TokenWave

SpecEngineeringBench: Evaluating Coding Agents as Autonomous Software Test Engineers

Xingjian Wang[1], **Shijian Wang**[3], **Minghao Liu**[2], **Ji Zou**[3], **Zetao Yang**[1], **Xinping Lei**[1], **Yiming Liang**[1], **Shen Yan**[3], **Enduo Zhao**[1], **Xiaoyang Shen**[1], **Weihao Xuan**[1], **Jiaheng Liu**[3], **Ge Zhang**[1]
[1]TokenWave AI, [2]2077AI, [3]Independent Researcher

Autonomous software testing requires coding agents to translate functional requirements into executable tests and use the results to discover and validate defects. However, existing benchmarks evaluate coding agents on explicitly specified test generation tasks, commonly providing an issue description, a designated test entry point, or files to inspect, leaving the ability to determine what to test and where to investigate insufficiently evaluated . Relying on such clues bypasses the challenge of mapping requirements to implementations and proactively uncovering unknown defects. We propose **SpecEngineeringBench**, comprising 100 instances from 78 open source Python repositories across 16 application categories, with multiple recorded defects in each instance. Two independent tasks evaluate complementary capabilities: generating test suites and discovering defects through execution analysis . The evaluation combines test validity, defect detection, and coverage with localization precision and recall . Across seven frontier coding agents, results reveal that test validity and coverage decouple from defect exposure: agents achieve a 91.47% Head Pass Rate and 80.48% Change Focused Coverage alongside a 26.36% Fail-to-Pass Rate, while localization recall drops from 78.11% at the file level to 51.29% at the function level . Autonomous test generation improves file, function, and hunk localization recall by 4.99, 2.77, and 2.69 percentage points over relying solely on existing repository test suites .

Correspondence: First Author at first.author@tokenwave.ai
Date: November 21, 2026
Website: https://tokenwave-ai.github.io/SpecEngineeringBench-Page/

1 Introduction

Coding agents driven by large language models (LLMs) increasingly perform complex software engineering tasks (Yang et al., 2024; Wang et al., 2025b; Chen et al., 2026). As agents write and modify production code, software testing verifies whether implementations satisfy intended behavioral requirements (Liang et al., 2026b; Rastenis et al., 2026). In realistic software development, assessing a prerelease candidate is fundamentally an exploratory, multi-defect investigation: test engineers are not handed localized bug reports or specific failure traces; rather, they must independently interpret high level product specifications (Long et al., 2026; Liang et al., 2026b), navigate codebases, author discriminating test cases, and write reproducible defect reports. This setting motivates our central research question:

> *Given only functional specifications and an uninspected repository, can coding agents autonomously uncover, localize, and substantiate software defects with reproducible evidence?*

Existing benchmarks address this question under two structural constraints (Figure 1): (1) *Investigation scopes assigned in advance:* Unit test generation and proactive defect search suites evaluate designated

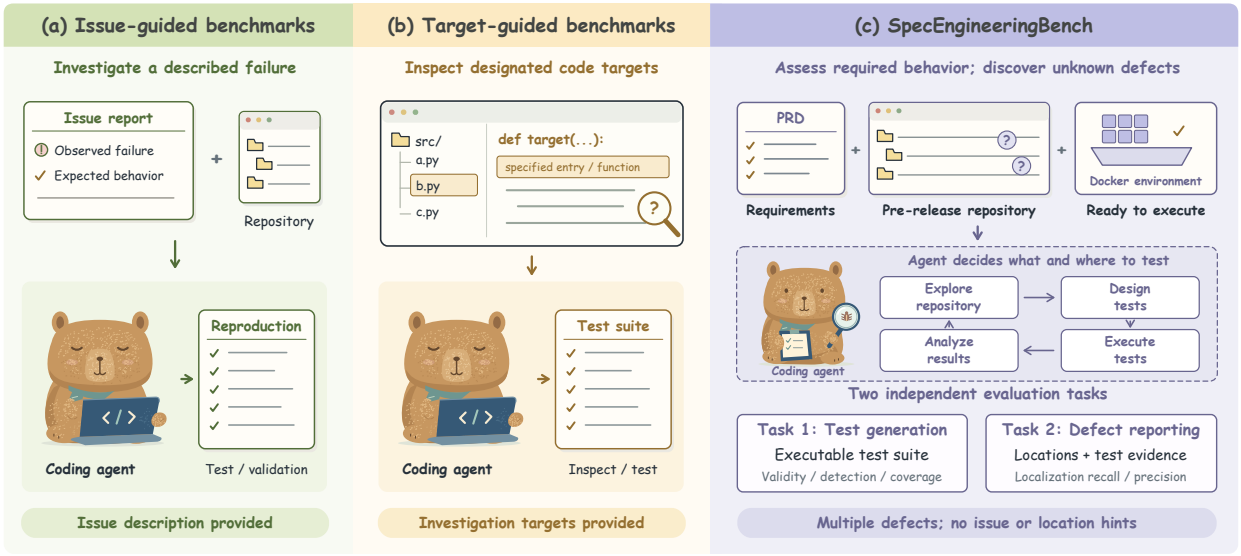


Figure 1 Conceptual comparison of representative benchmark settings and **SpecEngineeringBench**. **(a)** Issue-guided tasks provide a description of the faulty behavior (Mündler et al., 2024; Ahmed et al., 2024). **(b)** Target-guided tasks specify investigation targets, such as files, functions, or entry interfaces (Jain et al., 2025; Liu et al., 2026). **(c)** SpecEngineeringBench provides a PRD, a pre-release repository, and a configured environment, leaving agents to determine what and where to test. Agents explore the repository, design and execute tests, and analyze the results to investigate multiple unknown defects. Two independent tasks evaluate test-suite quality and defect reporting.

functions, entry interfaces, or target source files (Wang et al., 2025a; Zhang et al., 2024; Jain et al., 2025; Liu et al., 2026; Li et al., 2026). Fixing targets in advance evaluates localized code completion and omits the process of mapping functional requirements to production implementations. (2) *Reliance on issue reports and single bug assumptions:* Benchmarks driven by issue reports supply descriptions of faulty behaviors or stack traces (Jimenez et al., 2024; Mündler et al., 2024; Ahmed et al., 2024), while reproduction suites verify isolated, known bugs (Zhang et al., 2026b). Prerelease software testing involves multiple unknown defects within the same repository, requiring agents to determine which behavioral surfaces require verification.

To bridge this gap, we argue that Evaluating coding agents as autonomous test engineers requires removing location hints and issue descriptions, requiring agents to derive test scopes and defect oracles from specifications. We introduce **SpecEngineeringBench**, a benchmark of 100 instances curated from 78 open-source Python repositories across 16 application categories, embedding 531 real world defects (4–15 defects per instance). **SpecEngineeringBench** introduces four properties: **Requirements based exploration:** Agents receive only a Product Requirements Document (PRD) specifying functional capabilities, boundary conditions, and error invariants, requiring them to determine what functionality to test. **Unrestricted repository scope:** TThe benchmark supplies no designated entry points or target files, evaluating exploration across the repository. **Multiple coexisting defects:** Each instance contains multiple defects, evaluating whether agents uncover coexisting issues across the codebase. **Dual tasks:** *Task 1 (Test Generation)* evaluates whether an agent authors executable test suites; *Task 2 (Defect Discovery and Reporting)* evaluates whether an agent executes tests, isolates root causes, and substantiates findings with reproducible evidence.

Evaluating seven frontier coding agents (Anthropic, 2026b,c; OpenAI, 2026b,c; Z.ai, 2026; DeepSeek AI, 2026; Moonshot AI, 2026) across these tasks yields three findings on autonomous defect discovery: *Passing tests and broad coverage do not imply defect discrimination:* Agents achieve high Head Pass Rates (91.47%) and Change Focused Coverage (80.48%),while the Fail-to-Pass rate is 26.36%. In 67.44% of test outcomes, tests execute across both defective and repaired snapshots without detecting defects, as agents encode existing implementation outputs into assertions and omit discriminating oracles derived from the PRD. *Localization recall declines across structural granularities:* Agents attain 78.11% recall at the file level, while recall drops to 51.29% at the function level and 51.88% at the hunk level. For defects spanning multiple files, recall drops to 2.00%–7.84%, identifying multi-file causal tracing as a recurring limitation. (3) *Autonomous test generation increases defect recovery:* Agent authored tests increase file, function, and hunk localization recall by 4.99,

Table 1 Comparison of testing and software-repair benchmarks. **Exploration scope**: designated search or test scope, even when broader repository context is accessible. **Issue-free**: no issue report is supplied. **Location-hint-free**: the task guarantees no designated investigation files, functions, or entry points; issue-driven tasks may carry such clues. **Test generation**: an evaluated task or track requires agents to author or complete tests. **Bug location**: defect localization is explicitly scored. **Multi-bug**: an instance explicitly combines multiple recorded defects. **Executable evidence**: agent-generated tests are executed to substantiate defects; reference tests used only to grade repairs do not qualify. ✓ indicates that the property is present and ✗ indicates that it is absent. Independent replay of submitted report evidence is not implied.

Benchmark	Exploration scope	Issue-free	Location-hint-free	Test generation	Bug location	Multi-bug	Executable evidence
Test generation							
TestEval (Wang et al., 2025a)	Function	✓	✗	✓	✗	✗	✗
TestBench (Zhang et al., 2024)	Method / class	✓	✗	✓	✗	✗	✗
TestGenEval (Jain et al., 2025)	Source file	✓	✗	✓	✗	✗	✗
CLOVER (Xu et al., 2025)	Specified targets	✓	✗	✓	✗	✗	✗
Issue-driven reproduction and repair							
SWE-bench (Jimenez et al., 2024)	Repository	✗	✗	✗	✗	✗	✗
SWT-Bench (Mündler et al., 2024)	Repository	✗	✗	✓	✗	✗	✓
TDD-Bench Verified (Ahmed et al., 2024)	Repository	✗	✗	✓	✗	✗	✓
SWE-Cycle (Guan et al., 2026)	Repository	✗	✗	✓	✗	✗	✓
Proactive bug discovery and repair							
TestExplora (Liu et al., 2026)	Entry interfaces	✓	✗	✓	✗	✗	✓
Active-SWE (Li et al., 2026)	Review files	✓	✗	✓	✓	✓	✓
SWE-sweep (Lieret et al., 2026)	Repository	✓	✓	✗	✗	✓	✗
SpecEngineeringBench (Ours)	**Repository**	✓	✓	✓	✓	✓	✓

2.77, and 2.69 percentage points over the baseline of executing existing repository test suites, showing the effect of active test authoring on defect identification.

2 SpecEngineeringBench

SpecEngineeringBench evaluates coding agents under the operational conditions of prerelease quality assurance. Agents derive test scopes and defect oracles from product specifications without target hints or issue reports. The benchmark defines two complementary tasks: *Task 1 (Test Generation)* measures whether an agent authors valid and discriminating test suites from specifications; *Task 2 (Defect Discovery and Reporting)* measures whether executing and analyzing tests enables an agent to trace failures to root causes. Table 1 compares this benchmark with existing testing and repair suites.

2.1 Task Formulation

Each benchmark instance is formalized as a tuple $(r_i, d_i, e_i, \mathcal{B}_i)$, where r_i is the unrepaired repository snapshot, d_i is the product requirements document, e_i is the isolated Docker runtime, and $\mathcal{B}_i$ ($|\mathcal{B}_i| \geq 4$) represents the set of coexisting reference defects. The agent $\mathcal{A}$ has access only to (r_i, d_i, e_i), with no issue descriptions or target location hints provided. To separate test synthesis ability from runtime diagnostic reasoning, we establish two independent tasks:

Task 1: Test Generation (π_1). The agent generates a standalone test suite $\mathcal{T}_i^{(1)} = \mathcal{A}(r_i, d_i, e_i; \pi_1)$. The evaluation harness executes $\mathcal{T}_i^{(1)}$ against both the defective snapshot r_i and the ground truth repaired snapshot r_i^+ out of band, scoring execution validity (HP), defect discrimination rate (F2P), and functional coverage (EC, CFG).

Task 2: Defect Discovery and Reporting (π_2). The agent conducts active exploration by authoring and executing temporary test probes $\mathcal{T}_i^{(2)}$ on r_i, producing a structured diagnostic report:

$$\left(\mathcal{T}_i^{(2)}, \mathcal{R}_i\right) = \mathcal{A}(r_i, d_i, e_i; \pi_2). \tag{1}$$

The submission $\mathcal{R}_i$ must substantiate each suspected defect with reproducible execution logs and source-grounded locations, scored against reference defects $\mathcal{B}_i$ across file, function, and edit-hunk granularities.

2.2 Evaluation Protocol

The evaluation protocol assesses two capabilities: generating test suites that cover specified functionality and check correctness, and discovering defects to report exact source locations. We evaluate test validity, defect detection, and coverage in Task 1, and localization recall and precision in Task 2.

Test validity and defect detection. For instance i, let $\mathcal{T}_i$ be the registered submitted tests, $\mathcal{H}_i$ the subset passing on the repaired snapshot r_i^+, and $\mathcal{F}_i$ the subset failing on the defective snapshot r_i and passing on r_i^+. We compute Head Pass Rate (HP) by pooling test counts across the M scored instances. Fail-to-Pass Rate (F2P) averages the fraction of discriminating tests within each instance:

$$\mathrm{HP} = \frac{\sum_{i=1}^{M} |\mathcal{H}_i|}{\sum_{i=1}^{M} |\mathcal{T}_i|}, \qquad \mathrm{F2P} = \frac{1}{M} \sum_{i=1}^{M} \frac{|\mathcal{F}_i|}{|\mathcal{T}_i|}. \tag{2}$$

HP checks compatibility with the repaired implementation; F2P requires a failure that disappears after repair, confirmed by repeated execution on both snapshots. F2P measures the share of generated tests that expose a defect. We report $\mathrm{Num} = M^{-1} \sum_i |\mathcal{T}_i|$ as the mean submitted suite size.

Functional and change coverage. Entry Coverage (EC) measures how broadly tests exercise the target functionality, while Change focused Coverage (CFG) measures whether they reach the executable lines changed by the reference fixes. Let $\mathcal{E}_i$ denote executable file–line pairs in the reference coverage ranges, Δ_i executable changed file–line pairs, and $\mathcal{C}_i$ the union covered by the submitted suite, all on r_i^+. We average the coverage ratios across M evaluated instances with valid environments:

$$\mathrm{EC} = \frac{1}{M} \sum_{i=1}^{M} \frac{|\mathcal{C}_i \cap \mathcal{E}_i|}{|\mathcal{E}_i|}, \qquad \mathrm{CFG} = \frac{1}{M} \sum_{i=1}^{M} \frac{|\mathcal{C}_i \cap \Delta_i|}{|\Delta_i|}. \tag{3}$$

The suite level union counts each covered line once, and the outer mean gives each instance equal weight. These coverage targets remain evaluator-only; agents must identify the implementation from the requirements.

Defect localization. Localization Recall (LR) measures the proportion of reference defect locations recovered by a report, while Localization Precision (LP) penalizes unsupported locations. For granularity $g \in \{\mathrm{file}, \mathrm{function}, \mathrm{hunk}\}$, let G_i^g and P_i^g denote the reference and predicted location sets. With matched reference count a_i^g and credited prediction count b_i^g, we compute

$$\mathrm{LR}_i^g = \frac{a_i^g}{|G_i^g|}, \qquad \mathrm{LP}_i^g = \frac{b_i^g}{|P_i^g|}. \tag{4}$$

Files match by normalized paths; functions additionally require matching names. For hunks, a reported line group matches a reference edit when sharing a file and containing source lines within $\tau = 3$ lines of each other. Hunk matching uses defective snapshot coordinates, with source line anchors for insertion edits; empty localization denominators evaluate to zero.

Distinct bug localization. We measure the percentage of evaluable reference bugs successfully localized across all scored instances. Each source pull request defines one bug occurrence. A bug is localized when the report matches all reference files, function symbols, and more than half of the reference edit hunks. Function

Table 2 SpecEngineeringBench statistics by recorded defects per instance. **Repositories**: distinct upstream projects within each group; a project can occur in multiple groups, so the total is deduplicated. **Occurrences**: recorded defects counted separately in each instance. **Source kLOC**: non-documentation source code in the actual defective snapshots, excluding comments and blank lines but retaining tests, dependencies, and generated code (1 kLOC = 1,000 lines). Source kLOC, reference files, and reference edit hunks are shown as median (range); files and hunks are the evaluator's localization targets.

Defects / instance	Instances	Repositories	Occurrences	Source kLOC	Ref. files	Ref. hunks
4–5	74	62	341	73.9 (1.9–1,966.1)	5 (1–21)	10.5 (2–71)
6–9	24	20	162	102.3 (5.6–1,798)	7.5 (4–12)	19 (5–63)
10–15	2	2	28	147.5 (11.9–283)	8.5 (4–13)	21 (14–28)
Total	**100**	**78**	**531**	**80.4 (1.9–1,966.1)**	**5 (1–21)**	**11.5 (2–71)**

symbols include `<module>` for edits outside functions. Let $\mathcal{B}_i$ denote bugs with evaluable reference mappings in scored instance i, and z_{ib} indicate whether bug b satisfies these three conditions. We compute

$$\text{Bugs Localized } (\%) = 100 \times \frac{\sum_{i=1}^{M} \sum_{b \in \mathcal{B}_i} z_{ib}}{\sum_{i=1}^{M} |\mathcal{B}_i|}. \tag{5}$$

We apply the same calculation to bugs involving a single file or multiple files, and report the percentage of scored instances in which every evaluable bug is localized. Our success criterion evaluates submitted defect locations; SWE-sweep verifies repairs through hidden tests and checks for regressions.

2.3 Dataset Characteristics

SpecEngineeringBench comprises 100 instances from 78 open-source Python repositories spanning 16 application categories, with 531 recorded defect occurrences (Table 2). Each instance contains 4–15 defects, with a median of 5 and a mean of 5.31. To describe the extent of multi-defect assessment, Table 2 groups instances by recorded defect count: (1) **4–5 defects** (74 instances); (2) **6–9 defects** (24 instances); and (3) **10–15 defects** (2 instances).

Repository scope and requirements. The 78 repositories represent 16 application categories under the repository inventory's classification. Figure 2 shows the instance distribution across these categories, instance-level defect counts, and localization scopes. Their defective snapshots have a median of 80,449.5 non-documentation source lines (Table 2). Each instance pairs its snapshot with a product requirements document (PRD) describing the specified capabilities, expected results, boundary conditions, and error behavior, together with a Docker execution environment. The PRD specifies what should work; the agent must determine where and how to test it. Submitted suites distinguish unit, integration, and acceptance tests to exercise components, interactions, and observable behavior.

Defect scope and reference annotations. The recorded defects span a median of 5 reference files and 11.5 edit hunks per instance, requiring reports to identify multiple locations within the same repository. Our analysis covers 17 observed defect types in three families: **incomplete feature implementation** (5 types), **incorrect feature implementation** (7 types), and **generic programming errors** (5 types). The reference metadata links each recorded defect to its source PR, fix, regression tests, and source locations, and supplies the repaired snapshot and executable coverage ranges used in Section 2.2.

3 Experiments

We evaluate frontier coding agents on **SpecEngineeringBench** across five core research questions: **RQ1** assesses the quality of tests generated for the specified functionality and agents' ability to localize defects through testing; **RQ2** measures the proportion of distinct reference bugs localized within each instance; **RQ3** investigates whether agents are more effective at discovering certain defect types than others; **RQ4** examines

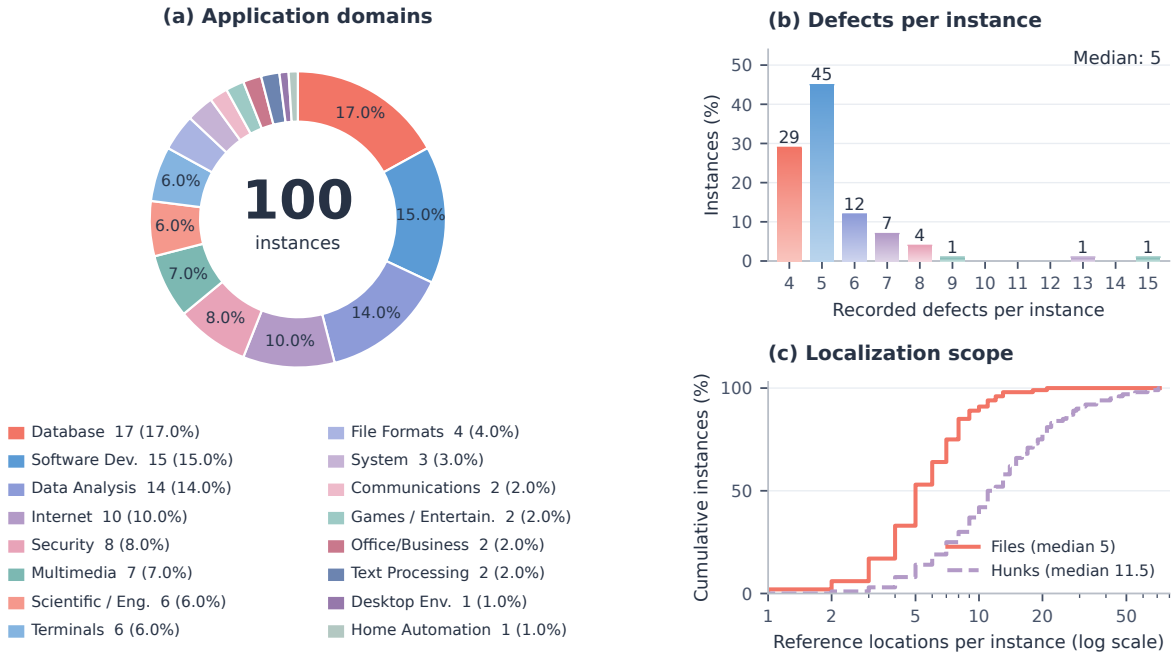


Figure 2 Dataset composition and localization scope across all 100 instances. (a) Application-domain shares; the percentages and counts sum to 100 instances, drawn from 78 distinct repositories. (b) The distribution of recorded defects per instance; because there are 100 instances, bar counts equal percentages. (c) Empirical cumulative distributions across instances for the number of reference files (solid line) and reference edit hunks (dashed line); the horizontal axis is logarithmic and the vertical axis is the percentage of instances at or below each scope. Defect occurrences are counted separately in each instance.

the mechanisms behind model failures in test construction and defect localization; and **RQ5** compares defect localization using repository-provided and agent-authored test suites.

3.1 Experimental Setup

Foundation models and agent harnesses. We evaluate frontier models across native production harnesses: Anthropic Claude (Opus 5 (Anthropic, 2026b), Sonnet 5 (Anthropic, 2026c) via Claude Code (Anthropic, 2026a)), OpenAI GPT (GPT 6-Astra (OpenAI, 2026c), GPT 5.6-Sol (OpenAI, 2026b) via Codex (OpenAI, 2026a)), Moonshot Kimi (Kimi-K3 (Moonshot AI, 2026) via Claude Code (Anthropic, 2026a)), DeepSeek (DeepSeek-V4.1-Flash (DeepSeek AI, 2026) via Claude Code), and Z.ai GLM (GLM-5.3 (Z.ai, 2026) via Claude Code) and under vendor recommended production configurations at high reasoning effort.

Container execution and environment isolation. Benchmark tasks are packaged and verified using the standardized Harbor (v0.23.0) containerized framework (Harbor Team, 2026), ensuring out-of-band verifier execution. Trials execute in isolated containers with outbound networking disabled, initialized from pinned shallow git clones on dedicated CPU cores with strict memory limits.

3.2 RQ1: Test Validity Decouples from Defect Detection

Table 3 Main results on SpecEngineeringBench. Task 1 evaluates test authoring and Task 2 evaluates defect reporting; HP pools test counts across scored instances; all other percentages and Num are averaged across scored instances. HP is Head Pass Rate (the fraction of registered tests that pass on the repaired snapshot), F2P is Fail-to-Pass Rate (the fraction that fail on the defective snapshot and pass on the repaired snapshot), EC is Entry Coverage (the fraction of executable target-functionality lines covered), and CFG is Change focused Coverage (the fraction of executable reference changed lines covered). LR and LP are localization recall and precision, computed over the repository, at file, function, and hunk granularity; hunk matches use a tolerance of $\tau = 3$ lines (see Section 2.2). Num is the mean number of submitted test cases; all other metric values are percentages. For each percentage metric, the top three distinct values are shaded from darkest (best, bold) to lightest; ties share a shade. Num reports suite size.

Model	Harness	Task 1: Test Authoring					Task 2: Defect Reporting					
							File		Function		Hunk	
		HP	F2P	EC	CFG	Num	LR	LP	LR	LP	LR	LP
Open-Source Models												
DeepSeek-V4.1-Flash (DeepSeek AI, 2026)	Claude Code	92.25	27.69	71.29	76.89	86.88	76.97	76.09	48.26	55.15	51.36	71.20
Kimi-K3 (Moonshot AI, 2026)	Claude Code	93.48	25.00	77.53	81.20	120.50	80.64	76.26	51.60	54.94	51.98	69.72
GLM-5.3 (Z.ai, 2026)	Claude Code	**94.40**	21.33	76.64	80.58	142.02	79.78	75.42	49.39	54.92	51.57	68.73
Closed-Source Models												
Claude-Sonnet-5 (Anthropic, 2026c)	Claude Code	90.65	21.78	68.23	72.88	68.74	62.89	73.82	40.08	50.49	41.20	66.33
Claude-Opus-5 (Anthropic, 2026b)	Claude Code	93.16	27.07	79.49	83.39	197.24	80.80	**78.85**	56.55	54.63	55.11	**71.90**
GPT-5.6-Sol (OpenAI, 2026b)	Codex	86.02	**34.09**	72.41	81.41	32.91	79.80	75.26	52.59	**58.55**	52.22	70.34
GPT-6-Astra (OpenAI, 2026c)	Codex	90.37	27.56	**81.41**	**87.01**	162.62	**85.85**	71.59	**60.56**	47.72	**59.72**	63.27

Execution validity converts into defect exposure at low rates. Table 3 shows a numerical gap between test validity and defect exposure: mean HP reaches 91.47% across seven agents, while mean F2P is 26.36%. In an inspection of 49,317 registered tests from 500 evaluations across five systems, 67.44% pass on the repaired snapshot without failing on the defective snapshot, 24.17% satisfy F2P, and 8.39% fail on the repaired snapshot. The primary outcome is a test that executes successfully while leaving defective behavior unexposed. Across these evaluations, 5.60% contain passing repaired snapshot tests with zero F2P tests. Tests must distinguish supported behavior from deviations present in the defective implementation. Cases 1 and 3 demonstrate this pattern: broad passing suites leave explicit options or boundary contracts unchecked (see also Astra Cases 4–6).

Table 4 F2P by Change-focused Coverage (CFG) interval for all seven models, each evaluated on 100 instances. Each entry is the mean per-instance F2P within the indicated CFG interval; both CFG and F2P are expressed as percentages. Intervals include their lower endpoint and exclude their upper endpoint, except the final interval also includes 100. N/A denotes an empty interval. The lowest-coverage interval contains at most three instances per model.

Model	Change-focused Coverage				
	0–20	20–40	40–60	60–80	80–100
F2P					
DeepSeek-V4.1-Flash	–	19.19	32.15	22.96	30.06
Kimi-K3	0.00	32.54	21.68	19.78	26.75
GLM-5.3	–	21.49	15.47	12.25	24.40
Claude-Sonnet-5	1.41	24.32	16.15	19.81	24.24
Claude-Opus-5	0.00	29.80	23.49	18.42	29.90
GPT-5.6-Sol	2.00	32.05	18.80	32.98	37.82
GPT-6-Astra	1.12	7.35	22.26	26.52	29.77

Reaching patched code does not ensure behavioral discrimination. Across seven agents, EC averages 75.29% and CFG averages 80.48%, indicating broad execution of target functionality and reference fix regions. Table 4 groups each model's instances by CFG. Even in the 80–100% coverage interval, mean F2P ranges from 24.24% to 37.82% across models. Executing most reference changed lines therefore does not ensure that submitted tests distinguish the defective and repaired snapshots. Tests can execute relevant lines without supplying defect-triggering inputs or checking the affected behavior. Cases 1 and 2 illustrate requirements-derived assertions exposing behavior left unchecked by repository tests, while Astra Case 5 provides a Task 1 example of assertions that distinguish the two snapshots.

Defect reporting loses recall at finer localization granularity. Task 2 measures agreement with reference defect locations across file, function, and hunk granularities. Mean Localization Recall (LR) drops by 26.81 percentage points from the file level (78.11%) to the function level (51.29%), and measures 51.88% at the hunk level. Localization Precision (LP) drops by 21.56 percentage points from the file level (75.33%) to the function level (53.77%) and rebounds to 68.78% at the hunk level under the $\tau = 3$ line tolerance. Agents identify relevant source files; once isolating a defective function, they pinpoint the edit hunk with minimal recall loss. Between these stages, agents face difficulty isolating root cause functions among surrounding callers and helpers within identified files.

3.3 RQ2: Agents Leave Substantial Gaps in Distinct Bug Localization

Table 5 Distinct bug localization in Task 2. A bug is localized when all reference files and function (including `<module>`) are matched and hunk recall exceeds 50%, with a tolerance of three lines. Bugs Localized is the number of localized reference bugs divided by the number of evaluable reference bugs across scored instances, with each bug occurrence weighted equally. Overall includes all evaluable bugs; Single file bugs and Multi file bugs apply the same calculation to bugs whose reference locations span one file or multiple files, respectively.

Model	Overall	Single file bugs	Multi file bugs
Open-Source Models			
DeepSeek-V4.1-Flash	37.93	46.19	3.92
Kimi-K3	42.72	52.14	3.92
GLM-5.3	42.34	51.90	2.94
Close-Source Models			
Claude-Sonnet-5	33.14	40.63	2.00
Claude-Opus-5	45.37	54.57	7.84
GPT-5.6-Sol	43.30	52.38	5.88
GPT-6-Astra	49.04	59.05	7.84

Agents localize less than half of reference bugs. Across evaluated frontier models, the percentage of reference bugs successfully localized ranges from 33.14% to 49.04% (Table 5). GPT-6-Astra attains 85.85% recall at the file level (Table 3), while localizing 49.04% of distinct reference bugs across scored instances . Identifying an affected file through test execution does not isolate the underlying defective functions and code modifications. Authoring tests that trigger visible runtime exceptions can expose repository files without covering the specific boundary inputs required to localize the full defect.

Defects spanning multiple files show lower localization. Localization rates depend on bug scope. Across models, agents localize 40.63%–59.05% of bugs located within a single file, while localization rates decrease to 2.00%–7.84% for bugs spanning multiple files (Table 5). Multi file defects involve contract mismatches and state transformations across component boundaries . Agents primarily write unit tests for isolated functions, producing test suites that omit execution paths traversing multiple modules . When a test fails, the agent investigates the immediate exception frame, leaving callers and dependent components uninspected.

Distribution of localized defects across instances. In instance level evaluations, agents localize every evaluable bug in 4.00% to 11.00% of instances. In 70.71% to 88.89% of instances, submissions contain a

combination of localized and unlocalized defects. After localizing at least one bug in an instance, GPT-6-Astra leaves remaining coexisting bugs unlocalized in 83.00% of instances. Exploration frequently halts after confirming an initial failure, leaving coexisting defects in the repository undetected.

3.4 RQ3: Discovery Capabilities Skew Across Defect Types

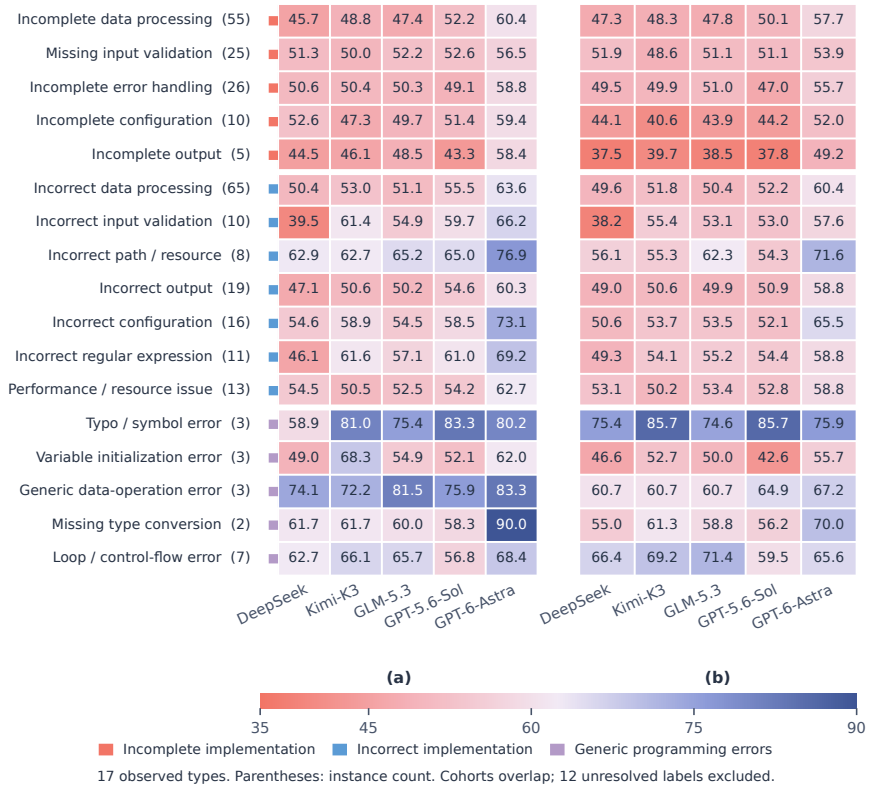


Figure 3 Instance averaged localization recall (%) by defect type cohort for the five evaluated systems with existing defect-type annotations. Panel (a) reports function-level recall and panel (b) reports hunk-level recall; columns within each panel are DeepSeek-V4.1-Flash, Kimi-K3, GLM-5.3, GPT-5.6-Sol, and GPT-6-Astra. Parentheses give the number of instances in each cohort; multiple defects of the same type within one instance do not increase its weight. These instances contain 442 recorded defect occurrences, of which 430 have type labels; 12 unresolved labels are excluded from cohort membership. The 14 replacement instances await type annotation.

Incomplete behaviors create shared blind spots across models. Figure 3 shows low recall across several recurring defect categories. In the *Incomplete data processing* cohort (55 instances), function and hunk LR average 50.88% and 50.23% across the five models. The *Missing input validation* cohort (25 instances) yields hunk LR between 48.64% and 53.87%. Output cohorts also show low recall, with mean hunk LR of 51.83% for *Incorrect output* (19 instances) and 40.56% for *Incomplete output* (five instances). Weaknesses recur across data transformations, validation routines, and output serialization, indicating that uncompleted behaviors challenge testing across all evaluated agents.

Localization accuracy within identified files varies across defect types. Conditioning on complete file recall isolates localization failures within known files. Across the five models, 46 model instance records in the *Missing input validation* cohort identify every reference file; 95.65% omit reference functions and 93.48% omit reference hunks, with mean function and hunk LR at 66.51% and 60.71%. In the *Incorrect configuration* cohort, 89.58% and 91.67% of the 48 records with complete file recall omit functions and hunks. In the *Incorrect regular expression* cohort, omission rates drop to 44.00% across 25 complete file records, where mean function LR reaches 90.57%. Discovering all relevant files leaves differing residual gaps across defect types, with input validation and configuration requiring more specific inspection within discovered files.

Table 6 Localization recall (LR) and precision (LP) across defect-type cohorts for the five evaluated systems on the 86 annotated instances retained in the current benchmark. LR is the percentage of reference locations recovered, and LP is the percentage of reported locations credited by the evaluator, computed at file, function, and hunk granularity. Each entry is the mean over all eligible model–instance records in the cohort at the indicated granularity; cohorts overlap and each instance is evaluated against its complete reference set.

Defect-type	File		Function		Hunk	
	LR	LP	LR	LP	LR	LP
Incomplete implementation						
Incomplete data processing	76.83	72.03	50.88	50.58	50.23	65.70
Missing input validation	81.87	74.95	52.54	53.59	51.32	70.36
Incomplete error handling	76.51	73.38	51.85	53.88	50.62	68.62
Incomplete configuration	73.97	80.30	52.08	57.37	44.95	69.66
Incomplete output	71.89	76.13	48.17	59.59	40.56	65.26
Incorrect implementation						
Incorrect data processing	79.04	75.09	54.71	54.28	52.89	67.95
Incorrect input validation	81.89	76.85	56.35	50.37	51.46	69.45
Incorrect path / resource	87.62	72.00	66.52	48.97	59.95	66.81
Incorrect output	76.08	71.71	52.54	56.16	51.83	66.54
Incorrect configuration	85.65	76.06	59.89	49.60	55.08	67.65
Incorrect regular expression	76.12	79.55	59.00	58.41	54.38	71.53
Performance / resource issue	81.15	84.40	54.87	62.57	53.67	82.28
Generic programming errors						
Typo / symbol error	87.69	73.58	75.75	50.24	79.46	69.34
Variable initialization error	81.43	85.38	57.28	63.88	49.54	76.01
Generic data-operation error	98.89	84.95	77.41	72.67	62.84	80.66
Missing type conversion	80.00	66.36	66.33	48.73	60.25	58.02
Loop / control-flow error	94.40	73.82	63.92	62.32	66.45	70.28

Cohort recall metrics encompass omissions and uncredited predictions. The *Performance / resource issue* and *Incomplete data processing* cohorts yield similar mean hunk LR (53.67% and 50.23%), while their reports combine omissions and uncredited predictions differently (Table 6). In the performance cohort, 35.38% of reports attain complete hunk precision while omitting reference hunks, recovering 55.10% of reference hunks on average. In the incomplete data processing cohort, 81.45% of reports simultaneously omit reference hunks and include uncredited predictions, compared with 56.92% in the performance cohort. Recovering these defects requires wider reference coverage alongside selective candidate reporting.

3.5 RQ4: Behavioral Failure Modes in Test Construction and Localization

Existing implementation behaviors replace specification based oracles. Assertion mismatches account for 31.6% of reviewed Task 1 failure mechanisms, and misconfigured mocks account for 37.5%; invalid input or setup accounts for 7.4%, API or parameter errors for 22.8%, syntax or collection failures for 1.5%, and runtime exceptions for 3.7%. Inspection of execution trajectories indicates that agents infer test oracles from repository outputs, default fixtures, and existing return patterns, leaving requirements from the PRD unencoded in assertions. The resulting test suites execute cleanly while validating existing defective behaviors, failing to produce assertions that distinguish defective implementations from repaired ones.

Causal tracing halts at initial failure locations. Among reviewed localization failure mechanisms, premature search termination accounts for 36.0%, evidence underuse for 30.9%, and dependency truncation for 25.0%. These failure modes reflect a shared pattern: upon observing a test failure or exception traceback, an agent anchors on the first visible error frame or entry point and halts exploration. Agents omit inspection of upstream caller hierarchies, cross file data flows, and execution logs needed to isolate the root cause. Submitted defect reports record immediate symptom locations while leaving the underlying defect path unexamined.

Defensive reporting trades completeness for precision. Reporting at an incorrect granularity accounts for 4.4% of reviewed localization failure mechanisms, while conservative omissions account for 19.9% and identifying symptoms as causes accounts for 25.0%. Faced with causal ambiguity, agents employ two reporting strategies: they report coarse scopes by naming test drivers, entry interfaces, or configuration files in place of production code, or they omit plausible candidate locations to prevent uncredited predictions from reducing precision scores. This pruning produces sparse reports that omit confirmed defect locations and reduce the completeness of findings.

3.6 RQ5: Autonomous Testing Improves Defect Recall Beyond Existing Suites

We compare GPT-5.6-Sol with Codex on the same 100 instances. With *Repository-provided tests*, the agent executes the existing test suite; with *Agent-authored tests*, it authors tests from the PRD. Table 7 compares localization recall and precision using the same paired reports and reference annotations, giving each instance equal weight.

Table 7 Defect localization with repository-provided and agent-authored tests for GPT-5.6-Sol with Codex on the same 100 instances. LR and LP denote localization recall and precision, respectively. Values are instance-averaged percentages. Green upward and red downward annotations show changes in percentage points, computed before rounding.

Test suite	File		Function		Hunk	
	LR	LP	LR	LP	LR	LP
Repository-provided tests	74.81	78.00	49.82	60.16	49.53	73.59
Agent-authored tests	79.80 ↑+4.99	75.26 ↓-2.74	52.59 ↑+2.77	58.55 ↓-1.60	52.22 ↑+2.69	70.34 ↓-3.25

Agent authored testing increases average recall across a subset of instances. Table 7 reports GPT-5.6-Sol results across 100 instances under two execution conditions: running repository provided tests and authoring tests from the PRD . Authoring tests increases instance averaged recall by 4.99 percentage points for files, 2.77 percentage points for functions, and 2.69 percentage points for hunks, while mean precision decreases by 2.74, 1.61, and 3.25 percentage points (differences computed before rounding) . These recall gains concentrate in specific instances: function recall improves in 31% of instances and hunk recall improves in 35%, while the median paired change across all three granularities is zero . Authoring tests improves defect coverage in instances where additional tests support further discovery .

Recall gains expand partial discoveries through broader file identification. Among instances with improved function and hunk recall, 87.10% and 91.43% obtain nonzero recall under repository provided tests, indicating that autonomous testing expands partial discoveries . Higher file recall accompanies 70.97% of function improvements and 68.57% of hunk improvements . Authoring tests broadens the set of affected files identified while recovering additional functions and hunks . Cases 1–3 in Appendix D illustrate how the two testing settings produce different evidence for defect investigation .

Report growth exceeds matched locations and reduces precision. Agent authored testing increases reported function locations from 868 to 1,011 (+16.47%), while recovered reference functions increase from 521 to 572 (+9.79%) . Reported hunks increase from 527 to 608 (+15.37%), while recovered reference hunks increase from 572 to 626 (+9.44%) . Total reported locations increase at a higher rate than confirmed defect sites, which lowers mean function and hunk precision by 1.60 and 3.25 percentage points (Table 7) . Among instances with improved recall, 51.61% at the function level and 40.00% at the hunk level show lower precision . Agent authored tests support broader discovery, while selecting candidate locations to report remains a constraint on precision .

4 Related Work

Coding Agents. Agents based on large language models (Yang et al., 2024; Wang et al., 2025b; Chen et al., 2026) use tool environments to search repositories, edit source files, and execute commands over multiple turns.

Existing benchmarks evaluate code generation (Liang et al., 2026a), repository level issue resolution (Jimenez et al., 2024), complex feature implementation (Zhou et al., 2026), and lifecycle workflows (Guan et al., 2026); SWE-smith (Yang et al., 2026) expands training distributions for software engineering agents. Executing existing tests evaluates command execution without measuring an agent's ability to determine which behaviors require testing (Shang et al., 2026; Chen et al., 2026). **SpecEngineeringBench** evaluates this test design capability under the configurations specified in Section 3.1.

Benchmarks for Test Generation. Automated test generation includes regression testing (Rastenis et al., 2026), search methods (Deljouyi et al., 2026), and iterative generation with execution feedback (Zhang et al., 2026a; Xu et al., 2026; Rastenis et al., 2026). Existing benchmarks evaluate agents on specified targets: TestEval (Wang et al., 2025a), TestBench (Zhang et al., 2024), and TestGenEval (Jain et al., 2025) isolate individual functions, classes, and source files, while CLOVER (Xu et al., 2025) examines coverage on specified components. Issue based benchmarks such as SWT-Bench (Mündler et al., 2024) and TDD-Bench Verified (Ahmed et al., 2024) evaluate translating issue descriptions into reproduction tests (Zhang et al., 2026b; Guo et al., 2026; Fei et al., 2026). In proactive defect discovery, TestExplora (Liu et al., 2026) supplies entry interfaces, Active-SWE (Li et al., 2026) restricts inspection to review files, and SWE-sweep (Lieret et al., 2026) grades patches without evaluating executable test evidence. Coverage and execution metrics can overlook subtle defects (Xue et al., 2026; Zhao et al., 2026; Ahmed et al., 2026). **SpecEngineeringBench** evaluates requirements based testing on prerelease repositories with multiple coexisting defects.

5 Conclusion

We introduce **SpecEngineeringBench**, a benchmark for evaluating coding agents as autonomous software test engineers through requirements-driven testing of repositories containing multiple real-world defects. Its two independent tasks assess generated test quality and defect localization without supplying issue descriptions or investigation targets. Initial evaluations reveal a gap between test validity and defect detection: most submitted tests pass on repaired implementations, while substantially fewer expose defects. Higher coverage and larger test suites do not consistently translate into better defect detection or more precise localization, so each capability requires direct assessment. Relying solely on a repository's existing test suite reveals fewer defects than allowing agents to explore the repository and generate additional tests, demonstrating the value of active investigation beyond existing test coverage. Advancing autonomous test engineering requires stronger test design grounded in requirements, investigation guided by execution, and validation of reproducible defect evidence. SpecEngineeringBench provides an empirical foundation for studying these capabilities, with future extensions to additional languages and independent verification of reported evidence.

References

Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. Tdd-bench verified: Can llms generate tests for issues before they get resolved? *arXiv preprint arXiv:2412.02883*, 2024.

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Investigating test overfitting on swe-bench. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering*, pages 1292–1296, 2026.

Anthropic. Claude Code: Overview, 2026a. https://code.claude.com/docs/en/overview. Official documentation. Accessed October 8, 2026.

Anthropic. Introducing Claude Opus 5, July 2026b. https://www.anthropic.com/news/claude-opus-5. Official model announcement, July 24, 2026.

Anthropic. Introducing Claude Sonnet 5, June 2026c. https://www.anthropic.com/news/claude-sonnet-5. Official model announcement, June 30, 2026.

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

DeepSeek AI. DeepSeek-V4.1-Flash model card, 2026. https://huggingface.co/deepseek-ai/DeepSeek-V4.1-Flash. Official model card. Accessed October 15, 2026.

Amirhossein Deljouyi, Annibale Panichella, and Andy Zaidman. Enhancing automated unit test generation for nlp libraries using large language models. *arXiv preprint arXiv:2609.14784*, 2026.

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. *arXiv preprint arXiv:2603.07326*, 2026.

Hao Guan, Lingyue Fu, Shao Zhang, Yaoming Zhu, Kangning Zhang, Lin Qiu, Xunliang Cai, Xuezhi Cao, Weiwen Liu, Weinan Zhang, et al. Swe-cycle: Benchmarking code agents across the complete issue resolution cycle. *arXiv preprint arXiv:2605.13139*, 2026.

Yaoqi Guo, Yang Liu, Jie M Zhang, Yun Ma, Yiling Lou, and Zhenpeng Chen. Swe-doctor: Guiding software engineering agents with runtime diagnosis from multi-faceted bug reproduction tests. *arXiv preprint arXiv:2607.00990*, 2026.

Harbor Team. Harbor: Task format overview, 2026. https://docs.harborframework.com/tasks/overview. Official documentation. Accessed October 8, 2026.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark. In *International Conference on Learning Representations*, volume 2025, pages 14947–14999, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.

Haobin Li, Ping Deng, Weizhong Qian, Liang Jiang, Zhenyu Huang, Mouxing Yang, and Xi Peng. Active-swe: Benchmarking coding agents for proactive bug fixing without issue reports. *arXiv preprint arXiv:2608.04682*, 2026.

Yunhao Liang, Chengguang Gan, Ruixuan Ying, Hanjun Wei, Zhe Cui, and Shiwen Ni. Do code language models follow tests? paired interventions on program behavior. *arXiv preprint arXiv:2607.26244*, 2026a. https://arxiv.org/abs/2607.26244.

Yunhao Liang, Chengguang Gan, Ruixuan Ying, Hanjun Wei, Zhe Cui, and Shiwen Ni. Test-driven approaches to software engineering with large language models: A survey of phases, tasks, and agent skills. *arXiv preprint arXiv:2609.12012*, 2026b.

Kilian Lieret, Jeffrey Jian Ma, Rahul Kindi, Yuxiang Wei, Jeremy Ma, Sten Sootla, Parth Thakkar, Chao Beyond Zhou, Pengcheng Yin, Rui Hou, Ofir Press, and John Yang. SWE-sweep: Can agents autonomously find and fix bugs?, 2026. https://swesweep.com/paper. Preprint.

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. *arXiv preprint arXiv:2602.10471*, 2026.

Minghui Long, Yanjie Zhao, and Haoyu Wang. Logichunter: Testing llm agent frameworks with an agentic oracle. *Proceedings of the ACM on Software Engineering*, 3(ISSTA):1171–1193, 2026.

Moonshot AI. Kimi K3 is here: Our most capable model, July 2026. https://forum.moonshot.ai/t/kimi-k3-is-here-our-most-capable-model/480. Official model announcement, July 22, 2026.

Niels Mündler, Mark N Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. *Advances in Neural Information Processing Systems*, 37:81857–81887, 2024.

OpenAI. Codex CLI, 2026a. https://learn.chatgpt.com/docs/codex/cli. Official documentation. Accessed October 8, 2026.

OpenAI. GPT-5.6 Sol model, 2026b. https://developers.openai.com/api/docs/models/gpt-5.6-sol. Official model documentation. Accessed October 8, 2026.

OpenAI. GPT-6 Astra model, 2026c. https://developers.openai.com/api/docs/models/gpt-6-astra. Official model documentation. Accessed October 8, 2026.

Matas Rastenis, Ben Chou, Shauvik Roy Choudhary, and René Just. Automated software test generation at industry scale using a multi-agent architecture and workflow integration. In *Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice*, pages 808–818, 2026.

Ye Shang, Quanjun Zhang, Haichuan Hu, Chunrong Fang, Liang Xiao, and Zhenyu Chen. Breaking, stale, or missing? benchmarking coding agents on project-level test evolution. *arXiv preprint arXiv:2605.06125*, 2026.

Wenhan Wang, Chenyuan Yang, Zhijie Wang, Yuheng Huang, Zhaoyang Chu, Da Song, Lingming Zhang, An Ran Chen, and Lei Ma. Testeval: Benchmarking large language models for test case generation. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 3547–3562, 2025a.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919, 2025b.

Jiacheng Xu, Bo Pang, Jin Qu, Hiroaki Hayashi, Caiming Xiong, and Yingbo Zhou. CLOVER: A test case generation benchmark with coverage, long-context, and verification. *arXiv preprint arXiv:2502.08806*, 2025. https://arxiv.org/abs/2502.08806.

WeiZhe Xu, Mengyu Liu, and Fanxin Kong. Enhancing llm-based test generation by eliminating covered code. *arXiv preprint arXiv:2602.21997*, 2026.

Haoran Xue, Reem Aleithan, Nafid Enan, Gias Uddin, and Song Wang. Swe-bench+: Enhanced llm coding benchmark. In *Proceedings of the 3rd ACM International Conference on AI-Powered Software*, pages 332–339, 2026.

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

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38, 2026.

Z.ai. GLM-5.3 model card, 2026. https://huggingface.co/zai-org/GLM-5.3. Official model card. Accessed October 8, 2026.

Quanjun Zhang, Ye Shang, Chunrong Fang, Siqi Gu, Jianyi Zhou, and Zhenyu Chen. TestBench: Evaluating class-level test case generation capability of large language models. *arXiv preprint arXiv:2409.17561*, 2024. https://arxiv.org/abs/2409.17561.

Quanjun Zhang, Ye Shang, Siqi Gu, Jianyi Zhou, Chunrong Fang, Zhenyu Chen, and Liang Xiao. Multi-agent llm collaboration for unit test generation via human-testing-inspired workflows. *arXiv preprint arXiv:2607.09101*, 2026a.

Quanjun Zhang, Yi Zheng, Ye Shang, Weifeng Sun, Haichuan Hu, Chunrong Fang, Zhenyu Chen, and Liang Xiao. Reproagent: Tool-augmented multi-stage agentic generation of bug reproduction tests from issue reports. *arXiv preprint arXiv:2607.09123*, 2026b.

Junda Zhao, Shurui Zhou, and Eldan Cohen. Do coverage and mutation scores of llm-generated test suites correlate with their effectiveness?(replicability study). *arXiv preprint arXiv:2607.22880*, 2026.

Qixing Zhou, Jiacheng Zhang, Haiyang Wang, Rui Hao, Jiahe Wang, Minghao Han, Yuxue Yang, Shuzhe Wu, Feiyang Pan, Lue Fan, et al. Featurebench: Benchmarking agentic coding for complex feature development. *arXiv preprint arXiv:2602.10975*, 2026.

Appendix

A Task Packaging and Instruction Specification

The Harbor evaluation framework To ensure consistent task execution and reproducible evaluation, all tasks in **SpecEngineeringBench** are evaluated using the Harbor framework. Each task package contains the following components:

- `task.toml`: Stores task metadata and execution configuration, with separate sections for the environment, agent, and verifier. These settings specify how Harbor runs the task, including applicable execution limits; they do not describe the defects the agent should discover.
- `instruction.md`: Defines the agent-facing objective, workspace paths, required actions, and submission format. In Task 1, it requests an executable test suite and testcase metadata; in Task 2, it additionally requests execution evidence and a structured defect report. The task instructions are supplied alongside the instance-specific PRD, which describes the functionality to assess.
- `environment/`: Contains the Dockerfile and the repository and PRD inputs used to initialize the task container. The environment exposes the unrepaired repository at `/workspace/repo` and the requirements at `/workspace/prd.md`, with dependencies configured for test execution. Agent-authored tests and reports are written to `/workspace/submission/`.
- `tests/`: Contains the verifier entry point, `tests/test.sh`, and the scoring implementation for the corresponding task. Task 1 evaluates submitted tests on the defective and repaired snapshots to measure validity, defect detection, and coverage; Task 2 compares reported locations with reference annotations at file, function, and hunk granularities. These verifier resources are distinct from both the repository's existing test suite and the tests authored by the agent.
- `solution/`: Retains `solution/solve.sh`, the solution entry point in the Harbor package layout. It provides a separate location for a packaged solution script, while the evaluated agent writes its own test suite or report to the submission paths specified in `instruction.md`.

The agent works with the unrepaired repository; repaired snapshots, reference patches, and defect annotations are reserved for evaluation. Paper Task 1 corresponds to the `task1/` package, and paper Task 2 corresponds to `task3/`. The `task2/` package supports the existing-test reporting baseline.

Task instruction specification. Each task specifies the agent's objective, workspace, and required submission. The PRD is available at `/workspace/prd.md`, the repository at `/workspace/repo`, and the submission directory at `/workspace/submission/`. The two boxes below reproduce the task prompts, including their submission requirements and JSON examples. Original Markdown syntax and indentation are retained; long lines wrap automatically for presentation. These are task-level instructions, distinct from the agent harness's system prompt; the instance-specific PRD is supplied separately.

Task 1: Test Generation

```
You are an experienced software test engineer applying a Test-Driven Development (TDD) approach.
Your task is to design tests that ensure only correct implementations (following the documentation) pass,
  while
incorrect implementations would fail.

Your goal is to discover software problems by writing and executing test cases. Do not only describe tests
   or
return them in chat: save the executable tests in the task output directory, run them in the provided
  environment,
inspect the results, and revise the tests when necessary.

You are given the following information:
- Dependencies
```

```
- Test entry inferences
- Prd Documentation

Your tasks:
1. Infer the intended behavior of the repo from the prd documentation.
2. Design a set of comprehensive test cases that cover:
   - Basic functionality with valid inputs and expected outputs.
   - Boundary conditions and edge cases.
   - Invalid inputs and error handling.
   - Potential issues with dependency interactions.
3. Write executable test code using Pytest.
4. Ensure tests are designed to differentiate between correct and incorrect implementations:
   - At least one test should expose an incorrect implementation that does not fully follow the documented
   behavior.
   - A correct implementation should pass all tests.
5. Use all three complementary testing approaches to test the software comprehensively:
   - Unit testing for isolated function or method behavior.
   - Integration testing for interactions among repository components or dependencies.
   - Acceptance testing for externally observable end-to-end or user-facing behavior.
6. Save the complete executable test suite and any required test data under `/workspace/submission/`.
7. Use the environment instructions to configure the provided environment and run the complete test suite.
   Ensure the generated tests are executable with Pytest and that a correct implementation passes the
  suite.

## Exploration and completion protocol

Do not stop after writing a small representative sample of tests. Treat test design as an
iterative exploration task and work through the repository behavior systematically:

1. Build a checklist from every capability and public surface in the PRD. For each item,
   consider at least one normal case, one boundary or unusual case, and one invalid,
   failure, or state-transition case when that behavior is applicable. Include dependency
   interactions and externally observable workflows, not only isolated helper functions.
2. Work in multiple batches. For a non-trivial repository, plan several exploration
   batches; start each batch with some tests, then run the complete suite,
   inspect failures, and keep the passing tests. After the first batch, continue
   with the next unchecked PRD behavior instead of declaring success because the current
   tests pass.
3. Use coverage as feedback when it is available in the task environment. Compare the
   executed lines with the production entry functions and ranges described by the task
   information. For uncovered regions, decide whether they represent another behavior
   or a genuinely unreachable/non-behavioral line; add a focused test for the former and
   record the reason for the latter in your final notes.
4. Before stopping, perform a completion audit: every applicable PRD capability has a
   test attempt, all three categories have meaningful tests, the suite has been run from
   the task-local environment, and each test has a stable unique ID in
   `testcase_categories.json`. A passing suite alone is not a completion signal.

There is no fixed universal number of test cases: the required amount is determined by
the behavior checklist and the remaining coverage gaps. Continue until that checklist
and the coverage-driven review are complete, subject to the available execution budget.

## Environment

Use the preconfigured Harbor environment. Read the PRD at `/workspace/prd.md`, then run tests from the
  repository root: `/workspace/repo`


## Requirements for the generated tests
- Use Pytest and keep the complete suite executable in the provided repository environment; if the
  environment has problems, you may install the missing dependencies yourself.
- Apply unit, integration, and acceptance testing as complementary approaches for comprehensive coverage.
- Write `/workspace/submission/testcase_categories.json` with one object per testcase, containing `id` and
   exactly one
  `category` value from `unit_test`, `integration_test`, or `acceptance_test`. The JSON must cover every
  testcase
  in the executable suite and include all three testing approaches.
- Put test modules, fixtures, test data, and `testcase_categories.json` in `/workspace/submission/`.
```

```
- Run the complete suite after writing it by using the preconfigured environment described above.
- Do not modify production code, dependency manifests, the test runner configuration, or files outside the
   output
  directory.

- Write /workspace/submission/test_cases.json with id, category, and an exact pytest_nodeid for every
  testcase. The logical ID need not appear in the Python function name; parametrized cases must use their
  full collected node ID. Verify every node ID with pytest --collect-only.

## Output Format
The output is the complete executable test suite and `/workspace/submission/testcase_categories.json`.
```

Task 2: Defect Discovery and Reporting

```
You are an experienced software test engineer applying a systematic test-execution and bug-localization
  approach.
Your task is to inspect one repository and its neutral PRD, write and run focused temporary test cases
  against the provided repository,
identify source-grounded software problems, and produce a report.

You are given:
- PRD document
- The visible repository source
- The repository's declared dependencies

Your tasks:
1. Read the PRD and inspect the repository source, public interfaces, dependencies, and relevant
  documentation.
2. Design a test suite, execute it, analyze the run results, and try to uncover all hidden defects in the
  repository. It may cover:
   - Basic functionality with valid inputs and expected outputs.
   - Boundary conditions and edge cases.
   - Invalid inputs and error handling.
   - Potential issues with dependency interactions.
3. Write executable test code using Pytest. Base assertions on intended behavior so that correct
  implementations pass and incorrect implementations fail.
4. If there are issues with the runtime environment, you may perform fixes yourself, such as installing
  dependencies via pip install.
5. Save the complete executable test suite and any required test data under `/workspace/submission/`.
6. Use the environment instructions to configure the provided environment and run the complete test suite.
   Analyze observed results against expected behavior, investigate failures, and reproduce suspected bugs.
7. Trace confirmed failures to their causal source files, functions, and lines. Write a complete report
   to `/workspace/submission/report.json`, including test results, reproduction evidence, and root-cause
  analysis.

## Environment

The Harbor image exposes the model inputs at fixed paths. Read the PRD first from `/workspace/prd.md`,
  then inspect
and run the repository from `/workspace/repo`:

    cd /workspace/repo

Use the dependencies configured by `environment/Dockerfile`; do not recreate or replace the task
  environment (you can still fix the environment via pip install), and do not look for the repository or
  PRD beside the task directory. If setup has an infrastructure problem, record it in `
  constraints_and_disclosures` and do not claim it is a product failure without an independent
  reproduction.

## Output Format

The output is the complete executable test suite, execution evidence, and `/workspace/submission/report.
  json`.
When a targeted reproduction script is useful, include `/workspace/submission/repro.sh`.
The report's top-level structure must match this format:
```

```json
{
  "title": "short problem title",
  "status": "confirmed|suspected|blocked",
  "severity": "critical|high|medium|low|unknown",
  "test_results": [
    {
      "scope": "test scope",
      "cwd": "working directory",
      "command": "exact command",
      "exit_code": 0,
      "outcome": "what happened",
      "observed": "relevant output or failure",
      "evidence": "relative raw evidence path"
    }
  ],
  "reproduction": {
    "script": "repro.sh",
    "command": "/workspace/submission/repro.sh",
    "uses_authored_tests": true,
    "steps": ["reproduction step"],
    "expected": "documented or correct behavior",
    "actual": "observed incorrect behavior",
    "evidence": "relative raw evidence path",
    "environment_note": "environment limitation or intervention"
  },
  "locations": [
    {
      "path": "repository-relative/source.py",
      "functions": [
        {"name": "Class.method"}
      ],
      "lines": [42, 43],
      "finding": "source-grounded causal finding"
    }
  ],
  "root_cause": {
    "summary": "causal explanation",
    "mechanism": ["causal step"],
    "required_guard": "required behavioral or code guard, when applicable",
    "evidence": ["relative evidence path"]
  },
  "impact": "user-visible or system impact",
  "constraints_and_disclosures": ["setup, dependency, intervention, or scope disclosure"]
}
```

Use one JSON report object to describe the bugs you identify. Preserve exact commands, exit codes, outcomes, observed failures, and evidence paths. If an authored test run is blocked by the environment, record it as infrastructure evidence and do not claim it is a product failure without an independent reproduction.

Localization Detail Requirements

The `locations` array must report every finding at all three localization granularities:

- File: `path` is the repository-relative source file.
- Function: include a `functions` array under that file, with each function's exact `name`. Use a qualified name such as `Class.method` when applicable; use `<module>` for module-level code.
- Code line: Each file entry must include a `lines` array containing the 1-based source-code line numbers of the exact lines relevant to the bug. Do not use vague ranges when precise line numbers can be identified.

The `path`, `functions`, and `lines` fields must be siblings within each file entry. The `lines` array describes relevant locations throughout that file, including locations outside function bodies. Do not nest `lines` inside individual function entries.

````
Use this shape (and repeat it for every affected file):

```json
"locations": [
  {
    "path": "pkg/module.py",
    "functions": [
      {"name": "<module>"},
      {"name": "Class.method"}
    ],
    "lines": [5, 42, 43]
  }
]
```

Always provide the file, function, and exact line details.

Do not produce a separate failure analysis or prose report in place of submission/report.json.
````

B Foundation Models and Agent Harness Configurations

Models and harnesses. Our evaluation setup includes seven foundation models from five model families, paired with the coding-agent harnesses described in Section 3.1. Each evaluated system combines a model with a harness that exposes repository inspection, file editing, and command execution, supporting an iterative cycle of test design, execution, and failure analysis. We use the following model–harness configurations:

- **Anthropic Claude:** We evaluate Claude-Opus-5 (Anthropic, 2026b) and Claude-Sonnet-5 (Anthropic, 2026c) through the Claude Code (Anthropic, 2026a). Both models use its repository inspection, editing, and shell tools to author executable tests and investigate observed failures, providing two Claude configurations within the same harness family.
- **OpenAI GPT:** We evaluate GPT-6-Astra (OpenAI, 2026c) and GPT-5.6-Sol (OpenAI, 2026b) through the Codex (OpenAI, 2026a). The models interact with the task workspace through Codex's file and command-execution tools, using test outcomes to guide subsequent exploration and report preparation.
- **Moonshot Kimi:** We evaluate Kimi-K3 (Moonshot AI, 2026) through Claude Code (Anthropic, 2026a) as the execution harness. This configuration extends the evaluation to the Kimi family while retaining the Claude Code (Anthropic, 2026a) tool interface used for the Claude configurations.
- **DeepSeek:** We evaluate DeepSeek-V4.1-Flash (DeepSeek AI, 2026) through Claude Code (Anthropic, 2026a), using its repository inspection, editing, and shell tools under the same task-level input and submission contracts.
- **Z.ai GLM:** We evaluate GLM-5.3 (Z.ai, 2026) through Claude Code (Anthropic, 2026a), using the same task-level input and submission contracts as the other model families. This configuration extends the comparison of requirements-driven test generation and defect localization to the GLM family.

The setup uses high reasoning effort where supported by the model interface. Comparisons therefore characterize the evaluated model–harness systems; they do not isolate the effect of the foundation model from that of its harness.

Task execution and submissions. Task 1 and Task 2 are evaluated independently: Task 2 does not receive the tests produced in Task 1. Both tasks use the instance's repository, PRD, and configured Docker environment. Task 1 submits executable tests and their registration metadata for evaluation on the defective and repaired snapshots. Task 2 submits independently authored tests, execution evidence, and a structured defect report.

C Case Study

Instance: `python-visualization_folium/episode-0001` (five reference bugs). **Model:** GPT-6-Astra with Codex. The PRD below is the task input. Task 1 and Task 2 outputs come from independent runs on this same instance. We reproduce selected output excerpts; ... marks omitted content, and test outcomes are supplied by the evaluator.

C.1 Input: Product Requirements Document

PRD

```
# Product Requirements Document

## Purpose

Folium is a Python library for creating interactive Leaflet.js maps. This PRD covers reliable
  JavaScript serialization, heatmap data handling, interactive drawing exports, TopoJSON overlays,
  and configurable Leaflet path styling for Python developers and integrators.

## User Interface And Interaction

Users create `folium.Map` objects and add layers or plugins through the public Python API, then
  render or save the resulting map as HTML. JavaScript options supplied through Folium configuration
  are emitted into the generated map. Users can add `folium.plugins.HeatMap` for weighted geographic
  points, `folium.plugins.Draw` for drawing and exporting GeoJSON, `folium.TopoJson` for TopoJSON
  overlays, and vector layers such as `PolyLine`, `Polygon`, `Circle`, and `CircleMarker` with
  Leaflet path options. The generated HTML exposes the corresponding Leaflet controls, overlays,
  styles, and downloads in a browser.

## Functional Requirements

### JavaScript serialization of dictionary options

- User goal: Configure map and layer options with Python dictionaries, including numeric keys.
- Primary interaction or public entry point: Pass dictionaries containing string or numeric keys to
  Folium options rendered with the `tojavascript` template filter.
- Expected outcome: The generated JavaScript is valid and preserves every dictionary key and value,
  including numeric keys, so Leaflet receives the intended options.
- User-visible errors or constraints: Values must remain JSON/JavaScript-serializable according to
  Folium's supported option types.
- Externally visible state change, when applicable: Rendered HTML contains the complete option
  object used by the browser.

### Weighted HeatMap data

- User goal: Display geographic points with optional intensity weights from Python lists,
  pandas-compatible data, or NumPy arrays.
- Primary interaction or public entry point: Instantiate `folium.plugins.HeatMap(data)` with rows
  shaped as `[latitude, longitude]` or `[latitude, longitude, weight]`.
- Expected outcome: Valid integer or floating-point NumPy weights are accepted and rendered as
  heatmap intensities without changing their geographic locations.
- User-visible errors or constraints: Coordinates must pass Folium location validation; data
  containing NaN values is rejected with a clear value error. The input must provide at least
  latitude and longitude for each point.
- Externally visible state change, when applicable: The generated map contains a Leaflet heat layer
  whose points and intensities reflect the supplied data.

### Draw export controls

- User goal: Draw and edit geometries on a map and download the collected shapes as GeoJSON.
- Primary interaction or public entry point: Add `folium.plugins.Draw(export=True)` to a map,
  optionally configuring `filename`, control `position`, `export_position`, drawing options, editing
```

```
  options, and event handlers.
- Expected outcome: Each enabled Draw instance displays a usable export link with a unique DOM id.
  Activating it prepares a download containing the current drawn feature group as GeoJSON, using the
  configured filename.
- User-visible errors or constraints: `export_position` accepts only `topright`, `topleft`,
  `bottomright`, or `bottomleft`; unsupported values produce a value error. Export is available only
  when enabled.
- Externally visible state change, when applicable: Drawn layers are added to the editable feature
  group, and the uniquely identified export link receives a data download target when activated.
  Multiple Draw controls on one map operate independently.

### TopoJSON geometry overlays

- User goal: Render a selected TopoJSON object on a Folium map, regardless of whether the object is
  represented as a collection or a standalone geometry.
- Primary interaction or public entry point: Instantiate `folium.TopoJson(data, object_path, ...)`
  with a mapping, an embedded file-like source, or another documented string source, and identify
  the desired object path.
- Expected outcome: The selected TopoJSON object is converted to a Leaflet GeoJSON layer and
  displayed, with optional styling, tooltips, layer-control settings, and smoothing.
- User-visible errors or constraints: The supplied object path and TopoJSON data must identify a
  supported TopoJSON object; invalid data or paths are reported during normal data loading or
  rendering.
- Externally visible state change, when applicable: A visible map overlay is created and can
  participate in layer controls, styling, and tooltips.

### Leaflet path option forwarding

- User goal: Use the full set of Leaflet path configuration options when creating Folium vector
  overlays.
- Primary interaction or public entry point: Pass documented snake_case or lowerCamelCase path
  options, including additional Leaflet options, through vector-layer constructors or `path_options`.
- Expected outcome: Recognised options are normalized to Leaflet names and unrecognised but valid
  Leaflet path options are retained and emitted to the browser instead of being discarded.
- User-visible errors or constraints: Option values must be compatible with Leaflet and
  JSON/JavaScript serialization. Explicit fill and color settings follow the documented precedence
  of the public API.
- Externally visible state change, when applicable: The rendered vector layer reflects all supplied
  path behavior, appearance, interaction, and placement options.

## Non-Functional Requirements

- Compatibility: Support the package's documented Python requirement of version 3.9 or newer and
  remain compatible with the Leaflet.js and plugin integrations used by Folium's generated HTML.
- Data fidelity: Preserve numeric values and option keys through Python-to-JavaScript rendering
  without silently dropping supported input.
- Validation and stability: Reject invalid geographic coordinates and NaN heatmap data predictably,
  and report invalid Draw control placement values clearly.
- Interoperability: Accept the documented data forms for heatmaps and TopoJSON, including
  NumPy-compatible point data, mappings, and file-like inputs.
- Isolation: Multiple interactive controls or overlays placed on the same map must retain
  independent browser identifiers and behavior.
- Resource and delivery model: Generated maps must remain self-contained with respect to embedded
  user data while loading the documented Leaflet and plugin resources required for browser
  interaction.
```

C.2 Task 1: Generated Tests

The submission contains 159 tests (85 unit, 60 integration, 14 acceptance). Evaluator results: HP 83.65%, F2P 20.75%, EC 96.30%, and CFG 100.00%. There are 33 F2P tests, 26 tests that fail on the repaired snapshot, and 100 remaining tests that do not expose a reference defect (including two skipped DataFrame cases). The excerpts below retain the submitted function bodies; parameter lists, imports, helpers, and other tests are omitted.

Generated tests that expose reference defects

Numeric dictionary keys (`test_serialization_unit.py`; selected parameter: `key=2`).

```
def test_numeric_dictionary_keys_are_preserved(key):
    assert run_js(tojavascript({key: 'color', 'other': 4})) == {str(key): 'color', 'other': 4}
```

Defective: fail, `AttributeError: 'int' object has no attribute 'split'`. **Repaired:** pass. **F2P:** yes.

Additional Leaflet options (`test_path_unit.py`; selected parameters: `key='pane'`, `value='customPane'`, `normalized='pane'`).

```
def test_additional_leaflet_path_options_survive(key, value, normalized):
    assert path_options(**{key: value})[normalized] == value
```

Defective: fail, `KeyError: 'pane'`. **Repaired:** pass. **F2P:** yes.

...

Generated tests that also fail on the repaired snapshot

Quoted dictionary keys (`test_serialization_unit.py`; selected parameter: `key='a"b'`).

```
def test_string_keys_produce_valid_javascript(key):
    assert run_js(tojavascript({key: 3})) == {key: 3}
```

Defective / repaired: fail / fail, `SyntaxError: Unexpected identifier 'b'`. **F2P:** no.

Draw export placement (`test_draw_unit.py`; selected parameter: `position='topright'`).

```
def test_all_documented_export_positions_are_accepted(position):
    draw = Draw(export=True, export_position=position)
    assert draw.export_position == position
```

Defective / repaired: fail / fail, `TypeError: Draw.__init__() got an unexpected keyword argument 'export_position'`. **F2P:** no.

...

Generated tests that pass on both snapshots

Ordinary JSON values (`test_serialization_unit.py`; selected parameter: `value=0`).

```
def test_json_values_round_trip(value):
    assert run_js(tojavascript(value)) == value
```

Defective / repaired: pass / pass. **F2P:** no.

Boolean options (`test_path_unit.py`; selected parameter: `gradient=False`).

```
def test_gradient_boolean_preserved(gradient):
    assert path_options(gradient=gradient)['gradient'] is gradient
```

Defective / repaired: pass / pass. **F2P:** no.

...

C.3 Task 2: Defect Report

The independently authored Task 2 suite has 64 failed tests, 29 passed tests, and seven pandas-only skips. The submitted report contains 15 agent-reported findings. Evaluator results: five of five reference bugs localized; file/function/hunk LR = 100.00%/100.00%/100.00%, and LP = 83.33%/40.00%/62.50%. The excerpt retains the report's wording, with omitted fields, findings, and explanations marked by ....

Task 2 output: report.json


```
{
  "title": "15 confirmed Folium serialization, heatmap, Draw, TopoJSON, and path-option defects",
  "status": "confirmed",
  "severity": "high",
  "test_results": [
    {
      "scope": "Final complete authored Pytest suite: 100 collected cases across all five PRD areas",
      "cwd": "/workspace/repo",
      "command": "/workspace/submission/repro.sh > /workspace/submission/evidence/run-console.log
  2>&1",
      "exit_code": 1,
      "outcome": "64 failed, 29 passed, 7 skipped; no collection/runtime infrastructure errors in
  the final suite. Failures represent 15 source-grounded findings, not 64 separate bugs.",
      "observed": "Draw: 10 failed/3 passed. HeatMap: 13 failed/14 passed/7 optional pandas skips.
  Serialization: 10 failed/4 passed. TopoJSON: 22 failed/5 passed. Vectors: 9 failed/3 passed. The
  script executes: python -m pytest /workspace/submission -vv --tb=short -rA
  --junitxml=/workspace/submission/evidence/results.xml .",
      "evidence": "evidence/pytest.log"
    },
    ...
  ],
  "reproduction": {
    "script": "repro.sh",
    "command": "/workspace/submission/repro.sh",
    "uses_authored_tests": true,
    ...
    "expected": "All supported serialization, heatmap, drawing, TopoJSON, and vector-option cases
  pass; deliberately invalid inputs produce the required errors. Independent Draw links preserve
  feature-group and filename identity.",
    "actual": "64 assertion/exception failures confirm B01-B15. Normal string options, supported
  JsCode/Element references, ordinary heatmaps and location/NaN validation, single Draw export/event
  configuration, collection TopoJSON styling/tooltips, and documented vector normalization/fill
  precedence pass. Seven DataFrame cases are skipped solely for missing pandas.",
    "evidence": "evidence/pytest.log",
    ...
  },
  "locations": [
    {"path": "folium/template.py",
     "functions": [{"name":"tojavascript"}],
     "lines": [17,18],
     "finding": "B01 [high]: Numeric dictionary keys crash JavaScript option serialization. ..."},
    {"path": "folium/plugins/heat_map.py",
     "functions": [{"name":"HeatMap.__init__"},{"name":"<module>"}],
     "lines": [51,80,81,82],
     "finding": "B03 [high]: NumPy heatmap intensities are not made JSON serializable. ..."},
    {"path": "folium/plugins/draw.py",
     "functions": [{"name":"<module>"}],
     "lines": [61,77,126,127,130,133,134],
     "finding": "B05 [high]: Multiple Draw exports share one DOM id and overwrite each other. ..."},
    {"path": "folium/features.py",
     "functions": [{"name":"TopoJson.style_data"},{"name":"TopoJson.render"}],
     "lines": [1056,1057,1059,1066],
     "finding": "B07 [high]: TopoJson rendering unconditionally assumes a GeometryCollection. ..."},
    {"path": "folium/features.py",
     "functions": [{"name":"GeoJsonDetail.render"}],
```

```
    "lines": [1202,1203,1204,1205,1206,1207],
    "finding": "B13 [medium]: TopoJSON tooltips independently assume a nonempty geometry
  collection. ..."},
   {"path": "folium/vector_layers.py",
    "functions": [{"name":"path_options"}],
    "lines": [81,110,123,125,126],
    "finding": "B14 [medium]: Valid additional Leaflet path options are silently discarded. ..."},
   ...
  ],
  "root_cause": {
    "summary": "The confirmed failures arise from unchecked input-shape/type assumptions, incomplete
  option forwarding, and non-isolated generated identifiers. The 15 findings are separated below by
  their causal source logic; repeated parameterized failures are counted only once per finding.",
   ...
  },
  "impact": "Valid maps fail during Python rendering for numeric option keys, common NumPy heatmap
  weights, and supported TopoJSON forms. Escaping defects can corrupt property names or break
  generated scripts. Multiple Draw exports can download the wrong feature group and filename, while
  one link is unusable. Export placement cannot be configured. Additional vector options silently
  lose interaction, placement, and attribution settings; show=False overlays remain initially
  visible; TopoJSON bounds can fail or encompass unrelated data. Severity is high overall for
  rendering and export/data-fidelity failures; individual severities are recorded in the finding
  entries.",
  "constraints_and_disclosures": [
    "Pandas, declared in requirements-dev.txt, was unavailable. Both installation attempts failed
  with package-index TLS/network errors. The initial collection error is preserved, and seven
  DataFrame-only tests skip explicitly. No pandas failure is claimed; NumPy/list reproductions
  independently confirm B03.",
    "Draw handlers and TopoJSON visibility were evaluated in Node.js against documented DOM/Leaflet
  interfaces implemented by deterministic test doubles. These establish generated-code behavior and
  duplicate-id effects, but are not full-browser visual, real-download, or CDN integration tests.",
   ...
  ]
}
```

D Verbatim Trajectory Case Studies and Failure Mechanics

Complementing the complete walkthrough in Appendix C, the following six cases provide focused qualitative evidence for the mechanisms discussed in RQ1 and RQ5. Cases 1–2 form a paired GPT-5.6-Sol comparison on the same Folium instance: the repository-provided suite leaves an option contract unchecked, whereas requirements-derived assertions expose the shared defect. Case 3 adds a Textual pair in which thousands of passing existing tests coexist with boundary failures exposed by authored tests. Cases 4–6 examine GPT-6-Astra test-generation outcomes. Each case combines task metadata, operational outcome, verbatim commands or test excerpts, and a short failure or success analysis. Excerpts are transcribed from saved trajectories, submitted tests, and execution logs; step numbers refer to the original trajectories.

Case 1: Passing Tests Leave an Option Contract Unchecked (GPT-5.6-Sol on Folium)

Task Metadata: `python-visualization_folium/episode-0004` (Python; archived evaluation).
Agent & Harness: GPT-5.6-Sol via Codex.
Test Suite: Repository-provided tests.
Execution Evidence: PRD-focused subset: 30 passed, zero failed.
Localization Recall (File / Function / Hunk): 20.00% / 0.00% / 0.00%.
Outcome: The report omits the recorded option-forwarding defect.

Verbatim Test Command and Output (Step 25). The agent runs the following command in `/workspace/repo`; the output is saved in `evidence/prd_suite.raw.txt` with exit code zero.

```
python -m pytest -q tests/plugins/test_heat_map.py tests/test_features.py
  tests/test_vector_layers.py

.............................                                        [100%]
30 passed in 1.15s
```

Verbatim Agent Narration (Step 28, excerpt).

> *"Focused reproduction matches both failures exactly, and the PRD-targeted test set passes 30/30."*

Failure Analysis: The existing vector-layer tests check default configurations and conversion of `fill_color` to `fillColor`, but contain no assertion that extra options such as `pane` are retained. Thus, passing tests of the relevant components leave this explicit requirement unchecked. The agent reports failures concerning stale map CSS and non-string icon colors, while the shared option-forwarding defect remains absent from its report.

Case 2: Requirements-Derived Assertions Expose a Shared Defect (GPT-5.6-Sol on Folium)

Task Metadata: `python-visualization_folium/episode-0004` (Python; archived evaluation).
Agent & Harness: GPT-5.6-Sol via Codex.
Test Suite: Agent-authored tests.
Execution Evidence: Five extra-option cases fail; all five fail again on reproduction.
Localization Recall (File / Function / Hunk): 80.00% / 71.43% / 71.43%.
Outcome: The report localizes the shared defect to `path_options`.

Verbatim Submitted Test (written at Step 13, excerpt). The agent hypothesizes at Step 8 that `path_options` drops unconsumed keywords. It then parameterizes the assertion below over `PolyLine`, `Polygon`, `Rectangle`, `Circle`, and `CircleMarker`. The excerpt preserves lines 210–224 of `test_prd_behaviors.py`; the parameterization is omitted.

```
def test_vector_layers_preserve_additional_leaflet_path_options(factory):
    layer = factory()

    assert layer.options["pane"] == "labels"
    assert layer.options["renderer"] == "canvas-renderer"
    assert layer.options["interactive"] is False
    assert layer.options["customOption"] == 7


def _extension_options():
    return {
        "pane": "labels",
        "renderer": "canvas-renderer",
        "interactive": False,
        "custom_option": 7,
    }
```

Verbatim Test Output (Step 16, PolyLine case). The 27-case suite has six passes and 21 failures, with zero JUnit errors. All five extra-option cases fail at the same assertion; `full-run.txt` records:

```
>       assert layer.options["pane"] == "labels"
               ^^^^^^^^^^^^^^^^^^^^
E       KeyError: 'pane'

../submission/test_prd_behaviors.py:212: KeyError
```

Success Analysis: The new assertion directly checks the PRD's extra-option contract. The agent traces the failures to `folium/vector_layers.py`, function `path_options`, including lines 123, 125, and 126. This helper consumes named options and merges `extra_options`, but discards the remaining `kwargs`; all five constructors depend on it. The reference fix for `python-visualization_folium-pr-2272` adds `default.update(kwargs)` before the return, agreeing with this diagnosis. The five failures therefore exercise one shared recorded defect.

Case 3: Thousands of Passing Tests Miss Boundary Failures (GPT-5.6-Sol on Textual)

Task Metadata: `textual/episode-0001` (Python).
Agent & Harness: GPT-5.6-Sol via Codex, paired testing conditions.
Execution Evidence: Existing suite: 3,448 passed; authored suite: 19 passed, three failed.
Localization Recall (File / Function / Hunk): Repository-provided: 0.00% / 0.00% / 0.00%; agent-authored: 18.18% / 9.52% / 2.78%.
Outcome: Authored boundary checks yield failures and nonzero reference agreement.

Verbatim Existing-Suite Command and Output. The initial run has 437 failures under `NO_COLOR=1` and `TERM=dumb`. The agent adjusts only these terminal-color variables and reruns the full suite. The command and final output in `evidence/full_pytest_color_enabled_output.txt` are:

```
env -u NO_COLOR TERM=xterm-256color python -m pytest -q

445 snapshots passed.
3448 passed, 3 skipped, 4 xfailed, 1 xpassed in 292.80s (0:04:52)
```

Verbatim Authored Test (submitted source, lines 189–195). The same PRD requires Rich-compatible rendering of plain and styled text. The new suite includes this boundary check for an unstyled renderable at zero opacity:

```
def test_zero_text_opacity_handles_unstyled_rich_segments() -> None:
    console = Console(width=20, color_system="truecolor")
    renderable = TextOpacity(Text("plain", end=""), opacity=0)

    rendered = list(console.render(renderable))

    assert "".join(segment.text for segment in rendered) == " " * len("plain")
```

Verbatim Failure Output (`pytest-full.txt`, excerpt).

```
>               invisible_style = _from_color(bgcolor=style.bgcolor)
                                                      ^^^^^^^^^^^^

E               AttributeError: 'NoneType' object has no attribute 'bgcolor'

src/textual/renderables/text_opacity.py:80: AttributeError

3 failed, 19 passed in 1.07s
```

Analysis: The zero-opacity branch assumes every Rich segment has a style, although valid unstyled text produces `style=None`. The other two failures concern an out-of-range CSI-u Unicode value and an rxvt control-down sequence decoded as control-up. Focused reproductions repeat the rendering failure and both parser failures; a separate run has 16 passing controls. The agent reports the parser table, parser function, and rendering branch. The pair shows that a large passing suite can leave specific requirements-derived boundary checks unresolved, while the low hunk recall also shows that these failures do not yield comprehensive reference localization.

Case 4: Broad Test Registration Without Defect Exposure (GPT-6-Astra on Apache Superset)

Task Metadata: `apache_superset/episode-0003` (Python).
Agent & Harness: GPT-6-Astra via Codex.
Operational Metrics: HP 100.00% / F2P 0.00% / EC 56.82% / CFG 69.23% / Num 99.
Outcome: The registered suite is compatible with the repaired snapshot but exposes none of the recorded defects.

Verbatim Agent Narration (Step 40).

> *"The suite now runs end to end: 99 tests collect, with 17 passing and 64 expected failures that directly expose supplied implementation regressions; 18 xpasses show behaviors already fixed in this checkout."*

Failure Analysis: The agent invests in broad unit, integration, and acceptance coverage and reports a clean collection, yet the verifier records no F2P test. The trajectory treats expected-failure outcomes and xpasses as evidence of progress, but those outcomes do not establish a test that separates the defective and repaired snapshots. This case isolates the RQ1 gap between a large, executable submission and defect-discriminating assertions.

Case 5: Requirements-Derived Assertions Expose Multiple Defects (GPT-6-Astra on Folium)

Task Metadata: `python-visualization_folium/episode-0004` (Python; archived evaluation).
Agent & Harness: GPT-6-Astra via Codex.
Operational Metrics: HP 95.54% / F2P 33.12% / EC 88.57% / CFG 100.00% / Num 157.
Outcome: Requirements-derived tests expose multiple defective behaviors while retaining high validity and change coverage.

Verbatim Agent Narration (Step 9).

> *"The code suggests several gaps worth testing: NumPy weights may remain non-serializable, TopoJSON styling assumes a geometry collection, and extra vector options appear to be dropped."*

Verbatim Submitted Assertion (Step 10, excerpt).

```
def test_each_vector_constructor_forwards_options(kind):
    layer = vector(kind, interactive=False, pane="customPane", ...)
    assert layer.options["interactive"] is False
```

Success Analysis: The agent converts the PRD into concrete serialization, geometry, and option-forwarding checks. On the defective snapshot, the focused run reports `KeyError: 'interactive'` for the vector-option assertion; the full submission records 99 passes and 58 failures, and the verifier credits 33.12% of registered tests as F2P. The case shows how a requirements-derived oracle can turn broad code coverage into discriminating defect exposure.

Case 6: Assertion Oracle Mismatch Under a Familiar API (GPT-6-Astra on Ax)

Task Metadata: `facebook_Ax/episode-0004` (Python).
Agent & Harness: GPT-6-Astra via Codex.
Operational Metrics: HP 19.26% / F2P 3.70% / EC 61.04% / CFG 20.00% / Num 135.
Outcome: Many tests fail on the repaired snapshot because their expected transformed values do not match the public API behavior.

Verbatim Failing Assertion (`test_standardization.py`).

```
assert oc.objective.weights == pytest.approx(
    [weights[0] * 2, weights[1] * 20]
)
E assert [2.0, -3.0] == approx([4.0, -60.0])
```

Failure Analysis: The test encodes an inferred rescaling rule that conflicts with the documented contract. The same assertion passes on the defective snapshot but fails on the repaired snapshot, so it cannot count as F2P and instead lowers HP. The trajectory's final audit reports 135 tests, 91 passes, and 44 failures; the verifier credits only 3.70% F2P. This case exposes assertion mismatch as a direct construction failure even when the suite exercises the target transformation code.

E Repository Inventory and Instance Counts

Table 8 lists 78 repositories, 100 instances, and 531 recorded defect occurrences. Instances from the same upstream repository are grouped together. Each instance supports both Task 1 and Task 2 and is counted once.

Table 8 Repository inventory by application type. License lists the current upstream primary code license, checked on 15 October 2026 for retained repositories and 20 October 2026 for additions, with links to the upstream terms. Defects are recorded occurrences in the current reference annotations, counted separately in each instance. kLOC is the median non-documentation source-code line count across each repository's released defective snapshots, expressed in thousands of lines (1 kLOC = 1,000 lines) and rounded to one decimal place. Counts exclude comments and blank lines while retaining tests, examples, generated code, and vendored source.

Repository	License	Instances	Defects	kLOC
Communications				
element-hq/synapse	AGPL-3.0	1	13	283.0
letta-ai/letta	Apache-2.0	1	5	161.8
Data Analysis				
apache/superset	Apache-2.0	4	26	1,017.0
bashtage/arch	NCSA	2	10	42.2
comet-ml/opik	Apache-2.0	1	5	1,005.1
Data-Centric-AI-Community/data-profiling	MIT	1	4	16.5
e2nIEE/pandapower	BSD-3-Clause	1	5	84.6
lmfit/lmfit-py	BSD-3-Clause	1	4	10.7
python-visualization/folium	MIT	2	11	6.5
pyvista/pyvista	MIT	1	7	119.8
skforecast/skforecast	BSD-3-Clause	1	4	112.8
Database				
airbytehq/airbyte	ELv2	1	4	525.5
ansible/awx	Apache-2.0	1	5	113.2
coleifer/peewee	MIT	1	5	35.5
dagster-io/dagster	Apache-2.0	1	4	1,107.0
datahub-project/datahub	Apache-2.0	2	13	1,760.3
dgunning/edgartools	MIT	3	14	922.5
django-haystack/django-haystack	BSD-3-Clause	1	5	15.5
facebook/Ax	MIT	3	14	150.7
Farama-Foundation/Gymnasium	MIT	1	6	32.0
mealie-recipes/mealie	AGPL-3.0	1	5	107.3
mlflow/mlflow	Apache-2.0	1	6	1,417.3
pyproj4/pyproj	MIT	1	5	94.2
Desktop Environment				
materialsproject/pymatgen	MIT	1	5	237.1
File Formats				
erocarrera/pefile	MIT	1	4	6.6
jmcnamara/XlsxWriter	BSD-2-Clause	1	5	33.8
jupyter/nbconvert	BSD-3-Clause	1	6	33.1
python-excel/xlrd	BSD-3-Clause / custom	1	5	9.4
Games/Entertainment				
rommapp/romm	AGPL-3.0	1	5	233.9
Tuxemon/Tuxemon	GPL-3.0	1	6	127.0
Home Automation				
home-assistant/supervisor	Apache-2.0	1	8	82.9
Internet				
boto/boto3	Apache-2.0	1	5	8.5
camel-ai/camel	Apache-2.0	4	25	160.2
cherrypy/cherrypy	BSD-3-Clause	1	6	17.3
encode/httpcore	BSD-3-Clause	1	8	8.2
googleapis/google-api-python-client	Apache-2.0	1	5	9.2
pallets/flask	BSD-3-Clause	1	6	10.4
python-restx/flask-restx	BSD-3-Clause	1	5	13.7

Continued on next page

Table 8 continued

Repository	License	Instances	Defects	kLOC
Multimedia				
cuthbertLab/music21	BSD-3-Clause	1	4	122.1
deepchem/deepchem	MIT	1	4	56.0
harry0703/MoneyPrinterTurbo	MIT	3	13	31.8
Kometa-Team/Kometa	MIT	1	5	31.5
morpheus65535/bazarr	GPL-3.0	1	4	993.8
Office/Business				
hummingbot/hummingbot	Apache-2.0	1	5	242.7
meta-pytorch/captum	BSD-3-Clause	1	9	43.6
Scientific/Engineering				
GPflow/GPflow	Apache-2.0	1	5	10.4
kubeflow/pipelines	Apache-2.0	1	5	425.5
NeuroTechX/moabb	BSD-3-Clause	2	9	64.6
online-ml/river	BSD-3-Clause	1	5	37.6
PyDMD/PyDMD	MIT	1	4	2.8
Security				
data-privacy-stack/presidio	MIT	3	15	37.1
demisto/content	MIT	1	7	1,647.4
domainaware/parsedmarc	Apache-2.0	1	5	27.3
oauthlib/oauthlib	BSD-3-Clause	1	7	10.3
ossf/cve-bin-tool	GPL-3.0	1	4	47.5
PurpleAILAB/Decepticon	Apache-2.0	1	4	123.5
Software Development				
arrow-py/arrow	Apache-2.0	1	15	11.9
ContextLab/hypertools	MIT	1	4	2.0
damnever/pigar	BSD-3-Clause	1	4	1.9
dask/distributed	BSD-3-Clause	1	4	96.4
davidhalter/parso	MIT / PSF-2.0	1	5	9.4
evennia/evennia	BSD-3-Clause	1	5	94.0
facebookresearch/hydra	MIT	1	5	35.2
flwrlabs/flower	Apache-2.0	1	4	202.9
inventree/InvenTree	MIT	1	5	186.2
jacebrowning/memegen	MIT	1	5	3.5
jupyter/jupyter_client	BSD-3-Clause	1	6	8.7
langchain-ai/langchain	MIT	2	10	257.5
pydantic/pydantic	MIT	1	5	148.1
UKGovernmentBEIS/inspect_ai	MIT	1	4	526.9
System				
saltstack/salt	Apache-2.0	1	5	839.3
siliconcompiler/siliconcompiler	Apache-2.0	2	10	69.3
Terminals				
aws/aws-cli	Apache-2.0	1	5	81.1
Azure/azure-cli	MIT	1	4	1,966.1
boto/botocore	Apache-2.0	3	14	70.7
Textualize/textual	MIT	1	4	79.8
Text Processing				
nltk/nltk	Apache-2.0	1	6	84.9
pemistahl/lingua-py	Apache-2.0	1	4	5.3