified code to ensure you haven't broken anything.

Phase 8. FINAL REVIEW: Carefully re-read the problem description and compare your changes with the base commit `{{ instance.base_commit }}`.

- 8.1 Ensure you've fully addressed all requirements.
- 8.2 Run any tests in the repository related to: the issue you are fixing, the files you modified, and the functions you changed.
- 8.3 If any tests fail, revise your implementation until all tests pass.

Be thorough in your exploration, testing, and reasoning. It's fine if your thinking process is lengthy - quality and completeness are more important than brevity.

TDD-style BRT Injection Prompt

I have access to a python code repository in the directory `{{ instance.repo_path }}`. You can explore and modify files using the available tools. Consider the following issue description:

```
<issue_description>
{{ instance.problem_statement }}
</issue_description>
```

A reproduction test has already been written for this issue. Apply the test below to the repository, then fix the source code so the test passes.

```
<reproduction_test_patch>
{{ instance.tdd_test_patch }}
</reproduction_test_patch>
```

The development Python environment is already set up for you (i.e., all dependencies already installed), so you don't need to install other packages. Your task is to make the minimal changes to non-test files in the `{{ instance.repo_path }}` directory to ensure the `<issue_description>` is satisfied and the reproduction test passes.

Follow these phases to resolve the issue:

Phase 1. READING: read the problem and the reproduction test

- 1.1 Read the issue description and reword it in clearer terms.
- 1.2 Read the reproduction test patch carefully. Identify which test file(s) and test function(s) it adds or modifies.
- 1.3 Understand what behavior the test asserts — what inputs does it use, what outputs does it expect?
- 1.4 Explain the connection between the issue and the test: how does the test expose the bug?

Phase 2. TEST SETUP: apply the reproduction test to the repository

- 2.1 Activate the environment by running `./opt/miniconda3/etc/profile.d/conda.sh ; conda activate testbed`
- 2.2 Apply the reproduction test patch using `git apply` (or manually create the test files).
- 2.3 Run the reproduction test to confirm it **fails** on the current (buggy) code.
- 2.4 If the test does not fail, re-read the patch and the issue — you may need to adjust how you apply it.

Phase 3. EXPLORATION: find the source code related to the failing test

- 3.1 Use `grep` to search for the methods, classes, and keywords referenced by the test and the issue.
- 3.2 Identify all source files related to the problem.
- 3.3 Propose candidate fix locations and explain why.
- 3.4 Select the most likely location to fix.

Phase 4. FIX ANALYSIS: plan the fix

- 4.1 State clearly what the problem is.
- 4.2 State clearly where the problem is located.
- 4.3 State clearly how the reproduction test exposes the issue.
- 4.4 State clearly the best practices to take into account in the fix.
- 4.5 State clearly how to fix the problem.

Phase 5. FIX IMPLEMENTATION: Edit the source code to implement your chosen solution.

- 5.1 Make minimal, focused changes to fix the issue.

Phase 6. VERIFICATION: Test your implementation thoroughly.

- 6.1 Run the reproduction test to verify the fix makes it pass.
- 6.2 Run existing tests related to the modified code to ensure you haven't broken anything.
- 6.3 If any tests fail, revise your implementation until all tests pass.

Phase 7. FINAL REVIEW: Carefully re-read the problem description and compare your changes with the base commit `{{ instance.base_commit }}`.

- 7.1 Ensure you've fully addressed all requirements.
- 7.2 Run any tests in the repository related to: the issue you are fixing, the files you modified, and the functions you changed.
- 7.3 If any tests fail, revise your implementation until all tests pass.

Be thorough in your exploration, testing, and reasoning. It's fine if your thinking process is lengthy - quality and completeness are more important than brevity.

COGEN Prompt

```
<uploaded_files>
/workspace/{{ workspace_dir_name }}
</uploaded_files>
```

I've uploaded a python code repository in the directory `{{ workspace_dir_name }}`. Consider the following issue description:

```
<issue_description>
{{ instance.problem_statement }}
</issue_description>
```

{% block task %}
Your task is to fix this bug and write a test that reproduces it. Both the fix and the test must be in your final patch.
{% endblock %}

Your fix should resolve the issue with minimal changes. Your test should reproduce the issue — i.e., it should fail before the fix is applied and pass after. Make sure the test is implementation-agnostic: it should verify the correct behavior described in the issue, not details specific to your fix.

Important: After you submit, your test will be automatically evaluated against alternative code patches — some may be correct fixes using a different approach, others may be plausible but subtly wrong. Your test should be robust enough to:

- PASS on any correct fix (regardless of implementation details)
- FAIL on patches that don't fully resolve the underlying bug

Use your understanding of the issue semantics to decide what the correct behavior should be, and test for that rather than for artifacts of your particular fix.

Follow these steps to resolve the issue:

1. As a first step, it might be a good idea to explore the repo to familiarize yourself with its structure.
2. Create a script `reproduction.py` to reproduce the error and execute it with `python reproduction.py` using the BashTool, to confirm the error.
3. The issue describes a SYMPTOM. Find WHERE the root cause is — it may be in a different module than where the symptom appears. Test one hypothesis at a time.
4. Fix the bug with a minimal code change.
5. Re-run `reproduction.py` to confirm the fix resolves the issue.
6. Run the existing test suite for the module(s) you modified to make sure your fix doesn't break anything. Fix any regressions before proceeding.
7. Edit the sourcecode of the repo to integrate your reproduction script into the test framework. Find the most relevant existing test file. Make sure your test fails on the buggy code and passes on the fixed code.
8. Verify F→P: revert your fix with `git stash`, run your test to confirm it FAILS, then restore with `git stash pop`.
9. Clean up: remove `reproduction.py` and any other scratch files before finishing. Only source and test file changes should remain in your patch.

BRT-ONLY Prompt

```
<uploaded_files>
/workspace/{{ workspace_dir_name }}
</uploaded_files>
```

I've uploaded a python code repository in the directory `{{ workspace_dir_name }}`. Consider the following issue description:

```
<issue_description>
{{ instance.problem_statement }}
</issue_description>
```

Can you help me implement the necessary changes to the repository to test whether the issue in `<issue_description>` was resolved? I will take care of all changes to any of the non-test files. This means you DON'T have to modify the actual logic and ONLY have to update test logic and tests!

Your task is to make the minimal changes to tests files in the `/workspace` directory to reproduce the issue in the `<issue_description>`, i.e., such that the generated tests fail in the current state (where the issue is unresolved) and pass when the issue will be resolved.

Follow these steps to reproduce the issue:

1. As a first step, it might be a good idea to explore the repo to familiarize yourself with its structure.
2. Create a script `reproduction.py` to reproduce the error and execute it with `python reproduction.py` using the BashTool, to confirm the error.
3. Edit the sourcecode of the repo to integrate your reproduction script into the test framework.
4. Run the test framework and make sure your tests fail! Only submit FAILING tests! Never submit passing tests.

Your thinking should be thorough and so it's fine if it's very long.