

Can Vibe Coders Really Pass the Vibe Check? An Empirical Analysis of Fault-Targeted Reasoning in Frontier LLMs

Srijan Bansal¹ Jiao Fangkai^{2,3} Yilun Zhou^{1,4*}

Austin Xu^{1,5*} Shafiq Joty^{1,2} Semih Yavuz¹

¹Salesforce AI Research ²Nanyang Technological University ³A*STAR

⁴Datadog ⁵Anthropic

 Code  Data

Abstract

We conduct a controlled study isolating reasoning in autonomous debugging by holding scaffolding and tool access fixed. We decompose fault-targeted reasoning into test generation and repair using LLM-generated solutions that pass most tests but fail on edge cases. Across 12 frontier and open-source models, we identify three limitations. *First*, despite 87.7% input validity, only 68.3% of inputs are discriminative, identifying fault-hypothesis formation as the bottleneck: finding fault-triggering inputs costs 19.4 points versus 2.7 points for output prediction. *Second*, diagnostic feedback has mixed effects on repair: tests generated internally within the repair response reduce success by 5.6 points relative to a bug label alone, while separately generated same-model tests differ little on average, but repair succeeds more often with discriminative than non-discriminative tests (66.8% versus 25.2%). *Third*, capabilities do not consistently compose: success across bug detection, test construction, and repair does not consistently occur on the same instances, with the largest losses in forming a fault-triggering input and converting a discriminative test into a correct repair. Together, these findings identify both transitions as key debugging limits.

1 Introduction

Recent advances in large language models (LLMs) have accelerated *vibe coding*: agentic workflows that generate substantial software with minimal human oversight (Dong et al., 2025). Although recent systems achieve near-ceiling pass rates on established benchmarks (Austin et al., 2021; Jain et al., 2024; Chen et al., 2021a), LLM-generated code can pass visible tests while failing on edge cases (Liu et al., 2023, 2024a, 2025). This mismatch is a reliability concern for coding assistants.

Agentic evaluations also combine model reasoning with scaffolding and tool use, making it difficult to attribute failures to reasoning or system design. Reliable deployment requires models to verify code, diagnose latent faults that evade tests, and synthesize repairs. We therefore conduct a controlled study that holds scaffolding and tool access fixed to identify where reasoning errors arise. This leads to the central question: **Given a partially correct program with no observed failure, can an LLM synthesize a fault-triggering input and use it for repair?**

We term this capability *fault-targeted reasoning*: the ability to surface failure-inducing inputs and use them to drive repair. Prior work evaluates its components in isolation. Test-generation research (Altmayer Pizzorno and Berger, 2025; Chen et al., 2024b; Huang et al., 2025a) relies on validity, coverage, or mutation scores as proxies for fault exposure. Repair research (Tian et al., 2024; Chopra et al., 2024; Huang et al., 2025b) often supplies an explicit fault signal, such as a location or bug description, or evaluates only the final patch. These protocols do not jointly measure the counterexample construction and diagnosis-guided repair required for autonomous debugging. We test whether proxy metrics track fault exposure and where diagnosis limits end-to-end reliability.

We conduct a controlled empirical study that systematically evaluates fault-targeted reasoning through two complementary tasks (Figure 1). **Fault-Triggering Test Generation (Task 1)** measures whether LLMs can generate tests that expose bugs in partially correct model-generated solutions. Unlike prior test-generation benchmarks focused on coverage or correctness verification, we require tests to be *discriminative*: valid input-output pairs on which buggy and accepted implementations diverge. **Fault-Targeted Program Repair (Task 2)** evaluates the ability to fix these bugs under three guidance conditions—a bug label

*Work done while at Salesforce AI Research.

alone, an externally supplied fault-triggering test generated by the same model, and a test generated within the repair response—enabling a controlled study of how diagnostic context affects repair success. Our setup uses 173 instances spanning 76 challenging problems from LiveCodeBench, each paired with a model-generated buggy solution and a platform-accepted human-authored solution, isolating reasoning from scaffold and tool effects.

Evaluating fault-targeted reasoning across 12 frontier and open-source models yields three findings. First, although 87.7% of generated inputs are valid, only 68.3% are discriminative, with fault-hypothesis formation the main bottleneck: the 19.4-point gap is approximately seven times the 2.7-point output-prediction gap. Second, tests generated internally within the repair response reduce repair success by 5.6 points relative to receiving only a bug label, while externally supplied same-model tests have mixed effects. Repair succeeds in 66.8% of cases with discriminative tests and 25.2% otherwise, a 41.6-point difference. Third, success across bug detection, test construction, and repair does not consistently occur on the same instances. The largest losses arise in forming a fault-triggering input and converting a discriminative test into a correct repair.

Our contributions are: (i) a controlled study of fault-targeted reasoning across 12 frontier and open-source LLMs, holding scaffolding and tool access fixed; (ii) an execution-based evaluation framework that separates valid-input generation, fault-triggering input construction, output prediction, and program repair; and (iii) a cluster-aware analysis identifying the main failures in forming fault-triggering inputs and converting discriminative tests into correct repairs, with mixed benefits from model-generated tests beyond a bug label.

2 Related Work

Code generation has been evaluated on function-level tasks (Chen et al., 2021a; Austin et al., 2021), algorithmic reasoning (Li et al., 2022; Hendrycks et al., 2021; Khan et al., 2024; Jain et al., 2024; Zheng et al., 2025), and software engineering workflows (Liu et al., 2024b; Jimenez et al., 2024; Ding et al., 2025). However, these benchmarks focus on solution synthesis or patch correctness rather than whether models can identify, localize, and reason about failures. By treating debugging as a downstream outcome, they obscure the diag-

nostic process required for informative counterexample construction. Consequently, strong coding performance does not imply robust fault-targeted reasoning or reliable autonomous debugging.

Understanding Model Capabilities and Limitations. While models achieve high performance on isolated generation tasks, Fried et al. (2023) highlights mismatches between autoregressive objectives and editing patterns. Software engineering workflows requiring long-context understanding, bug fixing, and repository-level reasoning expose persistent capability gaps (Liu et al., 2024b; Jimenez et al., 2024; Ding et al., 2025; Li et al., 2024). CodeChain shows that modular self-revision can improve correctness, but underlying reasoning limitations remain underexplored (Le et al., 2024).

Analyzing Debugging and Repair Capabilities. Models can leverage execution feedback for iterative refinement (Chen et al., 2024a; Wang et al., 2024; Ni et al., 2023), but reasoning limitations remain underexplored. CodeT (Chen et al., 2022) uses LLM-generated tests for solution selection through execution agreement, measuring consistency rather than discriminative power. UT-Gen (Prasad et al., 2025) proposes training methods for error-revealing tests with test-time compute and validation. Repository repair (Jimenez et al., 2024; Wei et al., 2025; Reddy et al., 2025) and unit-test synthesis (Lemieux et al., 2023; Zhu et al., 2025) demonstrate gains from scaffolding, but leave reasoning bottlenecks unresolved.

Despite these advances, existing evaluations combine model reasoning with agentic scaffolding and tool access, making attribution of failures difficult. We conduct a controlled empirical study of *fault-targeted reasoning*—bug detection, discriminative test construction, and program repair—while holding scaffolding and tool access fixed. Execution-based metrics then localize failures to specific stages.

3 Controlled Study Design

We decompose fault-targeted reasoning into two complementary capabilities (Figure 1), evaluating each under controlled conditions: **(Task 1) Fault-Triggering Test Generation (FT-Test)** measures the ability to craft discriminative tests that expose subtle semantic bugs, and **(Task 2) Fault-Targeted Program Repair (FPR)** measures the ability to repair these bugs under varying diagnos-

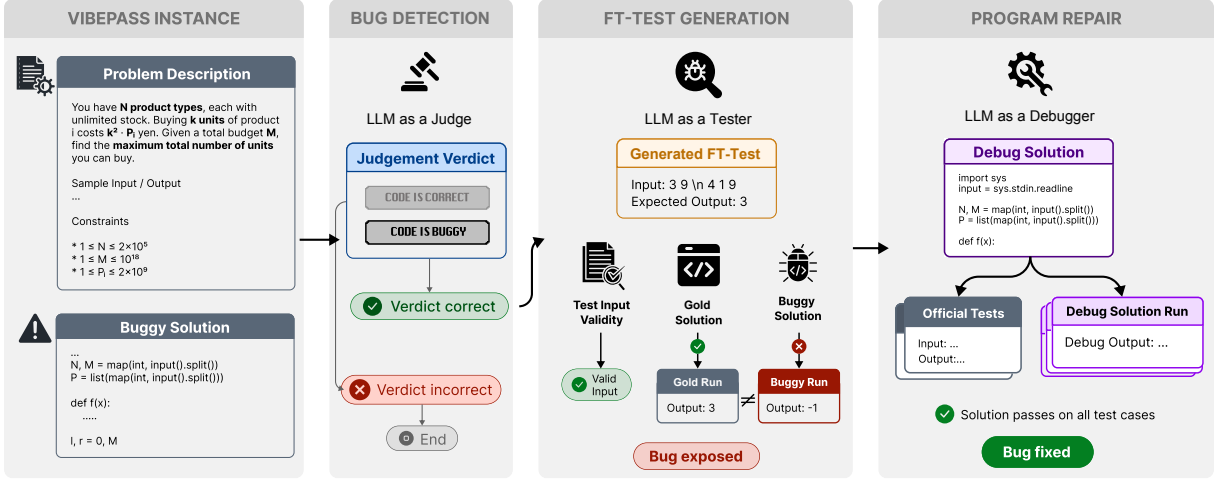


Figure 1: **Controlled decomposition of fault-targeted reasoning.** We isolate test generation from repair by evaluating each capability under varying diagnostic conditions. Given a problem description and a buggy solution, the model generates a fault-triggering (FT) test—and, in Bug-Discovery, first performs bug detection (**Task 1: FT-Test**). An FT-test is discriminative if its input is valid, the buggy solution fails, and the gold solution passes. The diagram shows the separately generated same-model test path. **Task 2: Fault-Targeted Program Repair** evaluates repair with a bug label alone (BUGLABELONLY), a separately generated same-model FT-test (EXTTEST), or an FT-test generated within the repair response (INTTEST).

tic guidance. Joint evaluation measures dependencies between test generation and repair, cascading errors through the diagnostic chain, and whether model-generated tests improve debugging. By controlling the problem unit (function-level code), bug type (semantic faults in near-correct solutions), and diagnostic context (BUGLABELONLY, EXTTEST, or INTTEST), we study reasoning while holding these factors fixed. Our design follows three core principles: diverse semantic bugs, execution-based verification, and multi-setting evaluation (Appendix A.1.1).

3.1 Dataset Construction

We construct VIBEPASS through a three-stage pipeline (Appendix A.2).

Problem Collection. We focus on function-level algorithmic problems for controlled analysis: they enable precise execution-based evaluation, fine-grained test generation, and isolate reasoning from repository navigation, build systems, and multi-file dependencies. We source problems from LiveCodeBench (Jain et al., 2024), whose continuous release mitigates train-test contamination. After discarding problems with conflicting test cases or non-standard evaluation, we apply capability-based filtering, retaining only problems not solved within three attempts by any of three reasoning models (GPT-4o, Claude Sonnet-4, Gemini-2.5-Pro). This yields 170 problems, 89% of which are

rated “medium” or “hard.”

Input-Validity Checker Generation. Evaluating test generation requires verifying that generated inputs satisfy problem constraints. For each problem, we automatically generate a Python checker `is_valid_test( $t_i$ )` $\rightarrow$ $\{\text{True}, \text{False}\}$ using GPT-5-Mini. We discard any checker that rejects an official input. As shown in Figure 4, 98.8% of problems yield a checker that accepts the official suite.

Solution Collection, Verification and Filtering. We collect 2,184 candidate solutions across 168 of the filtered problems from human and LLM submissions. Solutions passing all official tests are designated **gold**; wrong-answer failures are designated **buggy**—runtime errors are discarded as trivially detectable. We retain one gold and up to four diverse buggy solutions per problem.

Final Dataset. Our dataset contains 173 instances spanning 76 unique problems. Each instance comprises a problem specification, official test cases, an official-suite-validated input checker, a human-authored gold solution, and a model-generated buggy solution. We evaluate 12 models in five settings (Bug-Aware, Bug-Discovery, BUGLABELONLY, EXTTEST, and INTTEST), totaling more than 10,000 model evaluations. Across instances, buggy-solution pass rates range from 10% to 90% on the official tests, providing a range of fault difficulty. Table 5 provides full statistics.

3.2 Task 1: Fault-Triggering Test Generation (FT-Test)

Motivation. Effective fault localization requires targeted tests that expose bugs (Rafi et al., 2024, 2025)—central to test-driven development and automated program repair. Task 1 isolates the ability to generate *FT-tests* (inputs that trigger buggy behavior while passing correct implementations) from downstream repair, revealing where the diagnostic reasoning chain breaks: fault-hypothesis formation versus output validation.

Task Definition. Given a problem specification P and buggy solution S_{buggy} , generate a test $t = (t_i, t_o)$ with input t_i and expected output t_o . A test is FT iff $\text{is_valid_test}(t_i) = \text{True} \wedge S_{\text{gold}}(t_i) = t_o \wedge S_{\text{buggy}}(t_i) \neq t_o$.

Evaluation Settings. We control information availability through two settings. **Bug-Aware FT-Test** informs the model that S contains a bug, measuring test generation given a known fault. **Bug-Discovery FT-Test** requires the model to first determine whether S is correct or buggy, then generate an FT-test only if it judges S buggy, measuring combined bug detection and test generation. Comparing these settings isolates the effect of supplying the bug label.

Evaluation Metrics. We evaluate the FT-Test task using four progressively stricter criteria. **Validity** (V_I) measures whether the generated test input is valid, i.e., $\text{is_valid_test}(t_i) = \text{True}$. **Executability** (V_{IO}) additionally requires the gold implementation to produce the expected output: $S_{\text{gold}}(t_i) = t_o$. **Discriminative Input** (D_I) measures whether a valid test input distinguishes the buggy and gold implementations, i.e., $S_{\text{buggy}}(t_i) \neq S_{\text{gold}}(t_i)$. Finally, **Discriminative Test** (D_{IO}) is the strictest criterion, requiring a valid executable test whose expected output matches the gold implementation while differing from the buggy implementation: $S_{\text{gold}}(t_i) = t_o$ and $S_{\text{buggy}}(t_i) \neq t_o$.

The success events satisfy $\{D_{IO}=1\} \subseteq \{V_{IO}=1\} \cup \{D_I=1\} \subseteq \{V_I=1\}$. The $(V_I \rightarrow D_I)$ gap captures failures in fault-hypothesis formation, whereas the $(D_I \rightarrow D_{IO})$ gap captures failures in output prediction. For Bug-Discovery, we additionally report **Bug Detection Recall (BDR)**: the fraction of known-buggy solutions classified as buggy. Because every evaluated instance is buggy, BDR is a true-positive rate rather than classification accuracy and does not measure false posi-

tives on correct programs; an always-“buggy” classifier therefore achieves 100% BDR. Joint metrics ($\text{BDR}+*$) require both bug detection and the corresponding FT-Test criterion; for example, $\text{BDR}+D_{IO}$ denotes success in both stages.

3.3 Task 2: Fault-Targeted Program Repair (FPR)

Motivation. While Task 1 isolates test generation, Task 2 isolates repair capability by controlling diagnostic context. By varying whether models receive a separately generated same-model test, a test generated within the repair response, or no test beyond the bug label, we measure how model-generated diagnostic information affects repair.

Task Definition. Given a problem specification P , buggy solution S_{buggy} and diagnostic context C , generate a corrected solution S_{fixed} that passes all official test cases.

Evaluation Conditions. We control diagnostic context through three conditions, all of which state that the solution is buggy. **BUGLABELONLY** supplies the bug label but no test. **EXTTEST** provides an FT-test generated by the same model in Task 1 (Bug-Aware), decoupling test generation from repair; the generated test is not guaranteed to be discriminative. **INTTEST** requires the model to first generate its own fault-triggering test and then use it for repair within one response. We evaluate same-model test transfer only; cross-model transfer is outside this protocol.

Evaluation Metrics. A repair succeeds only if it passes every official test. **Pass@1** first averages success across variants of each problem and then across problems. Pair-level analyses report instance-level success percentages. For EXTTEST and INTTEST, we also assess test quality using Task 1 metrics (V_{IO} and D_{IO}).

3.4 Evaluated Models

We evaluate 12 frontier commercial and open-source models: **OpenAI** (GPT-5-Nano, GPT-5-Mini, GPT-5.2, GPT-5.2-Codex), **Google** (Gemini-3-Flash, Gemini-3-Pro, Gemini-3.1-Flash-Lite, Gemini-3.1-Pro), **Anthropic** (Sonnet-4.6, Opus-4.6), and **open-source** models GPT-OSS-120B and Nemotron-3-Nano-30B-A3B (abbreviated Nemotron-3-Nano). All models use their highest supported reasoning effort. Full experimental details are in Appendix A.4.

Table 1: **Bug-Aware and Bug-Discovery FT-Test Generation.** Bug-Aware assumes the model knows the code is buggy; its metrics directly evaluate FT-Test quality. Bug-Discovery additionally requires bug detection, reported as Bug Detection Recall (BDR) because all evaluated programs are known to be buggy. Joint metrics (BDR+*) require both bug detection and the corresponding FT-Test criterion. Pass@1 ($P_{@1}$) is the code-generation baseline.

Model	$P_{@1}$	Bug-Aware				Bug-Discovery				
		V_I	V_{IO}	D_I	D_{IO}	BDR	BDR+ V_I	BDR+ V_{IO}	BDR+ D_I	BDR+ D_{IO}
Gemini-3.1-Flash-Lite	25.0	89.6	69.4	31.8	26.0	39.9	33.0	24.3	20.8	15.6
Gemini-3-Flash	78.9	95.4	92.5	63.6	63.0	76.3	75.1	68.2	68.2	66.5
Gemini-3-Pro	81.6	94.8	89.6	72.3	69.4	83.2	80.4	71.1	64.2	59.5
Gemini-3.1-Pro	92.1	94.8	87.9	85.6	84.4	86.7	83.8	60.1	75.7	60.1
GPT-5-Nano	43.4	92.5	82.1	57.8	52.0	54.9	52.0	16.8	44.5	16.2
GPT-5-Mini	48.7	91.3	82.1	73.4	70.5	62.4	60.7	52.0	55.5	52.0
GPT-5.2	71.1	88.4	79.8	79.8	77.5	84.4	82.1	75.7	75.7	74.6
GPT-5.2-Codex	71.1	91.9	85.0	85.0	83.8	86.7	83.8	76.3	75.7	75.1
Sonnet-4.6	57.9	84.4	75.1	74.6	73.4	80.4	77.5	70.5	70.5	70.5
Opus-4.6	73.7	87.9	83.8	83.2	83.2	82.7	79.8	75.7	75.1	75.1
GPT-OSS-120B	67.1	76.3	69.9	66.5	65.3	74.0	70.5	66.5	67.1	65.9
Nemotron-3-Nano	26.3	65.0	48.0	46.3	39.3	57.2	52.0	15.0	39.9	13.3
Mean	61.4	87.7	78.8	68.3	65.6	72.4	69.2	56.0	61.1	53.7

3.5 Statistical Analysis

We compute 95% confidence intervals by cluster-bootstrap resampling the 76 problem clusters 10,000 times, keeping all buggy variants of a sampled problem together. Confidence intervals for cross-model averages additionally resample the 12 models in an outer bootstrap; all bootstrap analyses use seed 42. For paired binary outcomes, we use exact two-sided McNemar tests within each model. Comparisons between repairs receiving discriminative and non-discriminative tests use two-sided Fisher’s exact tests. We use $p < 0.05$ as the significance threshold. Because each prompt is sampled once at temperature 1 where configurable, the intervals quantify variation across sampled problems, not repeated-decoding variance.

4 Results and Analysis

Our controlled study reveals three key empirical findings about fault-targeted reasoning in frontier and open-source LLMs. By holding scaffolding and tool access fixed, we evaluate test generation and repair under controlled diagnostic contexts. Each finding identifies a capability bottleneck and failure patterns; Table 4 summarizes uncertainty and per-model robustness.

4.1 Finding 1: Fault-hypothesis bottleneck

Our hierarchical metrics (Table 1) isolate where test generation reasoning fails. In the Bug-Aware FT-Test setting, models generate valid inputs with

high reliability (average $V_I = 87.7\%$), yet only 68.3% achieve discriminative inputs (D_I) and 65.6% achieve discriminative FT-tests (D_{IO})—a 19.4-point fault-hypothesis gap followed by a 2.7-point output-prediction gap. This extends classical observations from automated testing (Chen et al., 2021b) to the LLM setting: input validity and fault detection are distinct capabilities. The gap arises because producing format-compliant inputs largely relies on learned specification patterns, whereas exposing faults requires reasoning about program behavior. The 58.4-point D_{IO} range between Gemini-3.1-Pro (84.4%) and Gemini-3.1-Flash-Lite (26.0%) shows substantial variation across models. Only 0.8% of the 2,076 Bug-Aware tests are reverse-discriminative—their expected output agrees with the buggy rather than the gold program—and the maximum rate for any model is 4.0%.

The decomposition isolates two failure modes: the *fault-hypothesis gap* ($V_I - D_I$, average: 19.4 points; 95% CI: 11.4–29.5) measures failures in identifying discriminative inputs, while the *output-prediction gap* ($D_I - D_{IO}$, average: 2.7 points; 95% CI: 1.2–4.8) measures failures in predicting outputs. The fault-hypothesis gap is approximately seven times larger and is significant for all 12 models under paired McNemar tests; the output-prediction gap is significant for 3 of 12 models. The fault-hypothesis gap varies across models, from 57.8 points for Gemini-3.1-Flash-Lite to 4.7 points for Opus-4.6. A sharp capacity

Table 2: **Model performance on FT-Test generation under Bug-Discovery and Bug-Aware settings.** B denotes a “buggy” prediction in Bug-Discovery. Bug-Aware uses the same B partition; Agr. denotes cross-setting D_{IO} agreement, and G denotes Gemini.

Model	Discovery		Aware		Agr.
	BDR	D_{IO}	B	D_{IO}	$\neg B$
G-3.1-Flash-Lite	39.9	39.1	34.8	20.2	87.0
G-3-Flash	76.3	87.1	74.2	26.8	76.5
G-3-Pro	83.2	71.5	75.0	41.4	84.0
G-3.1-Pro	86.7	69.3	90.0	47.8	75.3
GPT-5-Nano	54.9	29.5	66.3	34.6	48.4
GPT-5-Mini	62.4	83.3	89.8	38.5	84.3
GPT-5.2	84.4	88.4	84.9	37.0	93.8
GPT-5.2-Codex	86.7	86.7	89.3	47.8	96.0
Sonnet-4.6	80.4	87.8	82.0	38.2	92.8
Opus-4.6	82.7	90.9	89.5	53.3	97.2
GPT-OSS-120B	74.0	89.1	78.9	26.7	88.3
Nemotron-3-Nano	57.2	23.2	52.5	21.6	58.6
Mean	72.4	70.5	75.6	36.2	81.9

difference is visible at the lower end: GPT-5-Nano achieves 92.5% V_I but 57.8% D_I (a 34.7-point gap), and Gemini-3.1-Flash-Lite achieves 89.6% V_I but 31.8% D_I (a 57.8-point gap). In the paired analysis, the full $V_I - D_{IO}$ gap is 22.2 points (95% CI: 13.7–32.0) and is significant for every model. The tighter coupling $D_{IO}/D_I \approx 0.96$, compared with $V_{IO}/V_I \approx 0.90$, further shows that the larger loss occurs in identifying discriminative inputs.

Controlling information availability (Bug-Discovery versus Bug-Aware) isolates the effect of bug awareness. In Bug-Discovery, models must detect the bug before generating tests, creating a two-stage reasoning bottleneck. Because all evaluated programs are buggy, the detection metric is Bug Detection Recall (BDR), not general classification accuracy. Mean BDR reaches 72.4%, but end-to-end BDR+ D_{IO} falls to 53.7%—11.9 points below Bug-Aware D_{IO} (65.6%). This paired difference (95% CI: 5.4–19.6) is significant for 8 of 12 models. The drop from BDR to BDR+ D_{IO} is 26.6 points for Gemini-3.1-Pro, but only 7.6 points for Opus-4.6.

Conditioning on detection outcomes (Table 2) further isolates the reasoning bottleneck. Bug-Aware D_{IO} (65.6%) exceeds end-to-end Bug-Discovery BDR+ D_{IO} (53.7%), but conditioning on a “buggy” prediction (B) narrows the difference to 5.1 points (75.6% versus 70.5%). A pronounced asymmetry remains even in Bug-Aware:

Table 3: **Controlled comparison of diagnostic context in repair.** We report Pass@1 ($P_{@1}$) for the code-generation baseline (CG), a bug label alone (Label), a test generated within the repair response (Int), and a separately generated same-model test (Ext). Int and Ext tests are model-generated rather than gold.

Model	CG	Label	Int	Ext
Gemini-3.1-Flash-Lite	25.0	45.7	23.4	24.4
Gemini-3-Flash	78.9	74.6	68.9	73.5
Gemini-3-Pro	81.6	72.5	63.8	72.0
Gemini-3.1-Pro	92.1	75.7	71.8	70.9
GPT-5-Nano	43.4	27.7	29.0	37.2
GPT-5-Mini	48.7	45.2	41.7	45.7
GPT-5.2	71.1	68.0	65.6	64.4
GPT-5.2-Codex	71.1	65.5	69.2	64.3
Sonnet-4.6	57.9	59.8	53.5	60.3
Opus-4.6	73.7	69.5	62.2	70.8
GPT-OSS-120B	67.1	54.5	44.2	54.3
Nemotron-3-Nano	26.3	19.9	14.1	23.3
Mean	61.4	56.5	50.6	55.1

models achieve 75.6% D_{IO} on cases with B but only 36.2% on cases with $\neg B$. At the model level, Bug-Aware performs better than Bug-Discovery on the B subset for only 6 of 12 systems, indicating model-dependent bug-label effects. Nemotron-3-Nano, GPT-5-Nano, and GPT-5-Mini gain from Bug-Aware, while Claude models and GPT-OSS-120B do not; Gemini and GPT-5.2 are mixed.

In the Bug-Aware setting, strong models perform well on cases detected as buggy in Bug-Discovery (GPT-5.2-Codex: 89.3%; Opus-4.6: 89.5%) but less well on cases where detection fails (47.8% and 53.3%, respectively). GPT-5-Nano likewise drops from 66.3% to 34.6%. The 81.9% cross-setting agreement shows that test outcomes often remain consistent even when the Bug-Aware prompt supplies the bug label. This resembles selective prediction (Wen et al., 2025): Bug-Discovery permits implicit abstention through a “correct” prediction, whereas Bug-Aware requires test generation in every case. Most FT-Test failures arise before output prediction: models struggle primarily to identify inputs that expose the fault, while bug detection adds a separate upstream failure point in Bug-Discovery.

4.2 Finding 2: The value of model-generated tests depends on test quality

Controlling diagnostic context (Table 3) reveals whether test-based feedback improves repair beyond an explicit bug label. We evaluate three con-

Table 4: **Uncertainty and per-model significance.** Effects and cluster-bootstrap 95% CIs are in percentage points; Sig. counts models with $p < 0.05$.

Comparison	Effect [95% CI]	Sig.
$V_I - D_I$	19.4 [11.4–29.5]	12/12
$V_I - D_{IO}$	22.2 [13.7–32.0]	12/12
$D_I - D_{IO}$	2.7 [1.2–4.8]	3/12
BUGLABELONLY – INTTEST Bug-Aware D_{IO} vs.	5.6 [1.8–9.2]	8/12
Bug-Discovery BDR+ D_{IO}	11.9 [5.4–19.6]	8/12
Discr. vs. non-discr. EXTTEST	41.6 [34.7–48.4]	10/12

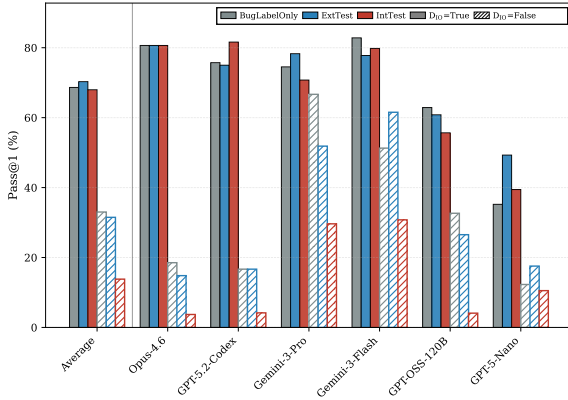


Figure 2: **Repair after matching external and internal test quality.** Pass@1 under BUGLABELONLY, EXTTEST, and INTTEST on instances for which the external and internal tests have the same D_{IO} status. Solid and hatched bars denote $D_{IO}=\text{True}$ and $D_{IO}=\text{False}$, respectively. The plot shows the average and six individual models.

ditions that isolate the impact of diagnostic information: BUGLABELONLY (the model knows the solution is buggy but receives no test), EXTTEST (the model receives its separately generated Task 1 Bug-Aware test), and INTTEST (the model generates a test and uses it for repair within the same response). No repair condition uses a gold test.

On average, adding an internally generated test degrades repair performance: mean Pass@1 drops from 56.5% for BUGLABELONLY to 50.6% for INTTEST, a 5.9-point descriptive gap. The paired instance-level estimate is 5.6 points (95% CI: 1.8–9.2) and is significant for 8 of 12 models. Separately generated same-model tests partially recover performance: EXTTEST reaches 55.1%, only 1.4 descriptive points below BUGLABELONLY. These aggregate comparisons do not separate test quality from how the repair response uses the test, which the analyses below examine.

Controlling for test quality (Figure 2) isolates

feedback use from test discriminativeness. We analyze instances for which the external Task 1 test and the test generated within Task 2 have identical discriminative status. When both tests are discriminative, BUGLABELONLY, EXTTEST, and INTTEST achieve 68.4%, 70.3%, and 68.2% Pass@1, respectively—minimal differences despite explicit fault evidence. When both tests are non-discriminative, the corresponding rates are 33.0%, 31.5%, and 13.8%, with INTTEST degrading most severely. This asymmetry shows error propagation when test generation and repair occur within the same response.

Across all 2,076 EXTTEST instance–model pairs, repair succeeds on 66.8% of the 1,363 pairs with a discriminative supplied test, compared with 25.2% of the 713 pairs with a non-discriminative test. The 41.6-point difference (95% CI: 34.7–48.4) is significant for 10 of 12 models under Fisher’s exact tests. Of the 985 EXTTEST failures, 54.1% receive a non-discriminative test, while 45.9% fail despite receiving a discriminative test.

A paired flow analysis of 1,021 cases for which both the external and internal tests are discriminative shows why the aggregate comparison is mixed: the external test changes a failed BUGLABELONLY repair into a successful EXTTEST repair in 7.0% of cases, but changes a successful repair into a failure in 8.4%. The remaining cases either succeed in both conditions (64.5%) or fail in both (20.1%). The 8.4% hurt flow is consistent with overfitting to, or otherwise misusing, the supplied test; the 20.1% both-fail flow instead contains cases where the discriminative signal does not suffice. These shares identify separate intervention points: non-discriminative tests require better upstream validation, whereas failures despite discriminative tests require better downstream use of diagnostic evidence. Discriminative tests are therefore highly informative at the population level, yet do not guarantee that a model will use the signal correctly in a particular repair.

4.3 Finding 3: Stage-level success does not ensure end-to-end success

Joint evaluation (Figure 3) reveals capability interactions that separate task evaluations cannot measure. We analyze correlations across six pipeline stages—Valid Input (V_I), Valid IO (V_{IO}), FT-Input (D_I), FT-IO (D_{IO}), Repair, and bug detection (BDR)—to identify where reasoning breaks down. FT-Input and FT-IO exhibit near-perfect model-

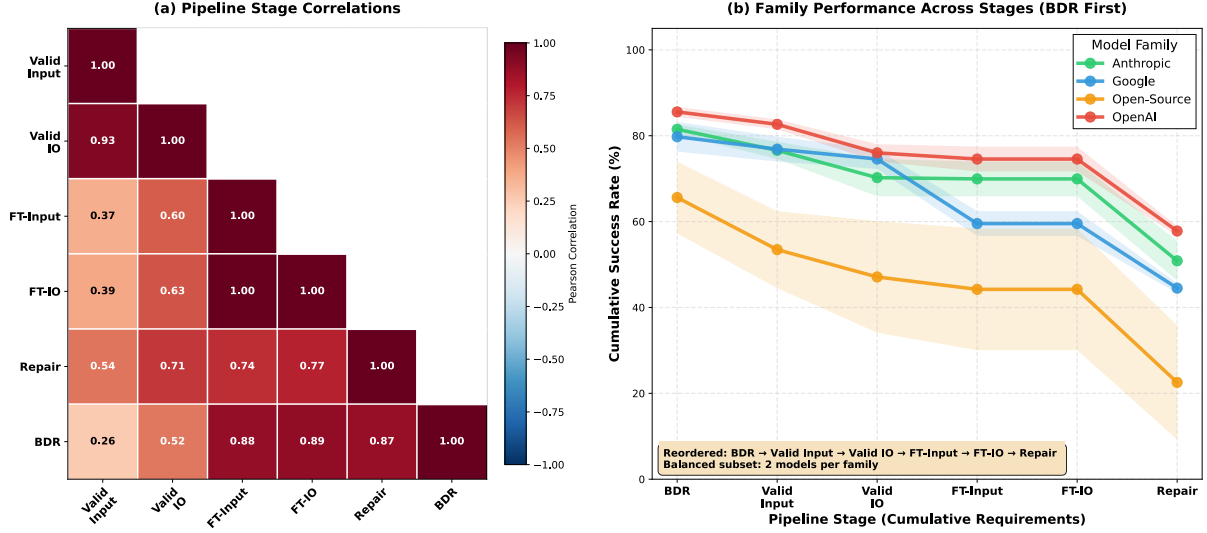


Figure 3: **Relationships among fault-targeted reasoning stages.** **Left:** Pearson correlations across model-level success rates; Repair denotes BUGLABELONLY repair and BDR denotes Bug Detection Recall. FT-Input and FT-IO are nearly collinear ($r = 0.998$), while FT-IO and repair correlate at $r = 0.77$. **Right:** Cumulative instance-level success across stages for a balanced subset of two models per family.

level correlation ($r = 0.998$), consistent with fault-triggering input generation being the larger test-construction bottleneck. However, FT-IO and BUGLABELONLY repair correlate at $r = 0.77$, indicating an imperfect association between their model-level rates. BDR correlates at $r = 0.87$ with BUGLABELONLY repair and at $r = 0.89$ with FT-IO.

The right panel evaluates cumulative success on aligned instance-model pairs. Starting from bug detection, each point adds the next requirement: valid input, correct expected output, discriminative input, discriminative test, and correct repair. Each drop therefore marks pairs that fail the newly added requirement after satisfying all preceding stages. Unlike model-level correlations, the resulting curves track whether success is preserved on the same pairs as the pipeline advances. A high stage-level average need not imply that the same instances remain successful downstream. This aligned view localizes observed attrition without assuming independence or introducing a separate aggregate composition estimate.

In the balanced two-model-per-family analysis, performance drops isolate where success is lost in the pipeline. The sharpest attrition occurs at *Valid IO* → *FT-Input* (9 points, fault-hypothesis formation) and *FT-IO* → *Repair* (19 points, test-to-repair conversion). Together with the paired repair flows in Finding 2, these losses show that success at one stage does not guarantee end-to-end de-

bugging success. Improving either test generation or repair alone may therefore be insufficient; both fault-hypothesis formation and the conversion of discriminative tests into correct repairs remain bottlenecks.

5 Conclusion

We isolate fault-targeted reasoning from scaffolding through a controlled study of frontier and open-source LLMs, using a multi-stage framework that decouples test generation from repair under varying diagnostic conditions. Three limitations emerge. First, models produce mostly valid inputs, but fewer are discriminative, localizing the main test-generation bottleneck to fault-hypothesis formation rather than output prediction. Second, diagnostic feedback has mixed effects on repair: internally generated tests reduce success on average, while separately generated same-model tests help some cases and hurt others depending on test quality and use. Third, success across bug detection, test construction, and repair does not consistently occur on the same instances. Overall, our multi-level analysis identifies fault-hypothesis formation and test-to-repair conversion as the largest losses. These results motivate systems that validate tests before repair and preserve evidence across stages; gains at one stage need not improve the full pipeline. Evaluations should likewise track aligned outcomes across stages rather than relying only on aggregate task averages.

6 Limitations

Our controlled study prioritizes attribution within a fixed setting over ecological validity: function-level algorithmic problems isolate fault-targeted reasoning from repository navigation, build systems, and multi-file dependencies, enabling focused analysis without directly predicting production-agent performance. The dataset contains 173 instances from 76 problems and supports controlled analysis rather than large-scale model ranking, with execution-based evaluation across 12 models and five settings. Because each prompt is sampled once, the cluster-bootstrap intervals capture variation across problems but not stochastic decoding variance. We focus on semantic bugs in near-correct, model-generated code, whose distribution may differ from human-written and production software. The study includes only known-buggy solutions, so Bug Detection Recall does not measure false positives on correct code. Input-validity checkers are validated against official tests, but this cannot establish that they encode every implicit constraint. We evaluate Python in single-turn settings, excluding multi-turn debugging, language-specific effects, and broader problem distributions. Extending the study across languages, bug sources, and interactive settings remains future work. Repository-scale extensions will also require audited specifications and test oracles because mis-scoped tests can reject functionally correct patches (OpenAI, 2026).

Potential Risks. This work could be misinterpreted as evaluating real-world autonomous coding performance, although it studies a controlled, function-level, single-turn setting without repository context. Such overgeneralization could suggest that diagnostic feedback is inherently harmful, whereas our results show that model-generated tests can help or hurt depending on their quality and how they are used. Suggested improvements to fault-hypothesis formation could also create brittle systems if applied without validation. Continued release of LiveCodeBench (Jain et al., 2024) tasks may cause future pretraining contamination, affecting longitudinal comparisons. Finally, autonomous debugging systems may inherit the observed failure modes without safeguards.

Acknowledgments

We thank the reviewers and area chairs for their constructive feedback. We would like to thank

Yingbo Zhou for useful discussions. The authors used LLM-based tools (Claude Code) to assist with manuscript editing, including rewording, restructuring sections, condensing verbose text, and typographical corrections. The authors performed all scientific work, including experimental design, implementation, analysis, and interpretation.

References

- Juan Altmayer Pizzorno and Emery D. Berger. 2025. [Coverup: Effective high coverage test generation for python](#). *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *arXiv preprint arXiv:2108.07732*.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. [Codet: Code generation with generated tests](#). *Preprint*, arXiv:2207.10397.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021a. [Evaluating large language models trained on code](#). *arXiv preprint arXiv:2107.03374*.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024a. [Teaching large language models to self-debug](#). In *ICLR*.
- Yinghao Chen, Zehao Hu, Chen Zhi, Junxiao Han, Shuiguang Deng, and Jianwei Yin. 2024b. [Chatunite: A framework for llm-based test generation](#). *Preprint*, arXiv:2305.04764.
- Yiqun T. Chen, Rahul Gopinath, Anita Tadakamalla, Michael D. Ernst, Reid Holmes, Gordon Fraser, Paul Ammann, and René Just. 2021b. [Revisiting the relationship between fault detection, test adequacy criteria, and test set size](#). In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20*, pages 237–249, New York, NY, USA. Association for Computing Machinery.
- Bhavya Chopra, Yasharth Bajpai, Param Biyani, Gustavo Soares, Arjun Radhakrishna, Chris Parnin, and Sumit Gulwani. 2024. [Exploring interaction patterns for debugging: Enhancing conversational capabilities of ai-assistants](#). *Preprint*, arXiv:2402.06229.
- Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan

- Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qing-shui Gu, and 29 others. 2025. [NL2repo-bench: Towards long-horizon repository generation evaluation of coding agents](#). *CoRR*, abs/2512.12730.
- Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. [A survey on code generation with llm-based agents](#). *arXiv preprint arXiv:2508.00083*.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. [In-coder: A generative model for code infilling and synthesis](#). In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. [Measuring coding challenge competence with APPS](#). In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*.
- Dong Huang, Jie M. Zhang, Mark Harman, Qianru Zhang, Mingzhe Du, and See-Kiong Ng. 2025a. [Benchmarking llms for unit test generation from real-world functions](#). *Preprint*, arXiv:2508.00408.
- JinYang Huang, Xiachong Feng, Qiguang Chen, Hanjie Zhao, Zihui Cheng, Jiesong Bai, Jingxuan Zhou, Min Li, and Libo Qin. 2025b. [MLDebugging: Towards benchmarking code debugging across multi-library scenarios](#). In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 5866–5879, Vienna, Austria. Association for Computational Linguistics.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fan-jia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. [Live-codebench: Holistic and contamination free evaluation of large language models for code](#). *CoRR*, abs/2403.07974.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. [SWE-bench: Can language models resolve real-world github issues?](#) In *The Twelfth International Conference on Learning Representations*.
- Mohammad Abdullah Matin Khan, M. Saiful Bari, Xuan Do Long, Weishi Wang, Md. Rizwan Parvez, and Shafiq Joty. 2024. [Xcodeeval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, pages 6766–6805. Association for Computational Linguistics.
- Hung Le, Hailin Chen, Amrita Saha, Akash Gokul, Doyen Sahoo, and Shafiq Joty. 2024. [Codechain: Towards modular code generation through chain of self-revisions with representative sub-modules](#). *Preprint*, arXiv:2310.08992.
- Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. [Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models](#). In *ICSE, ICSE '23*, pages 919–931. IEEE Press.
- Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. [Evocodebench: An evolving code generation benchmark aligned with real-world code repositories](#). *Preprint*, arXiv:2404.00599.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. [Competition-level code generation with alphacode](#). *arXiv preprint arXiv:2203.07814*.
- Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. [Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation](#). In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*, Red Hook, NY, USA. Curran Associates Inc.
- Kaibo Liu, Zhenpeng Chen, Yiyang Liu, Jie M. Zhang, Mark Harman, Yudong Han, Yun Ma, Yihong Dong, Ge Li, and Gang Huang. 2025. [Llm-powered test case generation for detecting bugs in plausible programs](#). *Preprint*, arXiv:2404.10304.
- Kaibo Liu, Yudong Han, Yiyang Liu, Zhenpeng Chen, Jie M. Zhang, Federica Sarro, Gang Huang, and Yun Ma. 2024a. [Trickybugs: A dataset of corner-case bugs in plausible programs](#). In *Proceedings of the 21st International Conference on Mining Software Repositories, MSR '24*, pages 113–117, New York, NY, USA. Association for Computing Machinery.
- Tianyang Liu, Canwen Xu, and Julian J. McAuley. 2024b. [Repobench: Benchmarking repository-level code auto-completion systems](#). In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
- Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-Tau Yih, Sida Wang, and Xi Victoria Lin. 2023. [LEVER: Learning to verify language-to-code generation with execution](#). In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 26106–26128. PMLR.
- OpenAI. 2026. [Separating signal from noise in coding evaluations](#).

- Archiki Prasad, Elias Stengel-Eskin, Justin Chih-Yao Chen, Zaid Khan, and Mohit Bansal. 2025. [Learning to generate unit tests for automated debugging](#). *Preprint*, arXiv:2502.01619.
- Md Nakhla Rafi, An Ran Chen, Tse-Hsun Chen, and Shaohua Wang. 2024. [Back to the future! studying data cleanness in defects4j and its impact on fault localization](#). *Preprint*, arXiv:2310.19139.
- Md Nakhla Rafi, Lorena Barreto Simedo Pacheco, An Ran Chen, Jinqiu Yang, Tse-Hsun, and Chen. 2025. [Sbest: Spectrum-based fault localization without fault-triggering tests](#). *Preprint*, arXiv:2405.00565.
- Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. 2025. [Swerank: Software issue localization with code ranking](#). *Preprint*, arXiv:2505.07849.
- Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Hui Haotian, Liu Weichuan, Zhiyuan Liu, and Maosong Sun. 2024. [DebugBench: Evaluating debugging capability of large language models](#). In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 4173–4198, Bangkok, Thailand. Association for Computational Linguistics.
- Hanbin Wang, Zhenghao Liu, Shuo Wang, Ganqu Cui, Ning Ding, Zhiyuan Liu, and Ge Yu. 2024. [INTERVENOR: Prompting the coding ability of large language models with the interactive chain of repair](#). In *Findings of ACL 2024*, pages 2081–2107, Bangkok, Thailand. AC:.
- Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. 2025. [Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution](#). *Preprint*, arXiv:2502.18449.
- Bingbing Wen, Jihan Yao, Shangbin Feng, Chenjun Xu, Yulia Tsvetkov, Bill Howe, and Lucy Lu Wang. 2025. [Know your limits: A survey of abstention in large language models](#). *Preprint*, arXiv:2407.18418.
- Zihan Zheng, Zerui Cheng, Zeyu Shen, Shang Zhou, Kaiyuan Liu, Hansen He, Dongruixuan Li, Stanley Wei, Hangyi Hao, Jianzhu Yao, Peiyao Sheng, Zixuan Wang, Wenhao Chai, Aleksandra Korolova, Peter Henderson, Sanjeev Arora, Pramod Viswanath, Jingbo Shang, and Saining Xie. 2025. [Live-codebench pro: How do olympiad medalists judge llms in competitive programming?](#) *CoRR*, abs/2506.11928.
- Yuxuan Zhu, Antony Kellermann, Dylan Bowman, Philip Li, Akul Gupta, Adarsh Danda, Richard Fang, Conner Jensen, Eric Ihli, Jason Benn, Jet Geronimo, Avi Dhir, Sudhit Rao, Kaicheng Yu, Twm Stone, and Daniel Kang. 2025. [Cve-bench: A benchmark for ai agents’ ability to exploit real-world web application vulnerabilities](#). *Preprint*, arXiv:2503.17332.

A Additional Study Details

A.1 Study Design & Dataset Statistics

A.1.1 Key Design Principles

The study follows three principles:

Diverse Semantic Bugs. We curate non-trivial buggy solutions that pass a subset of official tests. Bugs with higher pass rates and diverse behaviors (measured via Hamming distance) are prioritized, unlike trivially detectable syntax errors. These bugs capture genuine model reasoning failures.

Execution-Based Verification. All metrics rely on automated execution against the official test suite. A solution S passes if it produces the expected output on every test $t = (t_i, t_o)$ in the suite T , i.e., $S(t_i) = t_o$ for all $t \in T$. A *fault-triggering test* is a pair $t = (t_i, t_o)$ for which the buggy solution fails and the gold solution succeeds: $S_{\text{buggy}}(t_i) \neq t_o = S_{\text{gold}}(t_i)$.

Multi-Setting Evaluation. We evaluate models under varying information conditions—Bug-Aware versus Bug-Discovery FT-Test generation, and BUGLABELONLY versus EXTTEST or INTTEST program repair—to isolate different reasoning capabilities and measure how model-generated guidance affects performance.

A.2 Dataset Construction Details

Problem Collection. We source 1,055 function-level algorithmic problems from LiveCodeBench (Jain et al., 2024), spanning LeetCode, AtCoder, and Codeforces (May 2023–April 2025). Each problem includes a natural-language specification, official test cases with verified outputs, and a platform-assigned difficulty label. We discard problems with conflicting test cases (the same input with different outputs) or non-standard evaluation (interactive judges or approximate outputs), ensuring deterministic execution-based metrics. We apply *capability-based filtering*, excluding problems solved within three attempts by GPT-4o, Gemini-2.5-Pro, or Claude Sonnet-4. This yields 170 problems (16% retention), of which 89% are rated “medium” or “hard.” Within this filtered pool, domains include dynamic programming (28%), graph algorithms (22%), data structures (18%), greedy algorithms (15%), and others (17%).

Input-Validity Checker Generation. We prompt GPT-5-Mini to produce a Python function $\text{is_valid_test}(t_i) \rightarrow \{\text{True}, \text{False}\}$ that checks whether input t_i conforms to all constraints in

the problem specification. Figure 4 shows the prompt template and a model-generated sample. Each checker is validated against the official test suite; checkers rejecting any official input are discarded and regenerated (up to 3 attempts). This ensures no false negatives, though it does not guarantee coverage of all implicit constraints. In practice, 98.8% of problems yield valid checkers, with 100% acceptance on official tests.

Solution Collection and Verification. We collect candidate solutions from two sources: **human solutions** (accepted submissions from the official platforms, serving as high-confidence gold solutions) and **model-generated solutions** (5–10 samples per model per problem from LLMs spanning capabilities and architectures). All candidates are executed against the official test suites with time limits twice the platform limits. Solutions passing all test cases are designated **gold**; those producing a wrong answer on at least one test are designated **buggy**. We discard runtime errors, including exceptions and timeouts, to focus on semantic wrong-answer bugs. For each problem, we retain the fastest human-authored gold solution. Buggy solutions are prioritized first by official-test pass rate and then by diversity: among solutions with the same pass count, we maximize Hamming distance between their binary test-outcome vectors. Candidate-generation models span OpenAI (GPT-5-Nano, GPT-5-Mini, GPT-5, GPT-5.1, GPT-5.2), Google (Gemini-2.5-Flash-Lite, Gemini-2.5-Flash, Gemini-2.5-Pro, Gemini-3-Flash, Gemini-3-Pro), and Anthropic (Claude Opus-4/4.1/4.5 and Sonnet-4/4.5). In total, we collect 2,184 solutions from 168 filtered problems.

A.3 Per-Problem Variation

Table 6 reports uncertainty for the primary Bug-Aware discriminative-test metric. Confidence intervals use the problem-cluster bootstrap described in Section 3.5; standard deviations are computed over the 76 problem-level rates.

A.4 Experimental Setup

Models. We evaluate a diverse set of frontier and open-source language models spanning capability levels and architectures. Our evaluation includes: (1) OpenAI models (GPT-5-Nano, GPT-5-Mini, GPT-5.2, and GPT-5.2-Codex); (2) Google models (Gemini-3.1-Flash-Lite, Gemini-3-Flash, Gemini-3-Pro, and Gemini-3.1-Pro); (3) Anthropic models (Claude Sonnet-4.6 and Opus-4.6); and (4) open-

Table 5: Dataset statistics for the VIBEPASS controlled study.

Statistic	Value
Total instances (problem, gold, buggy tuples)	173
Unique problems	76
Avg. instances per problem	2.3
<i>Problem Difficulty (platform labels)</i>	
Easy	2%
Medium	25%
Hard	73%
<i>Buggy Solution Pass Rate on Official Tests</i>	
1st quartile	43%
Median	71%
3rd quartile	86%
Mean	61%
<i>Problem Domains</i>	
Array/Matrix	25%
Dynamic Programming	18%
Math/Number Theory	16%
Graph Algorithms	22%
Other	19%
<i>Bug Categories</i>	
Missing Edge Case	32%
Incorrect Condition	28%
Off-by-One Error	27%
Wrong Algorithm	8%
Others (DP Error, Incorrect Init)	5%
<i>Platforms</i>	
LeetCode	57%
AtCoder	43%

source models (Nemotron-3-Nano-30B-A3B, abbreviated Nemotron-3-Nano in the tables, and GPT-OSS-120B). For dataset construction, we use GPT-5-Mini to generate input-validity checkers and GPT-4o, Gemini-2.5-Pro, and Claude Sonnet-4 for capability-based filtering.

Inference Configuration. For all test-generation and repair tasks, we set temperature $T = 1$ for API models when available and use the highest supported reasoning effort. We set the maximum context window to 128K tokens for open-source models and use the provider maximum for API models, sufficient for problem specifications, code solutions, and generated tests.

Prompt Design. All tasks use structured prompts that include the problem description, input–output format, implementation, and task-specific instructions. Bug-Discovery prompts request both a correctness classification and, after a “buggy” prediction, a test. For EXTTEST, the same model’s Bug-Aware test is supplied in the repair prompt. For INTTEST, prompts instruct the model to generate a test within the response before attempting repair. Full templates and evaluation

Table 6: Bug-Aware D_{IO} with cluster-bootstrap 95% CIs and across-problem standard deviations.

Model	D_{IO}	95% CI	Std.
Gemini-3.1-Flash-Lite	26.0	19.1–34.2	43.3
Gemini-3-Flash	63.0	52.6–74.3	39.9
Gemini-3-Pro	69.4	59.9–78.6	38.4
Gemini-3.1-Pro	84.4	75.7–92.3	29.0
GPT-5-Nano	52.0	41.2–63.7	42.8
GPT-5-Mini	70.5	61.6–80.0	36.3
GPT-5.2	77.5	67.4–87.0	34.8
GPT-5.2-Codex	83.8	75.9–91.3	30.3
Sonnet-4.6	73.4	61.8–85.1	33.8
Opus-4.6	83.2	75.3–91.0	27.5
GPT-OSS-120B	65.3	55.0–76.2	38.7
Nemotron-3-Nano	39.3	29.9–49.7	43.5

code are available in the released repository.¹ The dataset is available on Hugging Face.²

Evaluation Protocol. All generated solutions and tests are executed in isolated sandboxes with time limits five times the original platform limits to focus evaluation on semantic correctness rather than efficiency. Generated inputs are verified using the input-validity checkers. We report Pass@1 for repair and the hierarchical FT-Test metrics for test generation, computed over the full set of evaluated instances. Each prompt is sampled once.

¹<https://github.com/SalesforceAIResearch/vibepass>

²<https://huggingface.co/datasets/Salesforce/vibepass>

Prompt

Given the following problem statement and starter code, write only the Python function `'is_valid_test'` (with the same signature as in the starter code, but without `'self'`). The function should return True if the inputs are valid according to the problem constraints, and False otherwise. Output only the code for the `'is_valid_test'` function, with no extra text or comments.

Problem Statement: You are given a 0-indexed integer array `nums`. You can perform any number of operations, where each operation involves selecting a subarray of the array and replacing it with the sum of its elements. For example, if the given array is `[1,3,5,6]` and you select subarray `[3,5]` the array will convert to `[1,8,6]`. Return the maximum length of a non-decreasing array that can be made after applying operations. A subarray is a contiguous non-empty sequence of elements within an array.

Example 1:

Input: `nums = [5,2,2]`

Output: 1

Explanation: This array with length 3 is not non-decreasing.

We have two ways to make the array length two.

First, choosing subarray `[2,2]` converts the array to `[5,4]`.

Second, choosing subarray `[5,2]` converts the array to `[7,2]`.

In these two ways the array is not non-decreasing.

And if we choose subarray `[5,2,2]` and replace it with `[9]` it becomes non-decreasing.

So the answer is 1.

Example 2:

Input: `nums = [1,2,3,4]`

Output: 4

Explanation: The array is non-decreasing. So the answer is 4.

Constraints:

$1 \leq \text{nums.length} \leq 10^5$

$1 \leq \text{nums}[i] \leq 10^5$

Starter Code:

```
class Solution:
    def findMaximumLength(self, nums: List[int]) -> int:
        pass
```

Sample Output

```
def is_valid_test(nums: list) -> bool:
    if not isinstance(nums, list):
        return False
    n = len(nums)
    if not isinstance(n, int) or n < 1 or n > 10**5:
        return False
    for x in nums:
        if not isinstance(x, int) or isinstance(x, bool):
            return False
        if x < 1 or x > 10**5:
            return False
    return True
```

Figure 4: Example prompt and corresponding model output for input-validity generation.