SWEPATCHTEST: EVALUATING SOFTWARE TESTS BEYOND REPRODUCTION

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents use generated tests to decide whether a proposed patch satisfies an issue's requirements. A suite that fails on the buggy program and passes after applying the reference patch can still accept an incorrect patch or reject a correct patch. We introduce SWEPATCHTEST[1], a test-generation benchmark with 619 tasks from SWE-bench Pro, SWE-bench Multilingual, and DeepSWE and 82,579 candidate patches collected from coding-agent trajectories. By replaying each suite on the buggy and patched codebases, SWEPATCHTEST jointly measures issue reproduction, acceptance of correct patches, and rejection of incorrect patches. Joint Validation Rate (JVR), which requires all three criteria on the same task, exposes failures hidden by single-reference evaluation: the best-performing configuration achieves only 9.21% JVR despite reproducing the reference transition on 59.29% of tasks. We further use validation scores to select and scale demonstrations for supervised fine-tuning (SFT). The resulting training recipe approximately triples JVR and improves downstream repair across three SWE-bench benchmarks. Patch discrimination thus provides both a diagnostic measure of test quality and an execution-based signal for curating agent training data.

1 INTRODUCTION

Large language model (LLM) coding agents address software issues by exploring repositories, editing code, and using execution feedback to revise their patches (Yang et al., 2024; Xia et al., 2024; Wang et al., 2025b; Li et al., 2026b). SWE-bench and SWE-bench Pro assess the resulting patches with executable test suites (Jimenez et al., 2024; Deng et al., 2025). During repair, however, agents generate their own tests to distinguish correct patches (passing official evaluation) from incorrect patches (Pan et al., 2026).

Limitations of existing benchmarks. Table 1 contrasts the scale, language coverage, and evaluation criteria of test-generation (TestGen) benchmarks. Pool sizes exclude reference patches and include mutants as negative candidates; checks denote unreported sizes, and crosses denote no pool in the main evaluation. SWT-Bench and TDD-Bench Verified evaluate generated suites through a fail-to-pass (F2P) transition: failure on the buggy program followed by success on the reference patch (Mündler et al., 2024; Ahmed et al., 2024). This establishes reference recognition but leaves two complementary requirements untested: **(i) accepting correct patches** and **(ii) rejecting incorrect patches**. We refer to implementation-dependent rejection of correct patches as *solution bias* and insufficient checking of required behavior as *weak specification*; either can coexist with a successful reference transition.

Test-suite overfitting lets patches satisfy available tests while retaining faults (Smith et al., 2015). Evaluating both candidate classes reveals whether a test captures the issue's requirements beyond one reference patch. Prior work already uses generated tests for patch filtering and evaluates suites against multiple patches or mutations (Mündler et al., 2024; Sonwane et al., 2026; Sun et al., 2026). We ask how reliably these tests accept correct patches and reject incorrect patches drawn from diverse coding-agent trajectories.

[1]Our data are available at https://huggingface.co/datasets/anonymous-swepatchtest/SWEPatchTest.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Comparison of test-generation benchmarks.

Benchmark	F2P	Correct pool	Incorrect pool	Joint Validation	Tasks	Test scale	Multilingual
TestEval	✗	✗	✗	✗	210	Function	✗
TestGenEval	✗	✗	✓	✗	1,210	File	✗
CLOVER	✗	✗	✗	✗	845	Block/file	✗
MultiFileTest	✗	✗	✗	✗	60	Multi-file	3
SWT-Bench	✓	✗	✗	✗	1,983	Repo.	✗
TDD-Bench Verified	✓	✗	✗	✗	449	Repo.	✗
OmniCode	✗	✗	1,109	✗	396	Repo.	3
SWE-Mutation	✓	✗	2,636	✗	800	Repo.	10
SWEPATCHTEST	✓	**27,749**	**54,830**	✓	**619**	Repo.	**10**

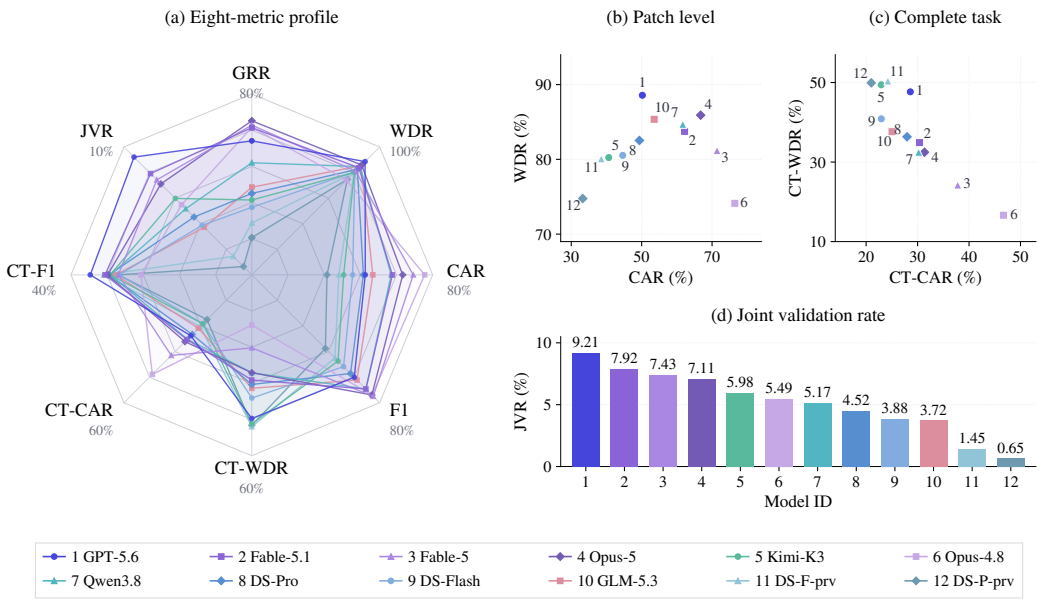

Figure 1: Patch-discrimination profiles of coding-agent configurations on SWEPATCHTEST.

SWEPATCHTEST: from one reference to a patch distribution. SWEPATCHTEST pairs 619 tasks from SWE-bench Pro, SWE-bench Multilingual, and DeepSWE with candidate patches (Deng et al., 2025; Yang et al., 2025b; Huang et al., 2026). We select tasks with suitable reference tests and nontrivial reproduction difficulty, then collect and deduplicate patches from diverse model and agent configurations. The benchmark contains 82,579 candidate patches, split into correct and incorrect pools; Section 2 reports the class counts.

Each generated suite is replayed on the buggy codebase and codebases obtained by applying the reference or each candidate patch to check **(i) issue reproduction**, **(ii) correct-patch acceptance**, and **(iii) incorrect-patch rejection**. Gold Recognition Rate (GRR), Correct Patch Acceptance Rate (CAR), and Wrong Patch Detection Rate (WDR) measure these requirements, respectively. Complete-task rates require success on every candidate in a class; we propose Joint Validation Rate (JVR) to require both complete acceptance and complete rejection together with reference recognition on the same task. Figure 1 summarizes these measures and their harmonic means, separating recognition of the reference patch from discrimination over candidate patches. The radar axes start at zero and scale linearly to the labeled percentage limits; all panels share model IDs and colors, with abbreviations listed in Appendix C.3.

Takeaways. We evaluate 12 coding-agent configurations and investigate whether patch-discrimination scores can guide supervised fine-tuning (SFT), yielding three takeaways. ❶ **Joint validation exposes**

failures hidden by reference recognition. Even the strongest configuration achieves JVR of only 9.21%, compared with its 59.29% GRR, revealing weaknesses that single-reference evaluation leaves unexamined. ❷ **Rejecting incorrect patches does not ensure reliable validation.** The configuration with the highest WDR (88.57%) accepts only 50.20% of correct patches, revealing why test quality must account for both failure detection and acceptance of correct patches. ❸ **Validation-guided test-generation improves both testing and repair.** Selecting and scaling test-generation supervision approximately triples JVR on SWEPATCHTEST and achieves competitive performance across downstream repair benchmarks.

Contributions.

- **(i)** We construct SWEPATCHTEST, a repository-level test-generation benchmark with 619 tasks across ten programming languages and 82,579 candidate patches, enabling evaluation over distributions of diverse correct and incorrect patches.
- **(ii)** We introduce Joint Validation Rate (JVR) and an execution-based protocol that formalize reliable test validation as jointly requiring issue reproduction, correct-patch acceptance, and incorrect-patch rejection on the same task; evaluation of 12 coding-agent configurations exposes limitations obscured by reference recognition alone.
- **(iii)** We develop a validation-guided test-generation pipeline that selects demonstrations using WDR, CAR, and JVR and show that scaling this supervision improves test-generation quality on SWEPATCHTEST and downstream repair on SWE-bench Verified (Jimenez et al., 2024), SWE-bench Multilingual (Yang et al., 2025b), and SWE-bench Pro (Deng et al., 2025).

Figure 2 provides an overview of test generation, benchmark construction, and validation-guided SFT in SWEPATCHTEST.

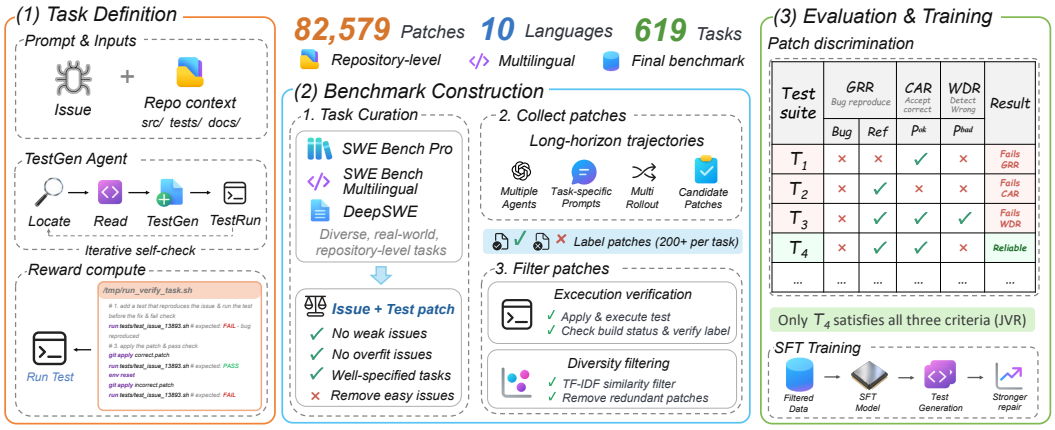

Figure 2: Overview of test generation, benchmark construction, and SFT in SWEPATCHTEST.

2 SWEPATCHTEST: A BENCHMARK FOR PATCH DISCRIMINATION

SWEPATCHTEST evaluates a generated test suite against multiple patches for the same issue, extending reference reproduction to judgments across implementations. We construct the benchmark in three stages: **(1) task curation** establishes reference-test quality and reproduction difficulty; **(2) candidate patch collection** gathers candidate patches; and **(3) candidate verification and selection** forms the correct and incorrect pools. Each task contains an issue, a buggy program, a reference patch, and correct and incorrect patch pools.

2.1 STEP 1: TASK CURATION

We begin with 1,144 repository-level tasks: 731 from SWE-bench Pro, 300 from SWE-bench Multilingual, and 113 from DeepSWE (Deng et al., 2025; Yang et al., 2025b; Huang et al., 2026). Task curation targets two properties: **(i) the reference tests should appropriately enforce the**

behavior required by the issue, and **(ii) reproducing the fail-to-pass transition should remain nontrivial for current test-generation configurations**. Applying these filters leaves 619 tasks for evaluation; Table 4 summarizes the resulting benchmark composition by source.

Reference-test quality. We apply an LLM-based audit that jointly examines the issue description, reference patch, and official tests. The audit excludes tasks with weak or implementation-dependent reference assertions, addressing test-oracle limitations (Barr et al., 2015; Chu et al., 2026). We additionally conduct a manual review of the filtering results before finalizing task eligibility. Appendix A provides the judging criteria and further details.

Reference-transition difficulty. We further filter tasks using a heterogeneous panel of test-generation configurations, removing issues whose fail-to-pass transition is consistently reproduced across the panel. This step removes tasks that are broadly easy under the panel and yields a challenge set with nontrivial reference-transition difficulty. Appendix A summarizes the task-curation procedure.

2.2 Step 2: Candidate Patch Collection

We collect candidate patches from diverse model families and coding-agent harnesses; Appendix B lists the model inventory in Table 5. For each task, we retain patches and their initial execution outcomes from coding-agent trajectories, capturing diverse implementations in both classes for evaluation beyond the reference patch.

2.3 Step 3: Candidate Verification and Selection

We apply each candidate patch and rerun the official task evaluation: passing patches form the correct pool and failing patches the incorrect pool. Within each task and correctness class, we remove duplicate and near-duplicate patches and select diverse subsets to preserve implementation diversity. The reference patch is retained as a separate evaluation target rather than included in the candidate pool. Appendix B details similarity filtering, diversity selection, and final curation.

2.4 SWEPatchTest Statistics

The final benchmark contains **619 tasks across ten programming languages**: 342 from SWE-bench Pro, 167 from SWE-bench Multilingual, and 110 from DeepSWE. It contains **82,579 candidate patches**: 27,749 correct patches and 54,830 incorrect patches, averaging 44.83 and 88.58 per task, respectively (Table 6). These pools support evaluation of both correct-patch acceptance and incorrect-patch rejection.

2.5 SWEPatchTest Evaluation Protocol

Codebases, patches, and test execution. Following the notation of SWT-Bench (Mündler et al., 2024), let R_i denote the buggy codebase for task i and $R_i \circ p$ the codebase obtained by applying patch p. Each task provides a reference patch p_i^{ref} and correct and incorrect patch pools P_i^{ok} and P_i^{bad}. For a test s and codebase R, $\mathrm{exec}(s, R) \in \{\mathrm{P}, \mathrm{F}, \perp\}$ records a completed pass, a stable behavioral failure, or an unresolved outcome. We extend this notation to a submitted nonempty suite T_i: $\mathrm{exec}(T_i, R) = \mathrm{P}$ when all tests pass, F for a completed run with at least one stable behavioral failure, and $\perp$ for missing or unresolved executions, including infrastructure-only errors. We replay the same suite unchanged on clean copies of R_i, $R_i \circ p_i^{ref}$, and each $R_i \circ p$; Appendix C.1 gives the replay protocol.

Measuring acceptance and rejection. We define reference reproduction g_i, candidate acceptance $a_i(p)$, and candidate rejection $r_i(p)$ directly from execution outcomes:

$$\begin{aligned} g_i &= \mathbf{1}[\mathrm{exec}(T_i, R_i) = \mathrm{F} \ \wedge \ \mathrm{exec}(T_i, R_i \circ p_i^{ref}) = \mathrm{P}], \\ a_i(p) &= \mathbf{1}[\mathrm{exec}(T_i, R_i \circ p) = \mathrm{P}], \qquad r_i(p) = \mathbf{1}[\mathrm{exec}(T_i, R_i \circ p) = \mathrm{F}]. \end{aligned} \tag{1}$$

An unresolved outcome earns neither acceptance nor rejection credit. Unresolved executions remain in the score denominators, so failures to produce a usable verdict cannot improve measured discrimination. We replay inconsistent outcomes and require stable behavioral evidence before crediting

rejection (Appendix C.1). The fractions of incorrect patches rejected and correct patches accepted are

$$w_i = \frac{\sum_{p \in P_i^{bad}} r_i(p)}{|P_i^{bad}|}, \qquad c_i = \frac{\sum_{p \in P_i^{ok}} a_i(p)}{|P_i^{ok}|}. \tag{2}$$

Gold Recognition Rate (GRR), Wrong Patch Detection Rate (WDR), and Correct Patch Acceptance Rate (CAR) average these outcomes over the same $N = 619$ tasks:

$$\text{GRR} = \frac{1}{N}\sum_{i=1}^{N} g_i, \qquad \text{WDR} = \frac{1}{N}\sum_{i=1}^{N} w_i, \qquad \text{CAR} = \frac{1}{N}\sum_{i=1}^{N} c_i. \tag{3}$$

All tasks receive equal weight, and the candidate pools are fixed across evaluated configurations. Appendix C.2 details the scoring conventions.

Complete-task and balanced scores. Complete-task (CT) rates count tasks with a correct verdict for every candidate in the corresponding class: CT-WDR requires rejecting all incorrect patches, and CT-CAR requires accepting all correct patches. F1 is the harmonic mean of WDR and CAR, and CT-F1 is the harmonic mean of CT-WDR and CT-CAR; Appendix C.2 gives their formulas.

Requiring joint validation. We propose **Joint Validation Rate (JVR)** to measure whether a suite jointly satisfies all three validation requirements on the same task: reproducing the issue, rejecting every incorrect patch, and accepting every correct patch.

$$\text{JVR} = \frac{1}{N}\sum_{i=1}^{N} \mathbf{1}[g_i = 1 \ \wedge \ w_i = 1 \ \wedge \ c_i = 1]. \tag{4}$$

We rank configurations by JVR because it requires all three capabilities on the same task; the other rates diagnose its components. All comparisons use fixed candidate pools, since larger or more diverse pools make complete validation harder.

3 How Reliably Do Generated Tests Validate Patches?

To evaluate generated tests as patch validators and examine the training value of their execution outcomes, we investigate the following research questions (RQs):

- **RQ1: Patch-Discrimination Performance and Failure Analysis.** *How reliably do generated tests reproduce issues, accept correct patches, and reject incorrect patches on the same task, and what failure modes emerge?*
- **RQ2: Selecting Test-generation Demonstrations.** *Does selecting and scaling SFT demonstrations using patch-discrimination scores improve test generation beyond reference-based selection?*
- **RQ3: Transfer to Downstream Repair.** *Does validation-guided test-generation supervision improve repair success on SWE-bench Verified, Multilingual, and Pro?*

Shared evaluation setup. All SWEPatchTest evaluations use the same 619 tasks, reference patches, candidate pools, and labels (Section 2.5). The 29 recorded runs span prompt variants, epochs, scaffolds, and replays: Figure 1 compares 12 main configurations, Table 2 reports eight representative configurations, Table 3 reports four SFT configurations, and other runs support execution audits and prompt comparisons. Both tables report percentages; the main table bolds column maxima among the displayed configurations. We evaluate models with their native coding agents, using OpenHands for the DeepSeek configurations; Appendix C.3 details the agent setup. The comparison characterizes complete agent configurations, including their scaffolds and reasoning settings.

3.1 RQ1: Patch-Discrimination Performance and Failure Analysis

Reliable patch discrimination remains far from solved. Strong performance on the reference patch leaves a substantial gap in validation across candidate patches (Table 2). GPT-5.6-sol achieves the highest JVR, 9.21%, despite recognizing the reference transition on 59.29% of tasks. Claude Opus 5 has the highest GRR at 68.17%, but its JVR is 7.11%, below GPT-5.6-sol. The change in ranking

Table 2: Patch-discrimination performance of representative coding agents on SWEPATCHTEST.

Model	Reference	Patch-level			Complete-task			
	GRR	WDR	CAR	F1	CT-WDR	CT-CAR	CT-F1	JVR
GPT-5.6-sol	59.29	**88.57**	50.20	64.08	47.66	28.59	**35.74**	**9.21**
Claude-Fable-5.1	65.11	83.74	62.19	71.37	34.89	30.37	32.47	7.92
Claude-Opus-5	**68.17**	85.91	**66.76**	**75.13**	32.47	**31.34**	31.89	7.11
Kimi-K3	33.12	80.24	40.64	53.95	**49.43**	22.94	31.34	5.98
Qwen3.8-max	49.60	84.69	61.68	71.38	32.47	30.21	31.30	5.17
DeepSeek-V4-Pro	36.03	82.53	49.36	61.77	36.35	27.95	31.60	4.52
DeepSeek-V4-Flash	29.89	80.53	44.62	57.42	40.87	22.94	29.39	3.88
GLM-5.3	38.77	85.36	53.55	65.81	37.64	25.04	30.07	3.72

shows that strong reference recognition does not necessarily translate into reliable discrimination among candidate patches.

Uncertainty in joint-validation performance. A paired, source-stratified bootstrap (Efron, 1979) with 10,000 replicates yields a 95% confidence interval of $[6.95, 11.47]\%$ for GPT-5.6-sol's JVR (Appendix D.1). Each resample preserves the three source sizes and uses identical task indices across configurations, retaining the source composition and task pairing for comparisons. Its observed lead over Claude-Fable-5.1 and Claude-Opus-5 is not statistically significant under exact two-sided McNemar tests with Holm correction ($p = .350$ and .222, respectively). Thus, the evidence establishes low joint-validation performance more clearly than a definitive ordering among the leading configurations.

Detecting failures and preserving correct patches are distinct requirements. Candidate-level scores reveal why reference success is insufficient: a suite can miss incorrect patches, reject correct patches, or do both on different tasks. GPT-5.6-sol has the highest WDR at 88.57%, yet accepts only 50.20% of correct patches on average. Among the 12 configurations in Figure 1, Claude Opus 4.8 is more balanced, with 74.12% WDR and 76.47% CAR, although its JVR remains 5.49%. This difference matters when tests filter candidate patches: rejecting an incorrect patch is useful only if the suite also accepts correct patches for the issue.

Average success hides incomplete task coverage. For GPT-5.6-sol, requiring a correct verdict for every candidate reduces the scores to 47.66% CT-WDR and 28.59% CT-CAR. For example, rejecting nine of ten incorrect patches gives a task substantial WDR credit but fails its complete-task rejection condition, because one sampled failure can still pass the suite. Thus, a high candidate average can coexist with errors on many tasks, each of which would prevent the suite from acting as a complete validator. Figure 1 makes this distinction explicit through the upper-right CAR/WDR and CT-CAR/CT-WDR scatter plots and the lower-right JVR bars. Even high complete-task acceptance or rejection rates need not overlap on the same issues; JVR additionally requires their intersection with reference recognition. Among the configurations in Table 2, Kimi-K3 attains the highest CT-WDR at 49.43% but only 5.98% JVR, illustrating why optimizing rejection in isolation does not establish reliable validation.

Source differences expose the same imbalance. For GPT-5.6-sol, WDR increases from 82.57% on Multilingual to 96.82% on DeepSWE, while CAR falls from 79.19% to 34.26%. Its JVR drops from 16.17% to 2.73%; the best Pro JVR is 8.77% with Claude-Fable-5 (Table 9). Thus, high rejection performance alone does not establish reliable joint validation, especially when many correct patches are rejected. The source splits differ in issue composition and candidate coverage, so these results characterize the observed difficulty without isolating its cause.

These scores reveal disagreements with the official evaluation; understanding them requires examining which behaviors a test actually checks.

Case study: the limits of reference reproduction. Figure 3 examines a GPT-5.6-sol test generated with Codex for Caddy instance 4774 from SWE-bench Multilingual. The issue arises when a server uses `default_bind` to specify multiple IPv4/IPv6 addresses: configuration parsing fails on the second address, preventing startup. The requested fix must support this global setting while allowing individual sites to override it.

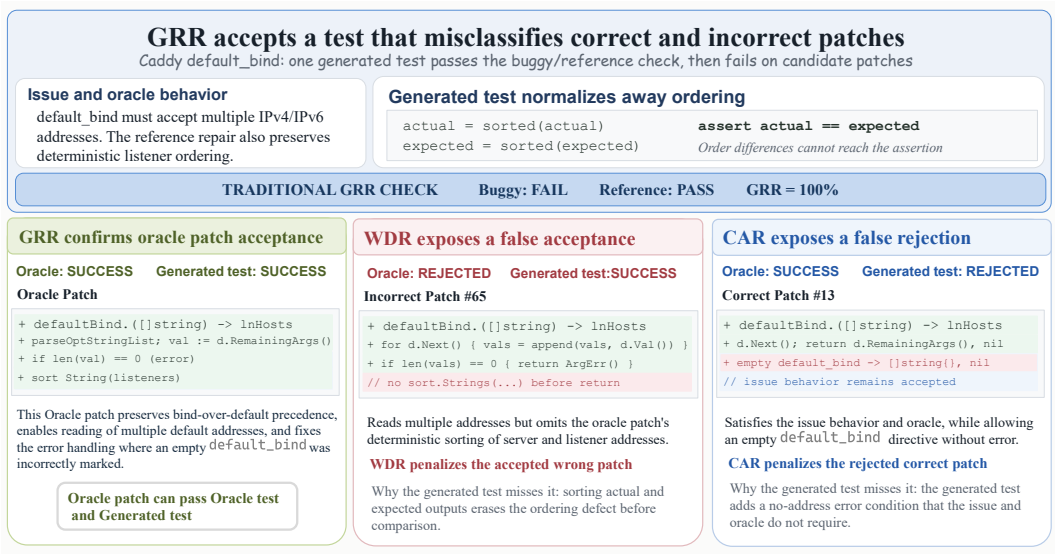

Figure 3: Patch-discrimination case study for Caddy's `default_bind` option.

The test fails on the buggy codebase and passes with the reference patch, yet accepts incorrect patch #65 and rejects correct patch #13 under the official labels. Its comparisons ignore address order, which the official tests check. Conversely, it requires an error for an empty `default_bind`, matching reference behavior that patch #13 does not preserve but the official tests leave unchecked. The issue specifies neither behavior. This suite thus overlooks one distinction enforced by the official tests while checking another beyond their coverage. Both choices are compatible with reference success, which therefore establishes neither the rejection of incorrect patches nor the acceptance of correct ones. Evaluating both candidate classes exposes this gap; interpreting individual disagreements still requires checking the issue requirements and official test coverage. Across this task's pools, the test rejects 66.2% of incorrect patches and accepts 49.1% of correct patches, failing joint validation. Appendix D.3 gives further audit details and examples.

Answer to RQ1: Reference success rarely extends to complete patch discrimination; evaluating both candidate classes exposes differences in coverage that the reference transition leaves untested.

3.2 RQ2: Selecting Test-Generation Demonstrations

All training configurations use Qwen3-30B-A3B-Instruct from the Qwen3 family (Yang et al., 2025a) for five epochs, with trajectories generated by DeepSeek-V4-Flash. The shared SWE SFT baseline is trained on 1.9 billion tokens of repository-level repair trajectories. We augment this baseline with TestGen trajectories while keeping all repositories used for SFT disjoint from those in SWEPatchTest and the downstream evaluation benchmarks. Appendix E describes the data composition and evaluation protocol. We compare three TestGen additions: **(i) TestGen Supervision**, selected by GRR; **(ii) Validation-guided TestGen**, selected by WDR/CAR/JVR; and **(iii) Scaled TestGen Supervision**, which expands the validation-selected data. We select and train on full TestGen trajectories rather than final test scripts alone. Table 3 abbreviates AVG@4 and Pass@4 as AVG and Pass, respectively, and annotates percentage-point changes from SWE SFT above each score. AVG@4 measures the mean per-round resolve rate, while Pass@4 measures the fraction of tasks resolved in at least one of four rounds.

Training-data budgets. TestGen Supervision and Validation-guided TestGen each add 2 billion tokens of TestGen trajectories to the same SWE SFT baseline, selected from the same trajectory pool using GRR and WDR/CAR/JVR, respectively. This pool is constructed from approximately 20,000 issue environments. Scaled TestGen Supervision retains the 2 billion tokens of validation-selected TestGen trajectories and adds another 2 billion tokens selected with the same criteria from an expanded pool, yielding 4 billion tokens of TestGen data in total.

Validation-guided TestGen improves quality beyond reproduction. We evaluate all four SFT configurations directly on SWEPATCHTEST using OpenHands and report their downstream repair scores obtained with Scale SWE in the same table. Compared with GRR-selected TestGen Supervision, Validation-guided TestGen increases JVR from 2.26% to 3.23% and F1 from 50.39% to 56.45%. GRR increases from 39.26% to 40.39%, WDR from 59.00% to 64.37%, and CAR from 43.98% to 50.27%, showing gains across all three validation dimensions: reproducing the issue, rejecting incorrect patches, and accepting correct patches. Scaled TestGen Supervision further reaches 3.88% JVR and 60.18% F1, compared with 1.29% and 22.79% for SWE SFT. These results suggest that patch-discrimination signals provide a useful basis for selecting TestGen demonstrations beyond GRR-based selection alone.

Table 3: Test-generation and downstream repair performance across SFT configurations.

Configuration	SWEPATCHTEST								Verified		Multilingual		Pro	
	GRR	WDR	CAR	F1	CT-WDR	CT-CAR	CT-F1	JVR	AVG	Pass	AVG	Pass	AVG	Pass
SWE SFT	8.89	20.75	25.28	22.79	9.21	19.39	12.49	1.29	67.53	79.20	55.08	72.00	46.56	60.05
+ TestGen Supervision	+30.37 39.26	+38.25 59.00	+18.70 43.98	+27.60 50.39	+4.52 13.73	+8.72 28.11	+5.96 18.45	+0.97 2.26	+1.94 69.47	+0.80 80.00	+6.97 62.05	+4.25 76.25	+2.41 48.97	+0.42 60.47
+ Validation-guided TestGen	+31.50 40.39	+43.62 64.37	+24.99 50.27	+33.66 56.45	+16.48 25.69	+12.11 31.50	+15.81 28.30	+1.94 3.23	+2.71 70.24	+1.40 80.60	+7.94 63.02	+4.33 76.33	+2.62 49.18	+0.63 60.68
+ Scaled TestGen Supervision	+32.31 **41.20**	+51.39 **72.14**	+26.34 **51.62**	+37.39 **60.18**	+22.13 **31.34**	+13.57 **32.96**	+19.64 **32.13**	+2.59 **3.88**	+4.31 **71.84**	+2.80 **82.00**	+8.22 **63.30**	+6.00 **78.00**	+4.14 **50.70**	+1.65 **61.70**

Selection improves complete-task coverage. At the same TestGen budget, validation-guided selection raises CT-WDR from 13.73% to 25.69% and CT-CAR from 28.11% to 31.50% relative to GRR-based selection. More tasks therefore receive correct verdicts for every sampled incorrect patch, while acceptance of the complete correct pool also improves. Scaling raises these rates further to 31.34% and 32.96%, respectively. These gains extend beyond reference recognition, although satisfying both conditions together with reproduction remains difficult.

Interpreting the training comparison. Paired tests quantify the gains over SWE SFT (Appendix E.2). The 95% bootstrap intervals for JVR improvements are $[0.81, 3.23]$ percentage points for Validation-guided TestGen and $[1.29, 4.04]$ for Scaled TestGen Supervision. Both improvements are significant under exact two-sided McNemar tests with Holm correction over the three baseline comparisons ($p = .008$ and $.001$, respectively), whereas the GRR-selected comparison is not ($p = .146$). These comparisons establish gains relative to SWE SFT; they do not directly test the difference between the two selection recipes.

Answer to RQ2: At the same data budget, validation-guided selection outperforms GRR-based selection, with further gains from scaling the selected trajectories.

3.3 RQ3: TRANSFER TO DOWNSTREAM REPAIR

TestGen supervision improves repair success. To test whether the training benefits extend beyond standalone test generation, we evaluate repository repair on SWE-bench Verified, Multilingual, and Pro using AVG@4 and Pass@4. Adding GRR-selected TestGen Supervision raises AVG@4 over SWE SFT by 1.94, 6.97, and 2.41 percentage points on Verified, Multilingual, and Pro, respectively. Validation-guided TestGen further improves AVG@4 by 0.77 percentage points on Verified, 0.97 on Multilingual, and 0.21 on Pro. Pass@4 also increases on all three benchmarks.

Scaled TestGen Supervision further reaches AVG@4/Pass@4 of 71.84%/82.00% on Verified, 63.30%/78.00% on Multilingual, and 50.70%/61.70% on Pro. Its AVG@4 gains over SWE SFT

are 4.31, 8.22, and 4.14 percentage points, respectively. Together with the improvements in patch-discrimination metrics in RQ2, these results show that scaling validation-guided TestGen supervision benefits both standalone test generation and downstream repair.

Scaled supervision also shortens interactions. Mean interaction length decreases from 106.68 to 80.70 steps on Verified, 157.62 to 115.70 on Multilingual, and 151.86 to 119.22 on Pro. These reductions accompany higher repair success, although interaction length does not directly measure elapsed time or token cost.

Answer to RQ3: Scaled TestGen Supervision improves repair success and task coverage across all three downstream benchmarks with fewer interaction steps.

4 Related Work

Generating tests for existing code and reported issues. TestEval, TestGenEval, CLOVER, and MultiFileTest assess test generation across code granularities (Wang et al., 2025a; Jain et al., 2025; Xu et al., 2025a; Wang et al., 2026). SWE-Compass evaluates patch-conditioned test generation for correctness and coverage (Xu et al., 2025b). TestGen-LLM filters test extensions for build success, reliable execution, and coverage gains (Alshahwan et al., 2024). CoverUp and TestForge refine tests using coverage and execution feedback (Altmayer Pizzorno & Berger, 2025; Jain & Le Goues, 2025). Coverage alone need not imply effective fault detection (Inozemtseva & Holmes, 2014). LIBRO, AEGIS, and Otter use issues to guide reproduction or validate changes (Kang et al., 2023; Wang et al., 2024; Ahmed et al., 2025).

Using generated tests to assess candidates. CodeT selects solutions to programming tasks using generated tests and execution agreement (Chen et al., 2022). DiffTGen uses generated tests and an oracle to detect test-suite-overfitted patches (Xin & Reiss, 2017). SWT-Bench and TDD-Bench Verified evaluate repository tests through buggy-to-reference fail-to-pass transitions (Mündler et al., 2024; Ahmed et al., 2024). BRT Agent ranks candidate patches by their pass rates over generated tests (Cheng et al., 2025). Our protocol fixes correct and incorrect patch pools across configurations, measuring acceptance, rejection, and reproduction jointly.

Assessing tests through program changes. Mutation testing measures whether tests detect injected faults (DeMillo et al., 1978; PAPADAKIS et al., 2019). Mutant detection correlates with real-fault detection, although available mutation operators leave some faults unrepresented (Just et al., 2014). OmniCode uses incorrect patches from agent attempts and LLM perturbations (Sonwane et al., 2026), while SWE-Mutation generates agentic mutants for 800 instances spanning ten languages (Sun et al., 2026). Our pools include correct patches, so evaluation penalizes their rejection as well as missed incorrect patches. Execution-based selection, trajectory supervision, and scaled repository data support training code models and tool-use agents (Wei et al., 2024; Pan et al., 2025; Yang et al., 2025b; Zhan et al., 2025; Tian et al., 2026). Our acceptance and rejection signals guide demonstration selection for test generation and downstream repair.

5 Conclusion

We introduced SWEPatchTest to evaluate whether generated suites reproduce issues, accept correct patches, and reject incorrect patches across 619 tasks and 82,579 candidate patches. The results show that reference recognition and high scores on individual candidate classes do not ensure complete validation. Validation-guided selection and scaled TestGen supervision improve both test generation and downstream repair, making patch discrimination useful for benchmarking and training-data selection.

Scope of the findings. Candidate labels follow the official oracle, and JVR evaluates joint validation over the released candidate pools rather than every possible patch. The reproduction-filtered challenge set targets difficult issues and does not represent all repository tasks. Bootstrap intervals quantify task-sampling uncertainty conditional on the recorded runs and candidate pools. Appendix G details these boundaries. Auditing these boundaries can guide training toward tests that generalize across implementations.

AI USE STATEMENT

LLM-based coding agents generated candidate patches and test-generation trajectories, and an LLM judge assessed reference-test quality for task selection, as described in Section 2 and Appendix E. Codex assisted with refining the evaluation framework and metrics, interpreting quantitative results and qualitative patch examples, locating literature, revising manuscript structure and language, and preparing LaTeX tables and plotting code. AI-assisted numerical summaries and tables were checked against the recorded results, and the manuscript was compiled and checked for numerical and reference consistency. The authors take responsibility for the final text, claims, data, and artifacts, including content produced with AI assistance.

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?, 2024. URL https://arxiv.org/abs/2412.02883.

Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate SWE patches. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 752–771. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/ahmed25b.html.

Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE '24, pp. 185–196. ACM, July 2024. doi: 10.1145/3663529.3663839. URL http://dx.doi.org/10.1145/3663529.3663839.

Juan Altmayer Pizzorno and Emery D. Berger. Coverup: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919, June 2025. ISSN 2994-970X. doi: 10.1145/3729398. URL http://dx.doi.org/10.1145/3729398.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015. ISSN 1939-3520. doi: 10.1109/tse.2014.2372785. URL http://dx.doi.org/10.1109/tse.2014.2372785.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022. URL https://arxiv.org/abs/2207.10397.

Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. Agentic bug reproduction for effective automated program repair at google, 2025. URL https://arxiv.org/abs/2502.01821.

Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O'Hearn, Earl T. Barr, Mark Harman, Federica Sarro, and He Ye. Terminalworld: Benchmarking agents on real-world terminal tasks, 2026. URL https://arxiv.org/abs/2605.22535.

R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978. ISSN 0018-9162. doi: 10.1109/c-m.1978.218136. URL http://dx.doi.org/10.1109/C-M.1978.218136.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

B. Efron. Bootstrap methods: Another look at the jackknife. *The Annals of Statistics*, 7(1), January 1979. ISSN 0090-5364. doi: 10.1214/aos/1176344552. URL http://dx.doi.org/10.1214/aos/1176344552.

Morten W. Fagerland, Stian Lydersen, and Petter Laake. The mcnemar test for binary matched-pairs data: mid-p and asymptotic are better than exact conditional. *BMC Medical Research Methodology*, 13(1):91, Jul 2013. ISSN 1471-2288. doi: 10.1186/1471-2288-13-91. URL https://doi.org/10.1186/1471-2288-13-91.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks, 2026. URL https://arxiv.org/abs/2607.07946.

Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pp. 435–445. ACM, May 2014. doi: 10.1145/2568225.2568271. URL http://dx.doi.org/10.1145/2568225.2568271.

Kush Jain and Claire Le Goues. Testforge: Feedback-driven, agentic test suite generation, 2025. URL https://arxiv.org/abs/2503.14713.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark, 2025. URL https://arxiv.org/abs/2410.00752.

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

René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT/FSE'14, pp. 654–665. ACM, November 2014. doi: 10.1145/2635868.2635929. URL http://dx.doi.org/10.1145/2635868.2635929.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2312–2323. IEEE, May 2023. doi: 10.1109/icse48619.2023.00194. URL http://dx.doi.org/10.1109/ICSE48619.2023.00194.

Han Li, Yifan Yao, Letian Zhu, Rili Feng, Hongyi Ye, Jiaming Wang, Yancheng He, Pengyu Zou, Lehan Zhang, Xinping Lei, Haoyang Huang, Ken Deng, Ming Sun, Zhaoxiang Zhang, He Ye, and Jiaheng Liu. Codetracer: Towards traceable agent states, 2026a. URL https://arxiv.org/abs/2604.11641v2.

Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. Contextbench: A benchmark for context retrieval in coding agents, 2026b. URL https://arxiv.org/abs/2602.05892.

Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT/FSE'14, pp. 643–653. ACM, November 2014. doi: 10.1145/2635868.2635920. URL http://dx.doi.org/10.1145/2635868.2635920.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 81857–81887. Curran Associates, Inc., 2024. doi: 10.52202/079017-2601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/94f093b41fc2666376fb1f667fe282f3-Paper-Conference.pdf.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-gym. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Yue Pan, Zimin Chen, Siyu Lu, Zhaoyang Chu, Xiang Li, Han Li, Yang Feng, Claire Le Goues, Federica Sarro, Martin Monperrus, and He Ye. Prometheus: Towards long-horizon codebase navigation for repository-level problem solving, 2026. URL https://arxiv.org/abs/2507.19942.

Mike PAPADAKIS, Marinos KINTIS, Jie ZHANG, Yue Jia, Yves LE TRAON, and Mark Harman. Mutation testing advances: An analysis and survey. In *Advances in Computers*. Elsevier, 2019. doi: 10.1016/bs.adcom.2018.03.015.

Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE/15, pp. 532–543. ACM, August 2015. doi: 10.1145/2786805.2786825. URL http://dx.doi.org/10.1145/2786805.2786825.

Atharv Sonwane, Eng-Shen Tu, Wei-Chung Lu, Claas Beger, Carter Larsen, Debjit Dhar, Simon Alford, Rachel Chen, Ronit Pattanayak, Tuan Anh Dang, Guohao Chen, Gloria Geng, Kevin Ellis, and Saikat Dutta. OmniCode: A benchmark for evaluating software development agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 40634–40661, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.2020. URL https://aclanthology.org/2026.findings-acl.2020/.

KAREN SPARCK JONES. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, January 1972. ISSN 0022-0418. doi: 10.1108/eb026526. URL http://dx.doi.org/10.1108/eb026526.

Yuxuan Sun, Yuze Zhao, Yufeng Wang, Yao Du, Zhiyuan Ma, Jinbo Wang, Mengdi Zhang, Kai Zhang, and Zhenya Huang. Swe-mutation: Can llms generate reliable test suites in software engineering?, 2026. URL https://arxiv.org/abs/2605.22175.

Zichen Tian, Jinpeng Chen, Cheng Gong, Suiyun Zhang, and Rui Liu. Sap: State-guided data synthesis with argument provenance for multi-turn tool use, 2026. URL https://arxiv.org/abs/2609.06124.

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 Luis Chiruzzo, Alan Ritter, and Lu Wang (eds.), *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 3547–3562, Albuquerque, New Mexico, April 2025a. Association for Computational Linguistics. ISBN 979-8-89176-195-7. doi: 10.18653/v1/2025.findings-naacl.197. URL https://aclanthology.org/2025.findings-naacl.197/.

Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. Aegis: An agent-based framework for general bug reproduction from issue descriptions, 2024. URL https://arxiv.org/abs/2411.18015.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents, 2025b. URL https://arxiv.org/abs/2407.16741.

Yibo Wang, Congying Xia, Wenting Zhao, Jiangshu Du, Chunyu Miao, Zhongfen Deng, Philip S. Yu, and Chen Xing. MultiFileTest: A multi-file-level LLM unit test generation benchmark and

impact of error fixing mechanisms. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 28151–28172, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. doi: 10.18653/v1/2026.findings-acl.1403. URL https://aclanthology.org/2026.findings-acl.1403/.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. Selfcodealign: Self-alignment for code generation. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 62787–62874. Curran Associates, Inc., 2024. doi: 10.52202/079017-2008. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/72da102da91a8042a0b2aa968429a9f9-Paper-Conference.pdf.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '17, pp. 226–236. ACM, July 2017. doi: 10.1145/3092703.3092718. URL http://dx.doi.org/10.1145/3092703.3092718.

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, 2025a. URL https://arxiv.org/abs/2502.08806.

Jingxuan Xu, Ken Deng, Weihao Li, Songwei Yu, Huaixi Tang, Haoyang Huang, Zhiyi Lai, Zizheng Zhan, Yanan Wu, Chenchen Zhang, Kepeng Lei, Yifan Yao, Xinping Lei, Wenqiang Zhu, Zongxian Feng, Han Li, Junqi Xiong, Dailin Li, Zuchen Gao, Kun Wu, Wen Xiang, Ziqi Zhan, Yuanxing Zhang, Wuxuan Gong, Ziyuan Gao, Guanxiang Wang, Yirong Xue, Mengtong Li, Mengfei Xie, Xiaojiang Zhang, Jinghui Wang, Wenhao Zhuang, Zheng Lin, Huiming Wang, Zhaoxiang Zhang, Yuqun Zhang, Haotian Zhang, Bin Chen, and Jiaheng Liu. Swe-compass: Towards unified evaluation of agentic coding abilities for large language models, 2025b. URL https://arxiv.org/abs/2511.05459.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

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. In D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (eds.), *Advances in Neural Information Processing Systems*, volume 38, Main Conference. Curran Associates, Inc., 2025b. doi: 10.52202/085713-3239. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/8b86cf5ace600c48fd188efbb8dedec8-Paper-Datasets_and_Benchmarks_Track.pdf.

Zizheng Zhan, Ken Deng, Jinghui Wang, Xiaojiang Zhang, Huaixi Tang, Minglei Zhang, Zhiyi Lai, Haoyang Huang, Wen Xiang, Kun Wu, Wenhao Zhuang, Shaojie Wang, Shangpeng Yan, Kepeng Lei, Zongxian Feng, Huiming Wang, Zheng Lin, Mengtong Li, Mengfei Xie, Yinghan Cui, Xuxing Chen, Chao Wang, Weihao Li, Wenqiang Zhu, Jiarong Zhang, Jingxuan Xu, Songwei Yu, Yifan Yao, Xinping Lei, C. Zhang, Han Li, Junqi Xiong, Zuchen Gao, Dailin Li, Haimo Li, Jiaheng Liu, Yuqun Zhang, Junyi Peng, Haotian Zhang, and Bin Chen. Kat-coder technical report, 2025. URL https://arxiv.org/abs/2510.18779.

Xiangxin Zhao, Han Li, Shuaiting Li, Tianyi Zhao, Earl T. Barr, Federica Sarro, and He Ye. Failure as a process: An anatomy of cli coding agent trajectories, 2026. URL https://arxiv.org/abs/2607.09510.

A Task Representation and Curation

A.1 Task Record and Evidence

We represent each task by its issue, program state, and evaluation evidence:

$$\mathcal{D}_i = (I_i, R_i, p_i^{ref}, X_i, H_i, P_i^{ok}, P_i^{bad}). \tag{5}$$

Here, I_i is the issue description, R_i the buggy codebase, p_i^{ref} the reference patch, X_i the official test patch, and H_i the original execution harness. Applying a patch produces the codebase $R_i \circ p$, following the notation in Section 2.5. The sets P_i^{ok} and P_i^{bad} contain correct and incorrect patches, respectively, excluding the separately evaluated reference patch. As in Section 2, a correct label records harness acceptance rather than complete coverage of the issue specification.

The record must support three complementary checks, because a successful reference transition alone does not establish useful candidate diversity. **(i) Reference quality:** the official tests establish a meaningful buggy-to-reference transition. **(ii) Correct-patch diversity:** correct patches allow tests to be evaluated across different implementations. **(iii) Incorrect-patch diversity:** incorrect patches expose plausible ways to leave issue requirements unsatisfied.

A.2 Reference-Test Quality

We apply the same LLM-based audit across all three benchmark sources. For each task, the audit jointly examines the issue description, reference patch, and official tests, and assigns a quality score from 0 to 100 based on assertion strength, avoidance of implementation-specific checks, whether the tests fail on the buggy program, and relevance to the reported issue. Using a threshold of 80, the quality audit retains 461 SWE-bench Pro tasks, 278 SWE-bench Multilingual tasks, and 110 DeepSWE tasks. We additionally manually review the audit outcomes before finalizing task eligibility. Together, the LLM-based audit and human review provide quality control over the official tests used as the execution oracle for candidate verification.

The quality audit determines task eligibility only; candidate-patch labels and generated-suite scores are subsequently determined through execution.

A.3 Reference-Transition Difficulty

We further filter tasks using a heterogeneous panel of test-generation configurations, removing tasks whose reference fail-to-pass transition is consistently reproduced across the panel. This filtering step removes tasks that are broadly easy for existing test-generation systems and retains tasks with nontrivial reference-transition difficulty.

After both curation stages, the benchmark retains 342 SWE-bench Pro tasks, 167 SWE-bench Multilingual tasks, and 110 DeepSWE tasks, for a total of 619 tasks. Table 4 summarizes the resulting task counts by source.

Table 4: Task-level curation by benchmark source.

Source	Initial	Removed	Retained (%)
SWE-bench Pro	731	389	342 (46.8)
SWE-bench Multilingual	300	133	167 (55.7)
DeepSWE	113	3	110 (97.3)
Overall	**1,144**	**525**	**619 (54.1)**

A.4 TASK-CURATION JUDGE PROMPT

We use an LLM-based quality audit to determine whether each candidate issue provides sufficiently reliable evaluation evidence for inclusion in SWEPATCHTEST. For each issue, the judge examines the issue description, reference patch, and official test patch. The audit evaluates whether the required behavior is objectively testable, whether the official tests meaningfully enforce that behavior, whether the assertions are sufficiently strong, whether the expected buggy-to-repaired transition is supported, and whether the tests remain independent of the particular implementation in the reference patch. The resulting recommendation is `include`, `review`, or `exclude`; cases with incomplete or ambiguous evidence are referred for manual review.

Figure 4 presents the task-curation prompt used to organize these judgments. The displayed version condenses repeated explanations while preserving the judge's inputs, evaluation criteria, decision policy, and output requirements.

System Prompt for Task Curation

- ROLE & OBJECTIVE -

You are a strict senior software engineer auditing candidate issues for inclusion in a software-engineering benchmark. Given an issue description, a reference patch, and an official test patch, determine whether the task provides a reliable and implementation-independent evaluation oracle.

Your goal: retain tasks whose tests meaningfully enforce the reported behavior while allowing valid alternative repairs. The reference patch is only one correct implementation; judge externally observable behavior, not internal code structure. When evidence is genuinely ambiguous, request human review.

Input evidence:

- Issue / problem statement: required behavior and scope.
- Reference patch: one official implementation of the repair.
- Official test patch: assertions intended to distinguish buggy from correctly repaired behavior.

- EVALUATION CRITERIA -

A. Issue clarity and alignment. Is the required behavior objectively testable? Does the reference patch address it, and do the official tests directly exercise and assert that behavior rather than unrelated effects?

B. Assertion strength. Could an incomplete or incorrect repair still pass? Flag tests that only check execution, nonempty output, broad types, incidental examples, setup behavior, or permissive conditions.

C. Implementation independence. Would another valid repair fail because the test encodes private helpers, exact call sequences, internal control flow, patch location, incidental logs or ordering, source matching, or copied constants?

D. Reference-transition reliability. Does the evidence support tests(Buggy) = fail and tests(Correct repair) = pass? Separate behavioral failures from compilation, dependency, timeout, flaky, external-service, or infrastructure failures.

- DECISION POLICY -

1. INCLUDE: clear, testable behavior; aligned and meaningful assertions; sufficient strength; alternative valid implementations remain admissible.

2. REVIEW: evidence is incomplete or genuinely ambiguous, or the judgment depends on repository context absent from the supplied artifacts.

3. EXCLUDE: issue-test mismatch, materially weak tests, reference-patch overfitting, unsupported fail-to-pass transition, or no objectively assessable requirement.

- OUTPUT FORMAT -

Return only one JSON object containing: an evidence-based summary; required and tested behavior; Boolean judgments for clarity, alignment, strength, implementation independence, and transition support; concrete evidence for weak tests, overfitting, or unrelated-failure risk; 0--100 component scores; decision = include | review | exclude; a primary reason; and confidence in [0, 1].

Task-specific fields:

`ISSUE_DESCRIPTION`

`REFERENCE_PATCH`

`OFFICIAL_TEST_PATCH`

Final instruction: Evaluate whether this task provides sufficiently reliable and implementation-independent test evidence for benchmark inclusion. Return only the required JSON object.

Figure 4: Simplified system prompt for the task-curation judge.

B CANDIDATE-POOL CONSTRUCTION

Patch producers and model roles. Table 5 lists the model inventory used to collect candidate patches. Candidate generation, task-selection voting, and the main test-generation comparison are separate uses of these models; a model's appearance in the inventory does not imply that it participates in every stage.

Table 5: Model families used for candidate patch collection.

Model family	Model configurations
Claude	Claude-Fable-5.1, Claude-Opus-5, Claude-Sonnet-5, Claude-Fable-5, Claude-Opus-4.8
GPT	GPT-6-Astra, GPT-5.6-Sol, GPT-5.4, GPT-5.2
DeepSeek	DeepSeek-V4-Pro, DeepSeek-V4-Flash
Gemini	Gemini-3.8-Flash, Gemini-3.1-Pro, Gemini-3-Pro, Gemini-2.5-Pro, Gemini-2.5-Flash
GLM	GLM-5.3, GLM-5.2
Qwen	Qwen3.8-Max, Qwen3.8-27B, Qwen3.6-35B-A3B, Qwen3.5-35B-A3B
Kimi	Kimi-K3

Normalizing diffs. Deduplication operates independently within each task and correctness class. We strip hunk line numbers, index lines, trailing whitespace, and unchanged context, retaining the added and deleted lines. This normalization reduces differences caused solely by diff formatting.

Filtering similar candidates. We filter the normalized diffs at character-level TF–IDF similarity $\tau = 0.9$. Inverse-document-frequency weighting downweights features shared across many patches (SPARCK JONES, 1972). The feature representation uses character n-grams with $n \in \{3, 4, 5\}$. We process diffs from longest to shortest, discarding a candidate if its cosine similarity to any retained diff is at least 0.9. Character features avoid language-specific tokenizers, although textual distance remains an approximation to implementation diversity.

Capping each class. If more than 128 candidates remain in a class, farthest-point sampling selects 128 spread across the feature space. Both classes use the same cap; the larger incorrect pool reflects the greater number of distinct incorrect patches retained. Final curation removes candidates overlapping official test hunks and entries without a usable solution artifact. The final pools contain 82,579 candidates: 27,749 correct and 54,830 incorrect patches. Table 6 reports their counts and per-task means over all 619 tasks.

Table 6: Candidate-pool composition by correctness class.

Patch label	Count	Average per task
Correct patches	27,749	44.83
Incorrect patches	54,830	88.58
All candidate patches	**82,579**	**133.41**

C EVALUATION PROTOCOL AND METRICS

C.1 GENERATED-SUITE PROTOCOL

Extracting and isolating artifacts. The submitted package consists of a test script, fixtures, and supporting files in a task-local `tmp` directory, without changes to existing repository files. Separating this package from the official test patch avoids conflicts that would confound applicability with test quality. We extract the script and referenced files directly from `tmp`, rather than reconstructing the package from a Git diff. Identical packages are installed in clean copies of the buggy codebase and codebases obtained by applying the reference or a candidate patch. We reset between executions so that fixtures, caches, and side effects cannot carry over to the next program state.

Required outcomes. Using the execution function defined in Section 2.5, the desired suite outcomes are

$$\begin{aligned} \mathrm{exec}(T_i, R_i) &= \mathrm{F}, & \mathrm{exec}(T_i, R_i \circ p_i^{ref}) &= \mathrm{P}, \\ \mathrm{exec}(T_i, R_i \circ p) &= \mathrm{P} \quad (p \in P_i^{ok}), & \mathrm{exec}(T_i, R_i \circ p) &= \mathrm{F} \quad (p \in P_i^{bad}). \end{aligned} \tag{6}$$

Suite-level acceptance requires every submitted test to pass. Individual tests can also be replayed to identify the assertions responsible for a suite verdict.

Recording outcomes. For each suite execution, logs record the command, exit code, timeout state, failing assertions, and harness output. Outcomes distinguish completed passes, stable behavioral failures, timeouts, infrastructure failures, and missing output. We replay inconsistent outcomes and withhold both acceptance and rejection credit if instability persists. Flaky tests can produce different outcomes for unchanged code (Luo et al., 2014); execution statuses must therefore be examined before assigning a behavioral explanation.

C.2 Metric Definitions and Scoring Conventions

Every benchmark rate retains the same denominator of $N = 619$ tasks. Equation 1 defines the indicators from exec: $a_i(p) = 1$ when the suite passes on $R_i \circ p$, and $r_i(p) = 1$ when a completed run produces a stable behavioral failure. Unresolved executions, missing artifacts, and infrastructure-only outcomes are recorded as $\perp$ and set both indicators to zero. Within-task denominators use the released pool sizes, preventing failed executions from improving a score by disappearing from its denominator.

Reference recognition: GRR. Gold Recognition Rate averages the reference fail-to-pass indicator over tasks:

$$\begin{aligned} g_i &= \mathbf{1}[\mathrm{exec}(T_i, R_i) = \mathrm{F} \ \wedge \ \mathrm{exec}(T_i, R_i \circ p_i^{ref}) = \mathrm{P}], \\ \mathrm{GRR} &= \frac{1}{N} \sum_{i=1}^{N} g_i. \end{aligned} \tag{7}$$

We set $g_i = 0$ when the suite or either required transition outcome is unavailable. This check establishes reference reproduction without imposing verdicts on candidate patches.

Candidate discrimination: WDR and CAR. For each nonempty pool, we compute the fraction of incorrect patches rejected or correct patches accepted:

$$w_i = \frac{\sum_{p \in P_i^{bad}} r_i(p)}{|P_i^{bad}|}, \qquad c_i = \frac{\sum_{p \in P_i^{ok}} a_i(p)}{|P_i^{ok}|}. \tag{8}$$

WDR and CAR are the macro averages of these within-task fractions:

$$\mathrm{WDR} = \frac{1}{N} \sum_{i=1}^{N} w_i, \qquad \mathrm{CAR} = \frac{1}{N} \sum_{i=1}^{N} c_i. \tag{9}$$

An issue contributes the same weight regardless of how many candidates it contains.

WDR evaluates rejection of incorrect patches, whereas CAR evaluates preservation of correct patches. A suite that rejects all candidates can score highly on WDR while failing CAR, illustrating why either rate alone is insufficient. Execution failures also lower these rates, so behavioral diagnoses must be checked against the recorded statuses.

Complete-task discrimination: CT-WDR, CT-CAR, and JVR. Complete-task rates count only issues on which the suite rejects every incorrect patch or accepts every correct patch:

$$\text{CT-WDR} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[w_i = 1], \qquad \text{CT-CAR} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[c_i = 1]. \tag{10}$$

JVR further requires reproduction and both complete-task conditions to hold together:

$$\mathrm{JVR} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{1}[g_i = 1 \ \wedge \ w_i = 1 \ \wedge \ c_i = 1]. \tag{11}$$

JVR determines the reported ranking, while CT-WDR and CT-CAR identify the class that prevents complete validation. Complete-task comparisons keep the pools fixed because adding or diversifying candidates can introduce further opportunities for failure.

Balanced summaries: F1 and CT-F1. The two balanced summaries use harmonic means of the corresponding rates:

$$\begin{aligned} \mathrm{F1} &= h(\mathrm{WDR}, \mathrm{CAR}), \qquad \mathrm{CT\text{-}F1} = h(\mathrm{CT\text{-}WDR}, \mathrm{CT\text{-}CAR}), \\ h(a,b) &= \frac{2ab}{a+b} \quad (a+b>0), \quad h(0,0) = 0. \end{aligned} \tag{12}$$

These summaries combine benchmark rates rather than pooled classifier precision and recall. They reward aggregate acceptance–rejection balance but do not impose the task-level conjunction measured by JVR.

C.3 Agent Configurations

The main benchmark evaluates each model with its native coding agent where available: GPT-5.6-sol uses Codex; the Claude Fable and Opus models use Claude Code; Kimi-K3 uses Kimi Code; Qwen3.8-max uses Qwen Code; and GLM-5.3 uses ZCode. All four DeepSeek-V4 configurations, including Pro, Flash, and their Preview variants, use OpenHands. These agents generate test packages under the common artifact and execution protocol above, and all configurations are scored with the same benchmark metrics. The four direct SFT evaluations also use OpenHands, while downstream repair evaluation uses Scale SWE. Figure 1 shortens GPT-5.6-sol to GPT-5.6, Claude models to Fable/Opus, DeepSeek-V4 to DS, Preview to prv, and Qwen3.8-max to Qwen3.8.

D RQ1: Additional Results and Analysis

D.1 Benchmark Leaderboard Reliability

Resampling protocol. We quantify uncertainty over the benchmark task set using a paired, source-stratified nonparametric bootstrap (Efron, 1979). Each of the 10,000 bootstrap replicates samples tasks with replacement within the three benchmark sources: 342 SWE-bench Pro tasks, 167 SWE-bench Multilingual tasks, and 110 DeepSWE tasks. The same sampled task indices are applied to all configurations, preserving both the source composition and the paired structure of model comparisons. For every replicate, we recompute GRR, WDR, CAR, CT-WDR, CT-CAR, and JVR from the sampled task-level outcomes. F1 and CT-F1 are subsequently recomputed as the harmonic means of WDR and CAR and of CT-WDR and CT-CAR, respectively. We report two-sided percentile 95% confidence intervals.

The RQ1 analysis covers the eight representative coding-agent configurations included in the main benchmark comparison (Table 2). Table 7 reports uncertainty intervals for all eight benchmark metrics. Each cell gives the point estimate and its 95% confidence interval as percentages. Point estimates match Table 2; intervals use unrounded task-level values.

Paired JVR comparisons. We compare each representative configuration with GPT-5.6-sol, the point-estimate leader on JVR. For each competitor, $\Delta\mathrm{JVR}$ denotes the competitor's JVR minus that of GPT-5.6-sol in percentage points under the paired stratified bootstrap. Because JVR is binary at the task level, we additionally apply an exact two-sided McNemar test to the discordant task outcomes (Fagerland et al., 2013). The seven resulting p-values are adjusted using Holm's procedure. Table 8 lists discordant task counts as GPT-5.6-sol-only and competitor-only JVR successes, respectively. Bootstrap rank stability is computed in every replicate; ties for first divide the corresponding first-place probability equally.

GPT-5.6-sol has the highest observed JVR at 9.21% and receives 84.5% of the bootstrap first-place probability. Claude-Fable-5.1 is the closest configuration, with a JVR difference of -1.29 percentage points and a 95% confidence interval of $[-3.72, +0.97]$. The paired interval and rank distribution therefore provide additional context for the point-estimate ordering in Table 2.

Table 7: Bootstrap estimates and 95% confidence intervals for the eight RQ1 coding-agent configurations.

Reference and patch-level metrics

Model	GRR	WDR	CAR	F1
GPT-5.6-sol	59.29 [55.74, 62.84]	88.57 [86.59, 90.44]	50.20 [47.20, 53.20]	64.08 [61.50, 66.54]
Claude-Fable-5.1	65.11 [61.55, 68.66]	83.74 [81.59, 85.82]	62.19 [59.14, 65.18]	71.37 [69.14, 73.51]
Claude-Opus-5	68.17 [64.62, 71.57]	85.91 [84.15, 87.59]	66.76 [63.96, 69.54]	75.13 [73.34, 76.90]
Kimi-K3	33.12 [29.56, 36.51]	80.24 [77.47, 82.95]	40.64 [37.33, 43.87]	53.95 [50.91, 56.80]
Qwen3.8-max	49.60 [45.88, 53.31]	84.69 [82.81, 86.46]	61.68 [58.70, 64.65]	71.38 [69.38, 73.25]
DeepSeek-V4-Pro	36.03 [32.31, 39.90]	82.53 [80.29, 84.74]	49.36 [45.91, 52.78]	61.77 [58.97, 64.39]
DeepSeek-V4-Flash	29.89 [26.33, 33.45]	80.53 [77.93, 82.98]	44.62 [41.31, 47.98]	57.42 [54.55, 60.17]
GLM-5.3	38.77 [34.89, 42.65]	85.36 [83.38, 87.32]	53.55 [50.35, 56.70]	65.81 [63.38, 68.17]

Complete-task metrics

Model	CT-WDR	CT-CAR	CT-F1	JVR
GPT-5.6-sol	47.66 [43.78, 51.53]	28.59 [25.20, 31.99]	35.74 [32.77, 38.52]	9.21 [6.95, 11.47]
Claude-Fable-5.1	34.89 [31.34, 38.45]	30.37 [26.82, 33.93]	32.47 [29.86, 34.91]	7.92 [5.82, 10.02]
Claude-Opus-5	32.47 [28.92, 36.03]	31.34 [27.79, 35.06]	31.89 [29.40, 34.37]	7.11 [5.17, 9.21]
Kimi-K3	49.43 [45.40, 53.47]	22.94 [19.71, 26.17]	31.34 [28.16, 34.26]	5.98 [4.20, 7.92]
Qwen3.8-max	32.47 [28.92, 36.03]	30.21 [26.66, 33.76]	31.30 [28.84, 33.60]	5.17 [3.55, 6.95]
DeepSeek-V4-Pro	36.35 [32.63, 40.06]	27.95 [24.56, 31.34]	31.60 [29.08, 33.94]	4.52 [3.07, 6.14]
DeepSeek-V4-Flash	40.87 [37.00, 44.75]	22.94 [19.71, 26.17]	29.39 [26.52, 32.10]	3.88 [2.42, 5.49]
GLM-5.3	37.64 [33.93, 41.36]	25.04 [21.65, 28.43]	30.07 [27.42, 32.58]	3.72 [2.42, 5.33]

Table 8: Paired JVR comparisons and bootstrap rank stability for RQ1.

Model	ΔJVR [95% CI]	Discordant tasks	p_{Holm}	First (%)	Rank [95% interval]
GPT-5.6-sol	+0.00 [+0.00, +0.00]	–	–	84.5	1.0 [1.0, 2.5]
Claude-Fable-5.1	-1.29 [-3.72, +0.97]	32 / 24	0.350	13.3	2.0 [1.0, 4.0]
Claude-Opus-5	-2.10 [-4.36, +0.16]	35 / 22	0.222	2.0	3.0 [2.0, 4.0]
Kimi-K3	-3.23 [-5.65, -0.81]	41 / 21	0.045	0.2	4.0 [2.0, 6.0]
Qwen3.8-max	-4.04 [-6.46, -1.62]	42 / 17	0.006	0.0	5.0 [4.0, 7.0]
DeepSeek-V4-Pro	-4.68 [-7.11, -2.26]	44 / 15	0.001	0.0	6.0 [4.0, 8.0]
DeepSeek-V4-Flash	-5.33 [-7.59, -3.07]	44 / 11	$< .001$	0.0	7.0 [5.5, 8.0]
GLM-5.3	-5.49 [-7.75, -3.23]	45 / 11	$< .001$	0.0	8.0 [5.5, 8.0]

D.2 Performance by Task Source

Table 9 expands the source comparison in RQ1 with the configuration and component rates for each source-specific JVR leader. Each percentage is computed within the corresponding source split. Issue composition and candidate-pool coverage vary across splits, with DeepSWE showing the lowest best JVR. These differences preclude attributing the score gap to trajectory length from the split comparison alone.

Table 9: Configurations with the highest JVR in each task source.

Source	Configuration	JVR	GRR	WDR	CAR	CT-WDR	CT-CAR
Multilingual (n=167)	GPT-5.6-sol	16.17	87.43	82.57	79.19	38.92	50.30
Pro (n=342)	Claude-Fable-5	8.77	67.25	84.23	66.19	30.70	32.16
DeepSWE (n=110)	GPT-5.6-sol	2.73	38.18	96.82	34.26	44.55	12.73

D.3 Qualitative Audits

From a disagreement to its cause. Complementary studies diagnose failure development in coding-agent trajectories (Li et al., 2026a; Zhao et al., 2026). A disagreement audit collects the issue, repository, required behavior, candidate changes, generated assertions, and execution traces. When a suite accepts an incorrect patch, we inspect which required behavior remains unchecked. When it rejects a correct patch, we check whether the assertion reveals an oracle gap or unnecessarily constrains the implementation. The collected pairs below establish patch diversity; their possible test failures illustrate audit questions and are not measurements of generated-suite errors.

Repairing different layers: Redis. For `redis__redis-13115`, Lua can format a numeric argument in scientific notation, but `HINCRBY` expects a decimal integer string. Two correct patches have normalized cosine similarity 0.017: one changes formatting in `src/script_lua.c`, while the other changes a conversion condition in `deps/fpconv/fpconv_dtoa.c`. Invoking Lua and checking the stored value would accommodate either repair location. An assertion requiring a particular helper call would instead make acceptance depend on implementation structure.

Repairing one function in different ways: Hugo. For `gohugoio__hugo-12343`, URL parsing can treat the portion after # as a fragment and truncate the path. Two correct `PathEscape` patches have normalized cosine similarity 0.043: one escapes # before parsing, while the other escapes individual path segments before reconstruction. The shared edit location therefore does not imply a shared repair strategy. The harness supplies both labels without proving equivalence on every possible path input.

D.4 Sensitivity to the Generation Prompt

Paired normal/ARH runs test whether the prompt variant improves the measured capabilities together. For Opus 5 at epoch 2, ARH slightly increases WDR and CAR while reducing GRR and JVR; the other reported pairs favor or tie the normal prompt on JVR. The use of ARH in patch collection therefore does not imply a consistent advantage when generating tests.

E SFT Data and Downstream Evaluation

SFT follows the model, teacher, and five-epoch training setup described in RQ2; downstream evaluation uses Scale SWE. Table 3 compares four training configurations: SWE SFT, + TestGen Supervision, + Validation-guided TestGen, and + Scaled TestGen Supervision.

E.1 RQ2: Training Data and Selection

Data separation. We enforce repository-level disjointness between all SFT training data and evaluation tasks in SWEPatchTest, SWE-bench Verified, SWE-bench Multilingual, and SWE-bench Pro.

Training-data composition. The SWE SFT baseline contains 1.9 billion tokens of repository-level repair trajectories. TestGen Supervision and Validation-guided TestGen each add 2 billion tokens of TestGen trajectories selected from the same pool, constructed from approximately 20,000 issue environments. TestGen Supervision selects trajectories using GRR, whereas Validation-guided TestGen uses WDR/CAR/JVR-based selection. Scaled TestGen Supervision adds a further 2 billion tokens of validation-selected TestGen trajectories from an expanded pool, bringing the total TestGen data to 4 billion tokens.

Evaluation and selection protocol. The TestGen metrics in Table 3 are evaluated on the final 619-task benchmark. Data selection follows the GRR and WDR/CAR/JVR criteria described in RQ2.

What the training comparison isolates. The unscaled comparison holds the source trajectory pool and the 2-billion-token TestGen budget fixed while changing selection criteria. Scaled supervision additionally changes the data quantity and environment pool, so its gains do not isolate selection from scaling or changes in data composition.

E.2 RQ2: Test-Generation Reliability

The RQ2 analysis covers the four SFT configurations reported in Table 3. We apply the same paired, source-stratified bootstrap to their task-level test-generation outcomes. Table 10 reports uncertainty intervals for all eight SWEPatchTest metrics. Cells use the same percentage convention as Table 7, with point estimates matching Table 3 and intervals computed from unrounded task-level values.

Table 10: Bootstrap estimates and 95% confidence intervals for the four RQ2 SFT configurations.

Reference and patch-level metrics

Configuration	GRR	WDR	CAR	F1
SWE SFT	8.89 [6.79, 11.15]	20.75 [17.91, 23.66]	25.28 [22.10, 28.41]	22.79 [20.44, 25.08]
TestGen Supervision	39.26 [35.70, 42.81]	59.00 [56.40, 61.61]	43.98 [40.76, 47.16]	50.39 [48.05, 52.70]
Validation-guided TestGen	40.39 [36.83, 43.94]	64.37 [61.40, 67.27]	50.27 [47.03, 53.54]	56.45 [53.99, 58.81]
Scaled TestGen Supervision	41.20 [37.64, 44.91]	72.14 [69.62, 74.69]	51.62 [48.19, 55.05]	60.18 [57.75, 62.50]

Complete-task metrics

Configuration	CT-WDR	CT-CAR	CT-F1	JVR
SWE SFT	9.21 [6.95, 11.47]	19.39 [16.48, 22.46]	12.49 [10.26, 14.56]	1.29 [0.48, 2.26]
TestGen Supervision	13.73 [11.15, 16.32]	28.11 [25.04, 31.18]	18.45 [15.86, 20.79]	2.26 [1.29, 3.39]
Validation-guided TestGen	25.69 [22.46, 29.08]	31.50 [28.27, 34.73]	28.30 [25.87, 30.65]	3.23 [1.94, 4.68]
Scaled TestGen Supervision	31.34 [27.79, 34.89]	32.96 [29.56, 36.35]	32.13 [29.77, 34.37]	3.88 [2.42, 5.49]

Paired comparisons with the SFT baseline. We compare each test-generation training configuration with SWE SFT. Here, ΔJVR denotes the trained configuration's JVR minus the SWE SFT baseline in percentage points under the paired stratified bootstrap. We apply exact two-sided McNemar tests to the paired task-level JVR outcomes and adjust the three planned comparison p-values using Holm's procedure. Table 11 lists discordant task counts as baseline-only and compared-configuration-only JVR successes, respectively. Rank stability is computed jointly over the four SFT configurations.

Table 11: Paired JVR comparisons and bootstrap rank stability for RQ2.

Configuration	ΔJVR [95% CI]	Discordant tasks	p_{Holm}	First (%)	Rank [95% interval]
SWE SFT	+0.00 [+0.00, +0.00]	–	–	0.0	4.0 [3.0, 4.0]
TestGen Supervision	+0.97 [-0.16, +2.10]	3 / 9	0.146	0.4	3.0 [2.0, 4.0]
Validation-guided TestGen	+1.94 [+0.81, +3.23]	2 / 14	0.008	17.2	2.0 [1.0, 3.0]
Scaled TestGen Supervision	+2.58 [+1.29, +4.04]	2 / 18	0.001	82.4	1.0 [1.0, 2.0]

The SFT configurations show consistent improvements over the SWE SFT baseline across the test-generation metrics in Table 10. Scaled TestGen Supervision achieves the highest observed JVR at 3.88% and receives 82.4% of the bootstrap first-place probability. Its paired JVR improvement over SWE SFT is 2.58 percentage points, with a 95% confidence interval of $[1.29, 4.04]$. Validation-guided TestGen also improves JVR over the baseline, with a paired difference of 1.94 percentage points and a 95% confidence interval of $[0.81, 3.23]$. These paired results support the gains of validation-guided and scaled supervision over SWE SFT.

E.3 RQ3: Downstream Evaluation

AVG@4 and Pass@4 follow the four-round definitions in RQ3: mean per-round resolution and resolution in at least one round, respectively. Table 3 reports AVG@4 and Pass@4 on SWE-bench Verified, Multilingual, and Pro.

F Records Needed for Reproduction

Reproducing a SWEPatchTest comparison requires three groups of records: **(1) Tasks and candidates:** identifiers, filtering decisions, source counts, producers, labels, final pool sizes, and empty-pool counts. **(2) Executions and scores:** suite counts, commands, statuses, timeouts, replays, metric numerators and denominators, and per-task results for macro averaging. **(3) Training and transfer:** checkpoint identities, data recipes, selection thresholds, retained trajectory manifests, and downstream attempt coverage. Improvements in test generation or repair must be supported by execution outcomes, rather than training loss or judge preferences alone.

G Limitations

What the oracle and execution can establish. Candidate labels follow the official oracle, providing an operational notion of correctness rather than a formal guarantee of semantic correctness. We mitigate oracle-quality concerns through task-level filtering of weak or implementation-dependent reference tests and manual quality checks. Generated tests may nevertheless exercise behavior beyond the official oracle. Timeouts, missing outputs, and applicability failures reduce fixed-denominator scores without necessarily evaluating behavioral assertions. Execution records therefore help distinguish behavioral misclassification from unresolved execution. Audits should connect test assertions and execution traces to issue requirements, distinguishing weaknesses in generated tests from gaps in official coverage.

Which issues and patches are represented. The reproduction panel defines task difficulty, and its model families partly overlap those evaluated on the final set. The selected challenge set therefore does not represent the full population of software issues. The 82,579 candidate patches cover observed attempts, not every possible implementation or fault, and textual deduplication does not guarantee semantic diversity. DeepSWE's thinner correct pools further constrain conclusions about correct-patch acceptance.