SWE-SWEEP: CAN AGENTS AUTONOMOUSLY FIND AND FIX BUGS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Most existing software engineering benchmarks evaluate coding agents on concrete, well-specified tasks, commonly by providing a codebase together with a user-reported issue to resolve. However, as users delegate increasingly broad outcomes to coding agents, the natural next step is for agents to determine not only how to perform useful work, but also what useful work needs to be done. An agent entrusted with a repository should be able to decide what is broken, which problems matter, and how to solve them before they are reported. We introduce SWE-sweep, a benchmark for this open-ended setting. Given a codebase containing many concurrent bugs and no information about their nature or location, an agent must autonomously discover and fix as many bugs as possible. For each repository, we collect real GitHub issue–pull request pairs, then identify a single commit where the maximum number of bugs are present at the same time. Each repair is evaluated against hidden tests from the corresponding pull request, along with the existing test suite to check for regressions. SWE-sweep encompasses 4,068 bugs across 22 programming languages and 100 repositories, including NumPy (Python), OpenCV (C++), and Lean 4 (Lean). We evaluate 7 LMs and find that the best agent resolves only 4.7% of all bugs, with the challenge generally lying in discovering bugs rather than repairing them. SWE-sweep exposes a substantial gap between resolving reported issues and autonomously maintaining software.

1 INTRODUCTION

Language Models (LMs) have become remarkably capable at software engineering. Given a concrete task, today's agents can navigate large codebases and produce substantial functional changes. Typical tasks include fixing bugs, developing features, and completing tasks in a terminal environment (Jimenez et al., 2024; Merrill et al., 2026). As LMs' coding abilities have improved, they have been applied to increasingly demanding tasks, including implementing significant features (Cursor Research, 2026; Huang et al., 2026), performing project-scale work (Desai et al., 2026), and writing entire programs from scratch (Ding et al., 2026; Yang et al., 2026a).

But increasing the amount and complexity of defined work that agents are asked to carry out is only one dimension of making them more useful as tools for developers. A complementary direction is to ask agents to operate with increasingly little specification and identify useful work themselves. For example, rather than waiting for a developer to notice, diagnose, and turn a software failure into an open bug report, such an agent would proactively inspect the codebase, find issues, and repair them.

Unlike issue-conditioned repair, much of the difficulty of this setting arises before implementation begins. Agents must systematically explore a large codebase and distinguish genuine defects from unusual but intended behavior, all before the more familiar task of repairing them can begin. The capabilities exercised in this setting, including reasoning about user intent, identifying subtasks, and working on interrelated problems throughout a long trajectory, also transfer to any open-ended software engineering tasks where the agent pursues broad objectives.

We introduce SWE-sweep to study this setting. In SWE-sweep, an agent is given an entire codebase and asked to identify and resolve as many bugs as possible with no hints about the nature or location of any bugs. The agent must do this without introducing regressions. We score the submitted patch against a reference set of previously identified bugs, counting how many the agent successfully repairs. Unlike our zero information setting, prior work (Li et al., 2026; Liu et al., 2026; Lee et al.,

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

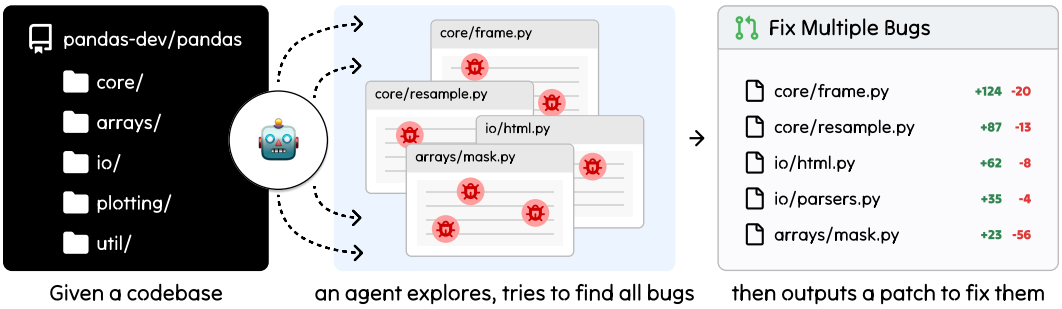


Figure 1: SWE-sweep evaluates coding agents on their ability to identify and fix *unspecified* bugs within a software repository. An agent is asked to scour a codebase for issues and resolve them without *any* information about specific faults. The agent is then graded using a suite of hidden tests.

2026) typically provides agents with substantial information about the bugs it needs to fix, such as failure types, failure signals, or where the bugs occur.

SWE-sweep tasks are constructed by crawling and filtering repositories from GitHub. Per repository, we collect all pull requests (PRs) with a linked issue, then retain those that ship tests and fix a bug. We then identify a commit of the repository where the most bugs are concurrently present, and use it as the starting state of a task instance. Each bug's PR tests are used to verify whether it has been fixed. Besides many quality filters shared with other benchmarks, we apply extensive filtering to retain only bugs that can be discovered from reading the repository alone. While any history-derived benchmark necessarily under-counts the bugs present in a repository, restricting SWE-sweep to defects confirmed by an upstream fix provides strong evidence that the observed behavior was considered erroneous by the repository maintainers.

SWE-sweep is made up of 100 repositories that contain a total of 4,068 bugs. The final test set covers 22 unique programming languages and includes notable libraries such as pandas, Lean 4, and scikit-learn. We evaluate 7 LMs, including GPT-5.6 Sol and Claude Opus 5. Even the strongest model, GPT-5.6 Sol (xhigh), resolves only 4.7% of all bugs, leaving substantial room for improvement in autonomous bug discovery and repair. Across several analyses, agents are considerably more successful at repairing discovered bugs than at finding them.

We open source SWE-sweep to enable the community to reproduce and build on our investigations.

2 SWE-SWEEP

We first formalize the SWE-sweep task in §2.1. We then discuss the construction process in §2.2, and finally review benchmark statistics in §2.3 and features in §2.4.

2.1 TASK FORMULATION

Given a code repository, a coding agent is asked simply to discover and fix all bugs in the repository. The agent is not given *any* additional information about any specific bug's location, description, or tests. The agent is expected to modify the codebase such that all bugs are resolved.

Evaluation. To evaluate correctness, we run the agent's submitted codebase against two sets of tests. First, we restore the repository's original test suite and run it to verify no existing behavior was broken. Second, for each bug, we run a set of hidden tests; at least one of these tests fails on the unmodified codebase and passes once the bug is fixed (fail-to-pass). A bug is considered resolved if all its hidden tests pass and the original suite still passes.

2.2 CONSTRUCTION

For each repository, construction proceeds in three stages: First, we collect bugs from a real-world software project. Second, we select a suitable base commit that maximizes the number of concurrent bugs. Third, we apply additional repair and filter operations. Full details appear in §B.

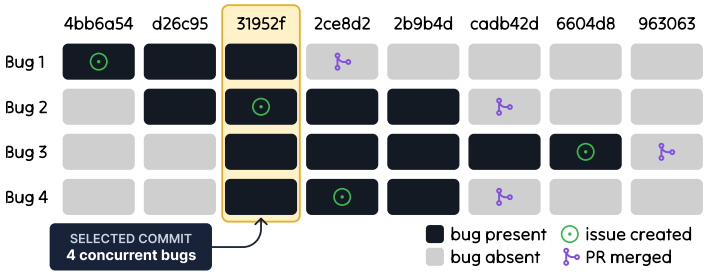


Figure 2: SWE-sweep tasks are constructed from a repository's *buggiest* commit. Rows represent bugs, columns are commits; dark cells indicate a bug is present at a commit. The `31952f` codebase becomes the task start state (*base commit*).

Table 1: SWE-sweep construction pipeline. Bug candidate counts correspond to the total number of bugs retained per stage across all 100 repositories represented in the final evaluation dataset.

Stage	Bug Candidates
Candidates	704,386 (100%)
Pre-filter	146,520 (20.8%)
Patches apply	19,470 (2.8%)
Reproduces	9,057 (1.3%)
Quality control	4,068 (0.6%)

Identify usable repositories. For conversion to be feasible, a GitHub repository must have a test suite along with an ample ledger of issues and pull requests. We select 100 codebases that satisfy these requirements and collectively represent a diverse mix of programming languages.

Source bugs from real-world issue–PR pairs. Per repository, we crawl all closed issues and merged pull requests (PRs). Echoing SWE-bench's collection strategy (Jimenez et al., 2024), we identify *bug candidates* as PRs with an associated issue, then check each PR for a meaningful fix (*gold fix*) and accompanying tests (*hidden tests*). Using an LM, we filter the collected PRs to retain those explicitly described as bug fixes in the associated descriptions or issue texts. Table 1 summarizes candidate attrition across the construction stages.

Estimate bug lifetime. To find commits in which many bugs coexist, we estimate each bug's lifetime using the SZZ algorithm (Śliwerski et al., 2005). A bug's lifetime ends at the merge commit of the corresponding PR that resolves it. To estimate where it begins, we `git blame` the lines modified by the PR to identify the commit that last touched them. Aggregating these estimated lifetimes produces a coverage curve over repository history, indicating how many candidate bugs may be present at each commit. Figures 25 and 26 show these curves for all benchmark repositories.

Select a base commit with the maximum number of concurrent bugs. Given each bug's lifetime, we can create a timeline of how many bugs exist simultaneously at any commit, as visualized in Figure 2. From this timeline, we identify several base commits with many bugs.

Per base commit, we check how many of the bugs are reproducible: the gold fix must apply, fail-to-pass hidden tests fail when the gold fix is not applied, and all hidden tests must pass when the gold fix is applied. Frequently, the patches for bugs do not apply cleanly to a base commit; we use an LM to rebase such patches onto the base commit without changing their behavior (details in §B.5). Per repository, we then select one base commit with the largest number of reproducible bugs.

Additional quality control. We then run additional quality control checks on the bug instance candidates to ensure that each corresponds to a single unit of work, that its bug is discoverable from reading the repository alone, and that its tests are not overly specific. We furthermore ensure that solving the bug instances does not contradict other bug instances or any of the base commit's tests. This quality control is performed with a mix of deterministic checks and several stages of LM annotations. For all annotations, we employ Gemini 3.1 Pro, Opus 5 (high), and Sol 5.6 (high) and require unanimous agreement between the three models. Appendices B.1 and B.6 give the full criteria and quality-control procedure; Appendix A.4 reports supporting validation studies. We acknowledge it is certainly possible these filters end up excluding meaningful bugs; our checks do not assess the importance or difficulty of a bug. That said, we intentionally trade away some coverage to ensure identified bugs can be graded fairly.

What makes a bug *discoverable*? We define a bug as behavior that diverges from the software's expected behavior (IEEE, 1990; ISO/IEC/IEEE, 2017; Avizienis et al., 2004), and call it *discoverable* when this expectation can be inferred from the repository alone, such as from contracts defined in the documentation, types, tests, referenced standards, and well-defined mathematical or scientific semantics (Herzig et al., 2013; Antoniol et al., 2008).

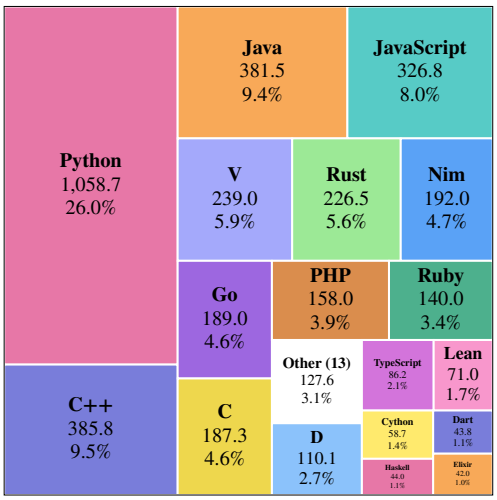


Figure 3: Bug counts and ratios by language, measured from the files modified by gold fixes. Details in Table 28.

Table 2: Repository statistics. The lower part of the table is measured at the base commit.

	Min	Mean	Max
GitHub stars	819	25.4k	137k
Contributors	10	690.8	5,605
# bugs	4	40.7	239
LoC	6k	292k	1.9M
Files	10	1,026	6,861
# bugs / 100k LoC	0.8	46.3	254.5

Table 3: Bug statistics. See Section 2.1 for definitions and Figure 24 for distributions.

	Min	Mean	Max
# F2P tests	1	1.6	268
# P2P tests	0	50.4	19,230
Gold fix lines	1	16.3	3,905
Test patch lines	1	37	2,533

To ensure that every bug in SWE-sweep is discoverable, we operationalize these definitions via a two-stage annotation effort. First, we query three LMs (Gemini 3.1 Pro, Opus 5, Sol 5.6) three times each to judge if a bug is discoverable, only keeping those that are unanimously accepted. This filter rejects 40% of bug candidates; examples of accepted and rejected bugs are in Appendix B.7. Second, four human reviewers independently audit a random sample of 50 bugs from the final benchmark. For 98% of sampled bugs, 3 of 4 reviewers agreed with the LM's determination that the bug is discoverable. Details about the annotation effort parameters and outcomes can be found in Appendix A.4.

2.3 Statistics

SWE-sweep's final test set consists of 100 tasks created from open-source GitHub repositories spanning 22 languages (Figure 3). The repositories covered represent a range of utilities, from small single-purpose libraries (e.g., date/time handling in `dayjs`, `pendulum`) to web and application frameworks (`rails`, `quarkus`), scientific computing (`numpy`, `astropy`, `qutip`), database engines (`duckdb`, `trino`), developer tooling (`babel`, `mypy`, `ruff`), and programming language toolchains (`go`, `roslyn`, `lean4`, `z3`). Repositories vary significantly across a number of dimensions, such as in size, from 6k to 1.9M non-test source code lines (Table 2). The test set represents a total of 4,068 bugs, where each repository contains between 4 and 239 bugs (Table 2). The gold fixes are similarly assorted in scope: a median fix edits 7 lines in 1 file, but the mean of 16.3 lines indicates a long tail of far larger repairs. Appendix D gives the full distributions and lists all repositories.

2.4 Features

Bugs lie dormant for a long time, well before their upstream fixes. At the repository state given to agents, the lines eventually repaired had already been unchanged for a median of 481 days, and the upstream fix did not arrive for another 293 days.

Different bugs lead agents to overlapping files but widely separated repair locations. More than half of bugs have a gold fix that modifies at least one file also modified by the gold fix for another benchmark bug. Yet same-file repairs almost always change distinct content and are usually more than 100 lines apart (Appendix Table 27). This structure reflects SWE-sweep's multi-bug design: agents can reuse repository context, but cannot solve the benchmark by repeating one type of repair.

Bugs are diverse in defect type and are discoverable from a variety of sources. We use an LM agent to review each bug in the context of its repository and annotate both its observable failure mode

Table 4: Main results, ordered by the percentage of bugs resolved. Cost, agent turns, and output tokens (*out tok*) are arithmetic means across repositories.

Model	% bugs resolved	Usage per repo $	turns	out tok
Sol 5.6 (xhigh)	**4.7%**	72.30	233	105k
Luna 5.6 (xhigh)	**2.5%**	2.24	204	61k
Terra 5.6 (xhigh)	**1.5%**	3.57	76	47k
Luna 5.6 (high)	**1.4%**	0.28	75	20k
Opus 5 (xhigh)	**1.3%**	53.63	323	193k
Kimi K3 (max)	**0.6%**	24.51	337	153k
Luna 5.6	**0.5%**	0.04	22	4k
GPT-5.4 Mini (high)	**0.5%**	1.22	75	41k
GPT-5.4 Mini	**0.2%**	0.05	13	2k
Gemini 3.5 Flash Lite	**0.1%**	0.06	34	5k

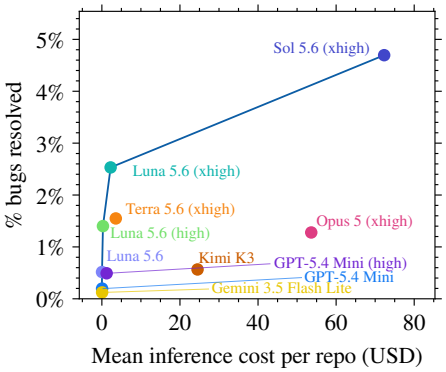


Figure 4: Bugs resolved versus inference cost; the line marks the Pareto frontier.

and the evidence showing that the behavior is incorrect. To avoid biases, the agent is only handed the hidden-test patch introduced in the PR corresponding to the bug, but no other information such as the issue description. The most frequent bug type causes the affected repository to directly produce incorrect outputs (41.3%), followed by issues related to validating various types of input (28.0%), and resource or runtime failures (22.3%). For 99.95% of labeled bugs, the agent finds evidence in the repository that shows that the observed behavior is incorrect. These repository contracts come from sources such as documentation, types, existing tests, and consistency within the repository. The remaining two bugs correspond to failures that are undesirable independent of the context (a race condition and an out-of-bounds read). Appendix A.4.2 reports the full breakdown of defect types.

3 Experiments

Models. We evaluate 7 language models in 10 configurations that have shown strong performance on current leading benchmarks including DeepSWE, Terminal-bench 2.1, and ProgramBench. Our list features models from Anthropic (Opus 5), OpenAI (GPT 5.6 Sol/Terra/Luna, GPT 5.4 mini), Google (Gemini 3.5 Flash Lite), and Moonshot AI (Kimi K3). Unless explicitly specified, we use the default reasoning strength.

Agent system. We use `mini-SWE-agent`, a simple coding agent scaffold that enables models to interact with a terminal environment via bash commands (Yang et al., 2024a). To focus on measuring *model* performance and reduce confounds between model capability and harness design, prior works including DeepSWE, Terminal-bench, ProgramBench, and CodeClash use `mini-SWE-agent` as the primary harness (Huang et al., 2026; Merrill et al., 2026; Yang et al., 2026a;b). A model attempts to repair each repository within a Docker container provisioned with 16 vCPUs and 64 GiB of memory. An inference run is constrained to 1,000 steps and 12 hours of wall-clock time. Appendix A.1 details the inference setting, and Appendix E.1 reproduces the exact prompt.

Metrics. The primary metric is **% bugs resolved**, the percentage of bugs from all repositories that are resolved (i.e., bugs for which all bug-specific hidden tests and all base version tests pass).

4 Results

Table 4 reports the main results; across all evaluated models, SWE-sweep remains extremely challenging. The best model, Sol 5.6 (xhigh), resolves 4.7% of bugs in the dataset for a total of $7k. The second leaderboard position is held by Luna 5.6 (xhigh), which resolved about half as many bugs for considerably less spending (only $224). We also observe a clear progression in scores across the three Luna runs: default 0.5%, high 1.4%, and xhigh 2.5%. Across the three effort levels, we also observe a scaling of turns and output tokens. However, compared with other models, high output tokens or step counts do not directly translate to higher benchmark scores. For example, Opus 5

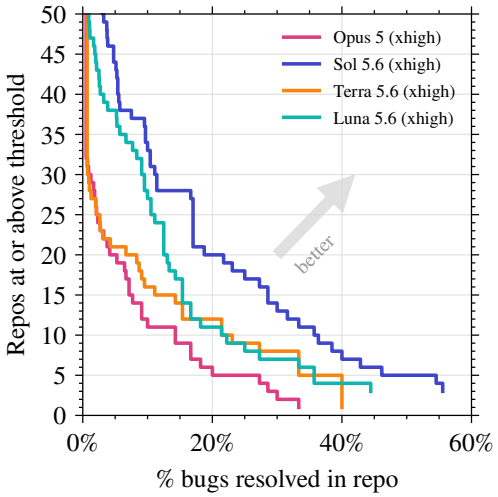


Figure 5: Distribution of repository-level bug-resolution scores. The endpoint of each line indicates the best repository-level score.

Table 5: Percentage of bugs resolved within subsets of repositories, grouped by bug count per repository.

# bugs/repo	# repos	Opus 5 (xhigh)	Sol 5.6 (xhigh)	Luna 5.6 (xhigh)
$<$ 25 bugs	56	4.2%	14.9%	9.3%
25–74 bugs	27	0.8%	3.2%	1.0%
$\geq$ 75 bugs	17	0.6%	2.2%	1.2%

Table 6: Percentage of bugs resolved within subsets of repositories, grouped by the number of lines of code in the repository (excluding test files).

Repo size	# repos	Opus 5 (xhigh)	Sol 5.6 (xhigh)	Luna 5.6 (xhigh)
$<$ 100k LoC	44	3.2%	13.4%	8.2%
100k–500k LoC	37	0.9%	3.3%	1.5%
$>$ 500k LoC	19	0.6%	1.2%	0.5%

(xhigh) has the highest output token count (and second highest step count), but is only ranked in the middle of the leaderboard (1.3%). The Pareto frontier with respect to cost is shown in Figure 4. A similar figure in terms of output tokens is shown in Appendix Figure 14.

Figure 5 shows repository-level scores. Sol 5.6 (xhigh) is consistently strongest across the full distribution, even though its median score is only 3.0% and it scores less than 30% on all but 13 repositories. Its highest score on any repository is 55%. Table 5 shows that the percentage of bugs resolved is lower among repositories with more bugs. Among repositories with fewer than 25 bugs, Sol 5.6 (xhigh) resolves 14.9%, while its resolution rate falls to 2.2% for repositories with at least 75 bugs. Similarly, Table 6 shows that the percentage of bugs resolved is lower among repositories with more lines of code, with the best rates ranging from 13.4% for repositories with fewer than 100k lines of code to only 1.2% for repositories with more than 500k lines of code.

Appendix C provides alternative aggregations and detailed per-language and per-repository results.

4.1 Ablations

To identify the main sources of difficulty in SWE-sweep, we ablate the information provided about each bug and evaluate alternative agent scaffolds.

Table 7 shows how the percentage of bugs resolved changes when we give the agent more information about the bugs or allow for separate work per issue. We use the first 20 repositories of Table 26 (a total of 684 bugs). Appendix E.1 shows the exact prompts.

Identifying the bug is the challenging part. When changing the prompt to include information from the original GitHub issue reports, similar to the setting used by SWE-bench (except that many bugs need to be fixed in a single run), the percentage of bugs resolved increases dramatically, with Sol 5.6 (xhigh) jumping to a 75.3% solve rate. This strongly suggests that identifying and localizing the bug is the most challenging part of the benchmark, while the actual repair is likely comparable in difficulty to SWE-bench-style benchmarks, which are nearing saturation (Anthropic, 2026).

Giving more information about the bug location makes the tasks easier. Since bug identification is the challenging part, we also test whether providing more information about the bug location improves performance. We test two different types of information: In the *filenames* ablation, we provide all filenames changed by the bugs' gold fixes. This doubles performance for both tested models (to 11.0% for Sol and 4.8% for Luna). In the *line ranges* ablation, we also provide the corresponding edited line ranges, leading again to a doubling of performance (19.2% for Sol and 10.7% for Luna).

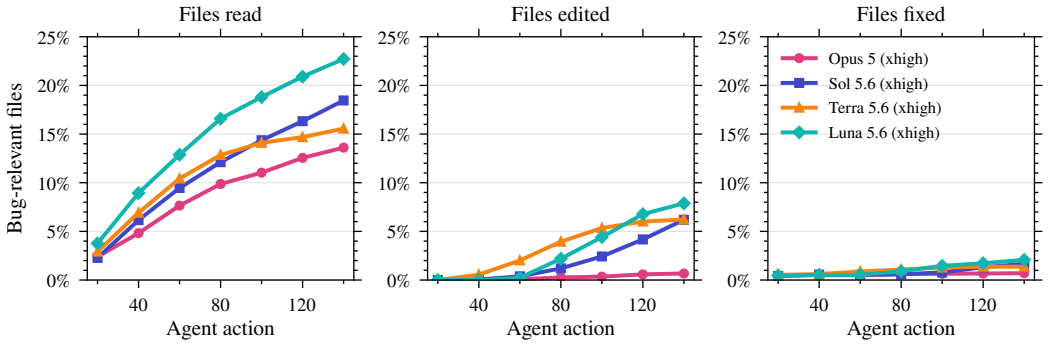


Figure 6: Agents reach far more bug-relevant files than they fully fix. We show every file changed by a gold fix and report whether each model has read or edited it and whether every bug whose gold fix changes that file is resolved. Read means an agent explicitly read the file; edited means an agent edited a file involved in a gold fix; a fixed file requires all accepted bugs related to the file to pass together at a checkpoint with no correctness test regression.

Handling multiple bugs in one trajectory adds to the challenge of the benchmark, particularly for smaller models. In the ablation where line ranges are provided for every bug, we can also isolate the additional difficulty of addressing many bugs within a single trajectory by allowing the model to solve each bug in a separate run. This doubles the performance of Sol (19.2% $\rightarrow$ 39.2%) and nearly triples the performance of Luna (10.7% $\rightarrow$ 31.3%). In the setting where both issue statements and line ranges are provided, the performance of Sol increases slightly, while Luna still almost doubles (43.8% $\rightarrow$ 82.9%). While the default setting of the benchmark does not allow a similar experiment to be conducted, we present test-time scaling results in the next section.

Additional agent scaffolds. We further evaluated additional harnesses on the set of 20 tasks of Table 26 and the 10 repositories with the most bugs. Sol 5.6 (xhigh) scores 1.6% with the native Codex harness, significantly lower than with `mini-SWE-agent` (2.9%). Opus 5 (xhigh) scores 1.1% with the native Claude Code harness, only slightly higher than with `mini-SWE-agent` (1.0%). We therefore conclude that our choice of harness is reasonable for the baseline results of this benchmark.

5 Analysis

We study SWE-sweep trajectories to investigate how different models approach SWE-sweep tasks and how performance scales with more test-time compute. Appendix C.2 provides additional trajectory analysis results.

Agents spend much of their trajectories searching, although models differ in exploration breadth. Reading is the largest observable action category for three of four models (Appendix Figure 16). Opus explicitly reads an average of 159.3 files per repository, more than three times Terra's 48.9, whereas Terra is the only model that writes more often than it reads. Thus, models differ not only in outcomes but also in how broadly they explore before editing.

Agents often read or edit files changed by gold fixes without resolving the corresponding bugs. As shown in Figure 6, by action 140, Sol has read 18.5% of these files and edited 6.2%, but only 1.7% have every corresponding bug resolved. In our evaluations, one or more models edit a file changed by the gold fix for 28.3% of bugs. For bugs with completed annotations, Appendix Table 21 breaks these results down by source of evidence. Reading or editing a file changed by a gold fix does not by itself mean that the corresponding bug is resolved.

Most agent-edited files are not bug-relevant. Across models, only 17.3%–27.9% of files changed in submitted patches are bug-relevant. This measures overlap with known repairs rather than correctness, since a valid repair may change other code (Appendix C.2).

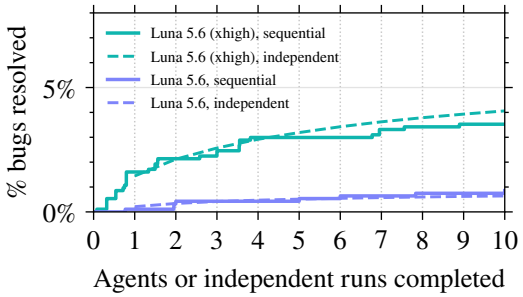


Figure 7: Repeated attempts fix additional bugs, but gains diminish. Results on the 5 repositories with the most bugs. Solid curves show sequential agent runs, each starting from the preceding agent's patch while dashed curves show the union of bugs resolved by independent attempts.

Table 7: Ablation results for the first 20 repositories (in the order of Table 26). Conditions provide fault-localization hints (gold fix filenames or line ranges), issue texts, or both. The settings marked *per bug* run a separate agent for each bug.

Ablation	Sol 5.6 (xhigh)	Luna 5.6 (xhigh)
No information (default)	4.8%	2.0%
Filenames	11.0%	4.8%
Line ranges (LR)	19.2%	10.7%
LR (per bug)	39.2%	31.3%
Issue texts	75.3%	36.8%
Issue texts + LR	78.1%	43.8%
Issue texts + LR (per bug)	87.3%	82.9%

Agents keep revising their patches, but later work does not consistently improve bug recovery. Across runs from four analyzed models, 42 trajectories lose at least one previously resolved bug after a patch change. In total, 28 bug repairs pass at an intermediate checkpoint but not at the end, and 19 runs finish with fewer bugs resolved than at their best intermediate checkpoint (Appendix Table 22). Among runs that resolve at least one bug, models spend a median 24.8%–37.6% of their actions after the final increase in bugs resolved. Later edits can therefore undo successful repairs.

Inference-time scaling through repeated attempts improves bug recovery, but gains diminish. We evaluate two simple inference-time scaling strategies on the 5 repositories with the most bugs: sequential runs, where we run repeated agents, each starting from the preceding agent's patch, and multiple independent agent runs, each starting from the initial task state. In the latter case, we report the union of bugs resolved by all independent attempts. The results are shown in Figure 7. The gains for Luna (default reasoning) are marginal, but for Luna (xhigh), sequential attempts raise bug recovery from 1.6% (first agent) to 3.5% (after 10 agents), while the union of 10 independent attempts reaches 4.1%.

6 Related Work

SWE-bench-style tasks. The SWE-bench benchmark piloted the use of GitHub issue and pull request pairs to construct software-engineering-style tasks in which models are asked to modify the corresponding codebase such that the original issue is resolved (Jimenez et al., 2024). Subsequent work extends the original task format along many dimensions, including more programming languages (Yang et al., 2024b; 2025; Rashid et al., 2025; Zan et al., 2025), increased difficulty (Deng et al., 2025; Desai et al., 2026; Huang et al., 2026; Ma et al., 2026), better alignment between task specification and test coverage (Chowdhury et al., 2024), and solving multiple associated task instances in succession (Deng et al., 2026; Lam et al., 2026). Across the many variations to the original task format, the task design choice to provide models with a clear specification has remained constant. A handful of prior works, notably Ambig-SWE and SWE-together, explored feeding models vague problem statements to study how well models can solicit feedback from a simulated user (Vijayvargiya et al., 2026; Wu et al., 2026). To our knowledge, SWE-sweep is the first to investigate whether models can surface issues in large repositories on their own, with no specification or hints given.

Fault Localization. SWE-sweep's key research questions most closely follow the line of research around fault localization: identifying the code responsible for erroneous behavior (Wong et al., 2016). Classical techniques typically ingest execution evidence, such as failing tests, test-coverage reports, and mutated programs, then rank how suspicious different programmatic entities are (Wong et al., 2016). With LMs, more recent approaches reason directly over the source code (Yang et al., 2023a; Kang et al., 2024; Qin et al., 2025). However, prior work generally still assumes a known bug and provides evidence to guide the system towards it (Liu et al., 2019). Our definition and

operationalization of bug discoverability discussed in Section 2.2 and Appendix B.7 obviate the need for explicit guidance that is typically provided in conventional fault localization evaluations.

Bug Detection. LMs have also been used, without a report, to find bugs or vulnerabilities in fuzzing (Deng et al., 2023; Xia et al., 2024) and cybersecurity benchmarks (Yang et al., 2023b; Zhang et al., 2025b; Wang et al., 2026; Zhang et al., 2025a; Lee et al., 2026). However, these settings target failures that are wrong regardless of context, such as crashes or exploits, whereas most SWE-sweep bugs can only be recognized by inferring the intended behavior from the repository. Concurrent work has begun to place more emphasis on proactive bug discovery but remains narrower in scope than SWE-sweep. In TestExplora, the agent is asked to generate a test that exposes a bug, not a fix (Liu et al., 2026). Active-SWE and TIDE directly fold bug discovery and repair into a single task as well, but are smaller in scale and inform the model which files or functions are defective (Jeong et al., 2026; Li et al., 2026). In contrast, SWE-sweep evaluates open-world, unguided debugging over a whole GitHub repository.

Long-Horizon Code Benchmarks. Recent code evaluations have primarily increased difficulty by challenging models to solve long-horizon tasks that take many turns and multiple hours to solve, such as writing a program from scratch (Yang et al., 2026a), out-coding opponents in competitive programmatic environments (Yang et al., 2026b), and manually curated collections of arduous tasks (Merrill et al., 2026; Zhao et al., 2024; Zhu et al., 2026). SWE-sweep demonstrates the effectiveness of bug discovery and repair as a long-horizon task.

7 Discussion

Limitations and Future Work. As discussed in §2.2, we recognize that SWE-sweep is constructed purely from the bugs that were explicitly identified with an issue and fixed by a corresponding pull request. Consequently, we omit bugs that were reported but were not fixed, or were never reported to begin with. SWE-sweep therefore necessarily under-counts the total bugs present in a codebase. That said, we intentionally keep our focus on explicitly addressed bugs because the upstream fix signals the maintainer's confirmation that a bug is legitimate and provides the tests for evaluating the repair objectively. Extending SWE-sweep to cover unreported bugs would be a natural next step. To tackle this successfully, it would be necessary to validate a proposed defect without a ground-truth fix to check against.

Conclusion. We introduce SWE-sweep to measure models' ability to proactively identify and repair bugs, without explicit specification of any bug's location or errant behavior. Our construction pipeline allows us to automatically synthesize GitHub repositories into tasks with a wide range of bugs. The experiment results suggest that models still struggle to exhaustively identify defects. We observe empirically that bug discovery is more difficult than repair. That said, the gradual increase in bugs resolved across more turns demonstrates promise that more capable models will be more adept at surfacing bugs. We release the SWE-sweep dataset and evaluation harness to advance coding agents towards autonomous repository maintainers.

AI USE STATEMENT

In this work, we used generative AI tools to clean and reformat the dataset, to implement methods, and to support qualitative and thematic data analysis through experimental annotation, while the authors interpreted the results and drew the conclusions. Specifically, the benchmark is built from public GitHub data, and AI tools were used to reformat, adapt, and filter this data; AI tools were also used to write code implementing our methods. We have not used generative AI tools to generate synthetic datasets, to develop theoretical models or conceptual frameworks, to propose or refine hypotheses, to design or provide feedback on research methodology or experiments, or to interpret results. Formulating mathematical claims, providing ingredients for proofs, assisting in the writing of proofs, and translation are not applicable to this work. Additionally, we used generative AI tools to create and edit software code and to edit and polish parts of the manuscript for clarity and readability. The authors have reviewed all AI-assisted work and all AI-generated code was reviewed and tested for correctness by the authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

To the best of our efforts, we have worked to ensure that SWE-sweep complies with the principles outlined in the ICLR Code of Ethics. No private data is included in the benchmark; SWE-sweep is constructed using bugs whose solution was previously published publicly, within publicly-facing repositories. All of our experiments run within Docker containers that do not have internet access. We believe that our work will lead to more resilient and higher-quality software libraries, driven by the autonomous bug finding and fixing systems that SWE-sweep could be used to build.

REPRODUCIBILITY STATEMENT

We open-source the SWE-sweep dataset and evaluation harness[1], along with rollouts for all models reported in the main results (Table 4) to ensure that all metrics and analyses discussed in our work can be recovered from those artifacts. The additional assets we use for carrying out evaluation, particularly `mini-SWE-agent`, are themselves open source. The codebase includes detailed instructions on how to run evaluation.

REFERENCES

Anthropic. Claude Mythos preview system card. https://www.anthropic.com/claude/mythos, April 2026. Accessed: 2026-09-25.

Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement? A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON)*, 2008. doi: 10.1145/1463788.1463819.

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. doi: 10.1109/TDSC.2004.2.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, et al. Introducing SWE-bench Verified. *OpenAI Blog*, 2024.

Cursor Research. Composer 2 technical report, 2026. URL https://arxiv.org/abs/2603.24477.

Gangda Deng, Zhaoling Chen, Zhongming Yu, Haoyang Fan, Yuhong Liu, Yuxin Yang, Dhruv Parikh, Rajgopal Kannan, Le Cong, Mengdi Wang, Qian Zhang, Viktor Prasanna, Xiangru Tang, and Xingyao Wang. Swe-milestone: Evaluating AI agents on continuous software evolution, 2026. URL https://arxiv.org/abs/2603.13428.

[1] https://anonymous.4open.science/r/swe-sweep-iclr-2027-C504/README.md

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.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023. URL https://arxiv.org/abs/2212.14834.

Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, Jiankai Sun, Yiyuan Li, Pramod Srinivasan, Ishan Gupta, Christopher Settles, Daniel Wang, Derek Chen, Pranav Raja, Albert Liu, Marek Šuppa, Nevasini Sasikumar, Luyang Kong, Erik Quintanilla, Xiangyi Li, Ivan Bercovich, and Steven Dillmann. Swe-marathon: Can agents autonomously complete ultra-long-horizon software work?, 2026. URL https://arxiv.org/abs/2606.07682.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. Nl2repo-bench: Towards long-horizon repository generation evaluation of coding agents, 2026. URL https://arxiv.org/abs/2512.12730.

Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pp. 392–401, 2013. doi: 10.1109/ICSE.2013.6606585.

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.

IEEE. IEEE standard glossary of software engineering terminology. IEEE Std 610.12-1990, 1990.

ISO/IEC/IEEE. ISO/IEC/IEEE international standard – systems and software engineering – vocabulary. ISO/IEC/IEEE 24765:2017, 2017.

Soyeong Jeong, Jinheon Baek, Minki Kang, and Sung Ju Hwang. TIDE: Proactive multi-problem discovery via template-guided iteration, 2026. URL https://arxiv.org/abs/2606.04743.

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?, 2024. URL https://arxiv.org/abs/2310.06770.

Sungmin Kang, Gabin An, and Shin Yoo. A quantitative and qualitative evaluation of LLM-Based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1(FSE): 1424–1446, July 2024. ISSN 2994-970X. doi: 10.1145/3660771. URL http://dx.doi.org/10.1145/3660771.

Man Ho Lam, Chaozheng Wang, Hange Liu, Jingyu Xiao, Haau sing Li, Jen tse Huang, Terry Yue Zhuo, and Michael R. Lyu. Swe-chain: Benchmarking coding agents on chained release-level package upgrades, 2026. URL https://arxiv.org/abs/2605.14415.

Hwiwon Lee, Jiawei Liu, Dongjun Kim, Wubing Xia, Ziqi Zhang, Chunqiu Steven Xia, and Lingming Zhang. Sec-bench pro: Can language models solve long-horizon software security tasks?, 2026. URL https://arxiv.org/abs/2605.26548.

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, 2026. URL https://arxiv.org/abs/2608.04682.

Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems, 2019. URL https://arxiv.org/abs/1812.07283.

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, 2026. URL https://arxiv.org/abs/2602.10471.

Jeffrey Jian Ma, Milad Hashemi, Amir Yazdanbakhsh, Kevin Swersky, Ofir Press, Enhui Li, Vijay Janapa Reddi, and Parthasarathy Ranganathan. Swe-fficiency: Can language models optimize real-world repositories on real workloads?, 2026. URL https://arxiv.org/abs/2511.06090.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces, 2026. URL https://arxiv.org/abs/2601.11868.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. Agentfl: Scaling LLM-based fault localization to project-level context, 2025. URL https://arxiv.org/abs/2403.16362.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents, 2025. URL https://arxiv.org/abs/2504.08703.

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR)*, pp. 1–5. ACM, 2005. doi: 10.1145/1082983.1083147.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Ambig-SWE: Interactive agents to overcome underspecificity in software engineering. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=X2yzXtH4wp.

Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. Cybergym: Evaluating AI agents' real-world cybersecurity capabilities at scale, 2026. URL https://arxiv.org/abs/2506.02548.

W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. Swe-together: Evaluating coding agents in interactive user sessions, 2026. URL https://arxiv.org/abs/2606.29957.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Fuzz4all: Universal fuzzing with large language models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pp. 1–13. ACM, April 2024. doi: 10.1145/3597503.3639121. URL http://dx.doi.org/10.1145/3597503.3639121.

Aidan Z. H. Yang, Ruben Martins, Claire Le Goues, and Vincent J. Hellendoorn. Large language models for test-free fault localization, 2023a. URL https://arxiv.org/abs/2310.01726.

John Yang, Akshara Prabhakar, Shunyu Yao, Kexin Pei, and Karthik R Narasimhan. Language agents as hackers: Evaluating cybersecurity skills with capture the flag. In *Multi-Agent Security Workshop@ NeurIPS'23*, 2023b.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024a. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. SWE-bench multimodal: Do AI systems generalize to visual software domains?, 2024b. URL https://arxiv.org/abs/2410.03859.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. Programbench: Can language models rebuild programs from scratch?, 2026a. URL https://arxiv.org/abs/2605.03546.

John Yang, Kilian Lieret, Joyce Yang, Carlos E. Jimenez, Muhtasham Oblokulov, Aryan Siddiqui, Ofir Press, Ludwig Schmidt, and Diyi Yang. CodeClash: Benchmarking goal-oriented software engineering, 2026b. URL https://arxiv.org/abs/2511.00839.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving, 2025. URL https://arxiv.org/abs/2504.02605.

Andy K. Zhang, Joey Ji, Celeste Menders, Riya Dulepet, Thomas Qin, Ron Y. Wang, Junrong Wu, Kyleen Liao, Jiliang Li, Jinghan Hu, Sara Hong, Nardos Demilew, Shivatmica Murgai, Jason Tran, Nishka Kacheria, Ethan Ho, Denis Liu, Lauren McLane, Olivia Bruvik, Dai-Rong Han, Seungwoo Kim, Akhil Vyas, Cuiyuanxiu Chen, Ryan Li, Weiran Xu, Jonathan Z. Ye, Prerit Choudhary, Siddharth M. Bhatia, Vikram Sivashankar, Yuxuan Bao, Dawn Song, Dan Boneh, Daniel E. Ho, and Percy Liang. Bountybench: Dollar impact of AI agent attackers and defenders on real-world cybersecurity systems, 2025a. URL https://arxiv.org/abs/2505.15216.

Andy K. Zhang, Neil Perry, Riya Dulepet, Joey Ji, Celeste Menders, Justin W. Lin, Eliot Jones, Gashon Hussein, Samantha Liu, Donovan Jasper, Pura Peetathawatchai, Ari Glenn, Vikram Sivashankar, Daniel Zamoshchin, Leo Glikbarg, Derek Askaryar, Mike Yang, Teddy Zhang, Rishi Alluri, Nathan Tran, Rinnara Sangpisit, Polycarpos Yiorkadjis, Kenny Osele, Gautham Raghupathi, Dan Boneh, Daniel E. Ho, and Percy Liang. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models, 2025b. URL https://arxiv.org/abs/2408.08926.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch, 2024. URL https://arxiv.org/abs/2412.01769.

Deyao Zhu, Xin Zhou, Shengling Qin, Xuekai Zhu, Hangliang Ding, Shu Zhong, Zixin Wen, Zhonglin Xie, Chenhui Gou, Linxuan Ren, Yueyang Wang, Junfeng Zhong, Rui Liu, Tian Gao,

Yangguang Lin, Jingyuan Zhang, Maojia Song, Xuan Qi, Jinhong Wu, Chenyang Zhang, Yinzhu Piao, Ziru Niu, Hongbin Lin, Lingxiang Meng, Peng Tang, Chengyao Tang, Shanyu Wu, Huanyu Zheng, Yu Liu, Liya Zhu, He Wang, Ming Ding, Ziyu Wan, Hao Liu, Sibo Wang, Haotian Zhu, Xintian Zhang, Nan Chai, Yipeng Liu, Panhao Lai, Sihang Yuan, Zixin Su, Ge Zhang, Wangchunshu Zhou, Yantao Du, Wenhao Huang, and Guang Shi. Edgebench: Unveiling scaling laws of learning from real-world environments, 2026. URL https://arxiv.org/abs/2607.05155.

APPENDIX

A THE BENCHMARK

This appendix supplements the main task description in §2 with inference settings (§A.1), a formal task definition (§B.1), and feasibility studies (§A.2). Appendix B describes the task collection process.

A.1 INFERENCE SETTINGS

This section details the exact setup used for the experiments described in section 3. To ensure a fair evaluation, agents are given enough information to understand what is expected of them and how their solutions are scored, while preventing access to information that would circumvent the intended discovery process.

In addition to the task ("fix all bugs"), the agent receives

1. The codebase at the base commit (which naturally includes the base commit tests).
2. Information about the scoring of its work, in particular the requirement not to break (or change) any base commit tests.
3. The selection criteria defining discoverable bugs, which were used during the construction of the benchmark (see §B.7). Providing these criteria reduces effort spent investigating behaviors that would not qualify as benchmark bugs.

Figure 8 shows excerpts of the prompt to the agent.

Appendix E.1 includes the complete inference prompt and all ablation-specific changes.

Task scope

```
...Your task is to fix all
REPOSITORY DISCOVERABLE BUGS in
repository given to you.
You MUST NOT make any OUT OF
SCOPE edits....

DEFINITIONS

(...)
```

Regression constraint

```
...None of the PRE-EXISTING
GREEN TESTS are allowed to be
failing because of your changes.
Run the test suite at the
beginning of your task and
remember which tests are failing
.
These are the only (!) PRE-
EXISTING TESTS hhat are allowed
to fail.
Breaking any (!) other PRE-
EXISTING TEST will result in
your score being set to zero....
```

Figure 8: Excerpts from the inference prompt.

Precautions against reward-hacking behavior. Agents do not have access to the internet during evaluation. Web access could reveal the original issue reports, pull requests, or later commits containing the reference fixes. Furthermore, the environment does not provide the `git` history of the repository (in particular, this prevents the agent from discovering commits after the base commit, which would reveal all fixes).

A.2 FEASIBILITY

In our experiments, the best agent (Sol 5.6 (xhigh)) resolves only 4.7% of bugs. This raises the question of whether a perfect agent could resolve all bugs, or whether some bugs cannot be resolved at all. To resolve all bugs, an agent must find each bug, write a fix that passes its hidden tests, keep all fixes compatible, and finish within its budget. Below, we address the possible concern for each of these steps.

Possible concern: The agent cannot distinguish between expected behavior and a bug (i.e., the bug is not discoverable). Agents do not see issue texts, so the expected behavior must be apparent from the repository. We detail our more precise definition of discoverable bugs that is built around *repository contracts* (i.e., expected behavior inferable from content of the repository) in §B.7. All bug candidates are checked for these discoverability criteria by 3 independent LMs during construction. We only keep bugs where all three LMs agree that the bug is discoverable.

We recognize that this is the concern that is most specific to the SWE-sweep setting. We therefore validate this part of the filtering process with two studies:

First, in a human annotation study (§A.4.1), we ask four human annotators to validate the discoverability of the bugs that pass all LM-based filters. Before seeing the LM judges' reasoning, the reviewers accept that 49 of 50 bugs are discoverable by at least a 3/4 majority, including 42 unanimously. After seeing the reasoning, several reviewers changed their verdicts and also the last bug was considered to be discoverable.

Second, we run a validation study where the test patches of all bugs that pass the construction filters are given to an agent. The agent is tasked to find evidence within the repository that supports the expected behaviors that the tests assert. The agent does not see issue texts, gold fixes, or similar information. In this study, the agent finds a repository-defined contract for 99.95% of bugs (§A.4.2). The most common evidence is a related, pre-existing test (88.1%), followed by internal invariants (82.4%) and docstrings (57.9%). The 2 bug instances where the agent does not find a direct repository-defined contract are explicitly shown in §B.7. Both bugs are still deemed discoverable because they concern race conditions and memory-safety issues, which are both considered to be inherently undesirable behavior.

Finally, the results show large headroom. Scores differ by more than $10\times$ between the evaluated models (Table 4). In the ablations, localization hints alone raise performance by about $8\times$ for Sol 5.6 (xhigh) and $16\times$ for Luna 5.6 (xhigh) over the default setting, when the agent gets the line ranges of each bug and one run per bug (Table 7). These hints do not reveal the expected behavior (or any new information that is not apparent from the repository); they only influence where agents spend their time. Likewise, separate runs per bug add no information and only spare the agent from tracking many bugs in one long context. Better discovery alone could therefore raise scores several-fold, which leaves ample room for agents to improve on SWE-sweep.

Possible concern: Some bugs may not exist at the base commit, or may have no working fix. Every bug passes executable checks at the base commit. At least one hidden test fails without the gold fix, and all hidden tests pass with it. We run the hidden tests 10 times with the gold fix applied and keep only bugs whose tests pass every time. To make sure that the regression tests are not flaky, we similarly rerun all base commit tests 10 times and keep only tests that pass every time. So every bug has a reliable test and a known fix, taken from the upstream pull request.

Possible concern: A correct fix may still fail the hidden tests. This concern is shared with most other coding benchmarks that use unit tests for validation, including SWE-bench, Terminal-Bench, and ProgramBench (Jimenez et al., 2024; Merrill et al., 2026; Yang et al., 2026a). Hidden tests can be too strict, for example if they check the exact wording of an error message that the gold fix introduced. We reject such bugs with the LM annotations detailed in §B.8. Three LMs have to agree that the tests are appropriate for a bug to be part of the benchmark. We further validate the appropriate-tests judgments in the human study described in §A.4.1. Before seeing the LM judges' reasoning, a 3-of-4 majority of reviewers accepts all sampled bugs under this criterion.

Furthermore, when Sol 5.6 (xhigh) receives the issue texts for all bugs in each repository, it resolves 75.3% of bugs across the runs in Table 7, increasing to 87.3% if it also receives the line ranges of each bug and runs separately for each bug. This puts a tight bound on the number of tests that might suffer from the above concern (note that this ablation merely frames the bug in a SWE-bench-style setting that removes the difficulty of discovering the bug; solving the bug might still be challenging for an LM, which is why a performance of 100% is not necessarily expected).

We also conduct a stricter source-blind validation in which an LM judge sees only the base repository and test patch and checks whether the repository supports every material test expectation. The judge finds direct support for every expectation in 87.6% of cases (§A.4.3), which also puts a bound on the number of tests that might suffer from this concern.

Possible concern: Fixing one bug may break another bug or a base commit test. If two bugs required opposite behavior, no agent could resolve both. We rule out conflicts with the base commit tests by execution: we apply each gold fix alone, run all base commit tests, and remove every bug whose gold fix breaks one. We also remove base commit tests that do not pass in all of 10 runs, so a flaky test cannot count as a regression. For conflicts between bugs, an LM judge reviews all test patches of a repository together. It reports no conflicting pair in any of the 100 repositories. The issue-text ablation also sets a clear upper bound on any such problem: across the tested repositories, Sol 5.6 (xhigh) fixes 75.3% of bugs at the same time.

Possible concern: The budget is too small to fix every bug. Agents stop long before they reach their limits (Table 24). The longest run uses 648 of 1000 steps and 5.5 of 12 hours. We do not limit cost or tokens per run and do not observe any failures due to models exceeding their context window. Every run ends because the agent submits (i.e., declares the task to be done), and no run states that it ran out of budget. More steps are also not always helpful, since agents spend a median of 24.8%–37.6% of their actions after their last gain in bugs resolved (§5). Appendix C.2 discusses stopping behavior and the full trajectory analysis in detail.

A.3 Other concerns

Possible concern: The selection of benchmark bugs might systematically disadvantage certain models. Two models may fix a similar number of genuine bugs overall, while differing in the kinds of bugs they tend to identify. If one model's bug distribution happens to more closely match the bugs represented in SWE-sweep, it could receive a substantially higher benchmark score despite comparable overall performance. However, agents are given the same bug-discoverability criteria that we use when selecting bugs for SWE-sweep, reducing potential mismatch between the benchmark definition and the task presented to the agent. In addition, we evaluate this concern directly using a substantially looser definition of a bug in §A.4.4. For each repository, an LM judge counts the total number of bugs that each model attempts to fix, including bugs outside the SWE-sweep reference set. The attempted-fix count gives the same model ordering as the benchmark score, which is consistent with benchmark performance reflecting models' broader tendency to identify and fix bugs.

Possible concern: The benchmark might favor models that simply attempt more changes, rather than identifying the most relevant bugs. A model that modifies more files or attempts more fixes may have more opportunities to match one of the benchmark bugs, even if much of its additional work is unnecessary. However, our strict bug-selection process and requirement for maintainer-confirmed ground truth already limit this effect: models receive credit only for fixing known bugs that were validated by the PR merge by the repository maintainers. In addition, models are limited by the edits they can make by the base commit tests that check for any regression. Repositories in SWE-sweep generally have high test coverage, which should severely limit the impact of any possible introduced issues.

We nevertheless observe (§A.4.4) that the model with more attempted fixes also has the higher benchmark score. Penalizing models for additional work, however, would introduce a more fundamental problem. We cannot reliably determine whether work outside the reference set is unnecessary: it may address code quality improvements, or address genuine bugs that are absent from our ground truth, including previously unknown defects. We therefore avoid penalizing additional fixes and restrict the benchmark score to the set of bugs described. We also believe that models which surface more bug candidates can still be useful systems for human users: even when some candidates are incorrect or lower priority, they can be ranked or filtered downstream, whereas models that fail to identify potential bugs in the first place provide no such opportunity.

A.4 Validation studies

The construction filters (§B.6) rely on LM judges that see the issue text and the gold fix. This leaves two questions. First, do the LM judges apply our criteria as a human would? Second, is the expected behavior still clear with only the information an agent has? We address the first question with a human annotation study on 50 randomly sampled bugs (§A.4.1). We address the second with two blind LM studies in which an agent sees only the base repository and the test patch. The first finds the repository evidence for each bug's main failure (§A.4.2). The second checks that the repository supports every material test expectation (§A.4.3). Finally, we present a side study that compares benchmark scores with the total number of potential bugs that LMs attempt to fix (§A.4.4).

A.4.1 Human annotation validation of rubrics used in construction

We use a human study to check the four LM-based quality rubrics.

Four human reviewers judge 50 of the included bug instances. We draw a seeded uniform random sample of 50 bugs from the 4,068 bugs. Every reviewer judges the same fixed sample. Four authors review each sampled bug separately.

Reviewers are presented with an annotation interface that contains the raw GitHub issue, GitHub pull request, gold fix, and test patch. The interface first shows an LM-written summary of these artifacts to aid understanding. This summary was produced by GPT-5.6 Terra (high), and was neither instructed about the rubrics nor saw the LM judges' verdicts.

Reviewers can mark a bug as accept, reject, or unsure based on the following criteria:

1. Is this a genuine bug whose expected behavior can be found from the repository without the issue?
2. Do the tests represent one defect, or several symptoms of one shared root cause?
3. Would any reasonable fix for the bug pass the added tests?
4. Does the test patch preserve all expectations already recorded in the test suite?

These criteria correspond to the LM annotations that are used in benchmark construction (§B.1).

To reject a bug, a reviewer must name at least one failed criterion and explain the rejection in writing. A reviewer who cannot decide may instead mark the bug unsure, optionally naming the criteria in doubt. Reviewers cannot skip a bug. If the reviewer accepts the bug, the verdict is saved and the interface moves to the next bug.

If the reviewer rejects the bug or selects unsure, the first verdict is locked. The interface then shows a summary of the LM judges' decision for each criterion, followed by the full verdicts and reasoning from the individual models. These reasoning traces often contained mentions of specific domain knowledge and considerations which were not previously directly available to the reviewers as part of the issue or PR texts or summaries thereof. Reviewers were then asked to cast a second vote (*final vote*), either confirming or changing their judgment. A written explanation was required for a change of judgment.

The interface is shown in Figures 9 and 10.

Human reviewers accept 94% of bugs by at least a 3/4 majority before seeing the LM judges' reasoning. Reviewers spent between 0.9 and 2.5 hours on the review task. Table 8 summarizes the verdicts. In their first votes (before seeing the reasoning of the LM judges), 47 of the 50 bugs were accepted by a 3/4 majority (counting any unsure votes as rejects). Reviewers therefore strongly agreed with the LM judges that 94% of the reviewed bugs belong in the benchmark. Of the 200 votes that were cast, 178 were accept verdicts, 11 were reject verdicts, and 11 were unsure verdicts.

After seeing the LM judges' reasoning, reviewers accept 49 of 50 bugs by a 3/4 majority; no consensus is reached on the final bug. For the reject and unsure votes, the reviewers were then shown the LM judges' reasoning and given the option to change their opinions. 12 changed to accept, while 5 rejects and 5 unsure verdicts were kept. With the final verdicts, 49 of the 50 bugs (98%) were accepted by at least a 3/4 majority. The remaining bug received two accept votes, one unsure vote, and one reject. The latter votes were cast by the two most critical judges, who did not agree on why the bug might be considered problematic (discoverability vs overly specific tests).

Split by individual rubrics, agreement is even higher. We further break down the final verdicts by rubric in Table 8. The discoverability rubric is the only rubric with a single bug that was not initially accepted by at least a 3/4 majority; however, it was accepted unanimously after seeing the LM judges' reasoning. The *human validation* paragraphs in Appendix B discuss agreement on each rubric with examples.

A.4.2 Using LM agents to collect evidence for expected behavior

During construction, a panel of 3 LM judges checks a bug is discoverable by reading the issue, test patch and gold fix (§B.7). Each judge is a single LM call, and all 3 judges must unanimously agree that a bug is discoverable using the given info.

In this validation study, we collect stronger evidence for this criterion. We give each bug to an LM agent instead of a single LM call, so it can search and read the repository over many steps. Rather

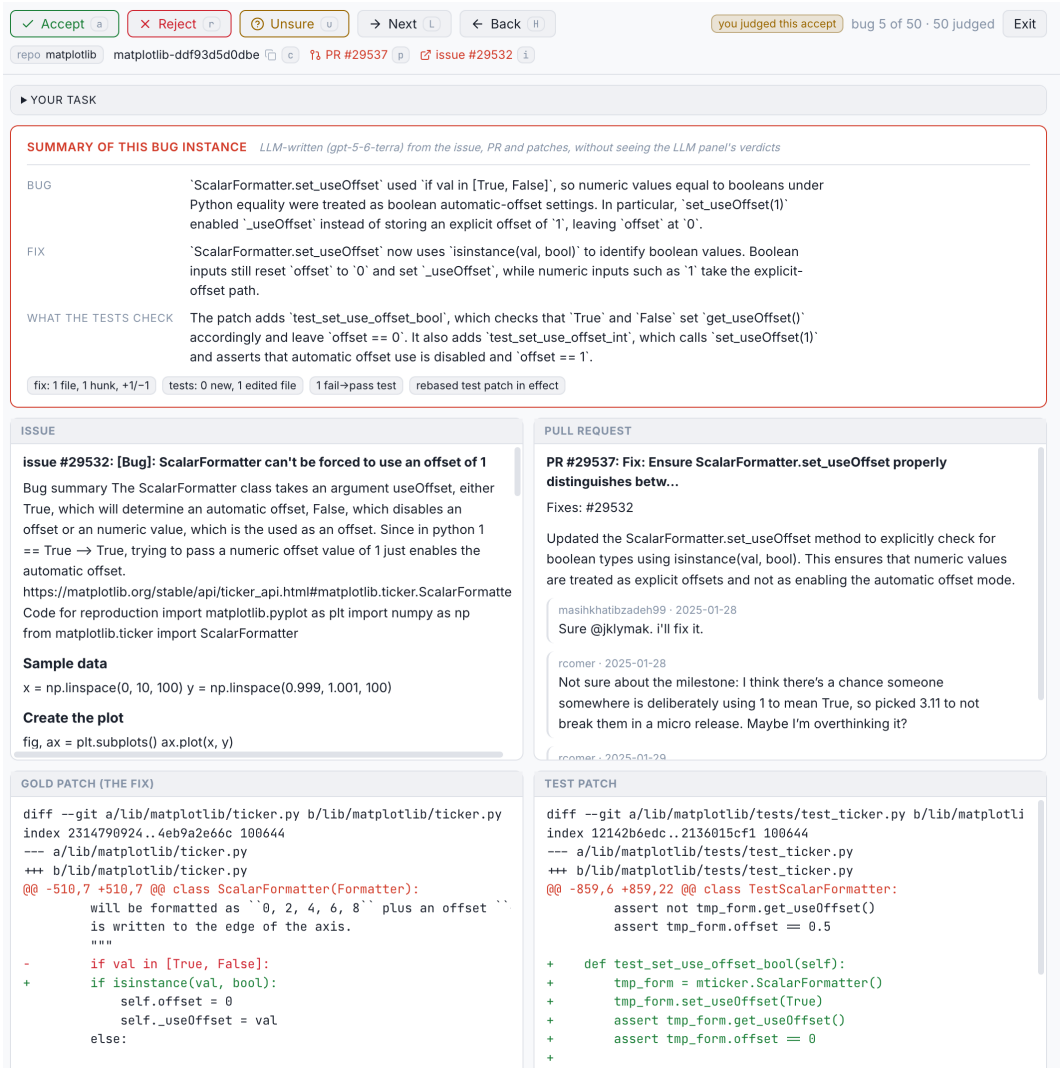


Figure 9: Human-annotation interface. In this example of a bug, the `ScalarFormatter` in `matplotlib` cannot be set to have an offset of 1. This is due to a bug in handling boolean values (in Python `1==True`, so `1 in [True, False]` evaluates as true, causing the integer 1 to be handled as the boolean `True`).

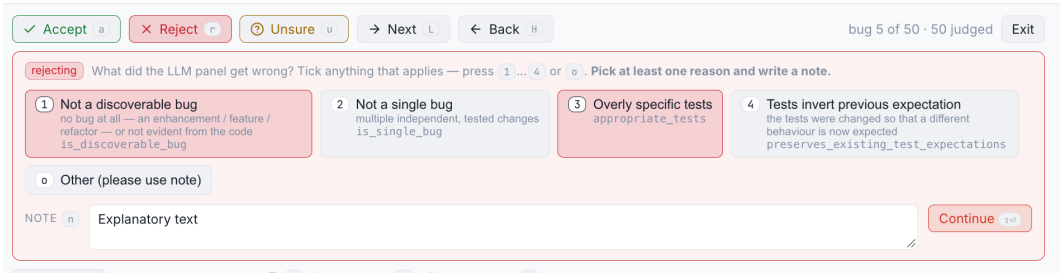


Figure 10: Rejection step in the rubric-study interface. After selecting reject, the reviewer names the failed criteria and writes an explanation.

Table 8: **Human verdicts on the 50 sampled bugs with non-accept votes split by rubric.** Reviewers cast a single accept/reject/unsure vote on each bug and name any rubric(s) they reject or are unsure about. For this table, unsure votes count as rejects for each rubric they name. *First* is the verdict before seeing the LM judges' reasoning; *Final* is the verdict after, which only reject and unsure verdicts reach.

	Majority accept		Majority reject		Split judges	
	First	Final	First	Final	First	Final
Discoverable bug	98%	100%	0%	0%	2%	0%
Single bug	100%	100%	0%	0%	0%	0%
Appropriate tests	100%	100%	0%	0%	0%	0%
Preserves expectations	100%	100%	0%	0%	0%	0%

than just asking for a boolean result, the agent must find and cite the repository contract (e.g., a piece of documentation) that shows the expected behavior that the actual behavior contradicts.

The agent sees only the base repository and the hidden tests. It does not see the issue, the gold fix, later versions of the code, or the construction annotations. In particular, it is not told which source lines the fix changes. These lines are often close to the repository contract, so the agent must find the contract on its own.

We run mini-SWE-agent with GPT 5.6 Sol at high reasoning effort in a read-only, network-disabled copy of the repository at the base commit without git history. We run this study on all 4,068 bugs.

The annotation returned a valid label for 98.9% of selected bugs. The remaining 1.1% of reviews exceeded run limits or produced invalid submissions and are excluded from these percentages. We group each completed annotation into one of two broad categories. Of these annotations, 4,023, or 99.95%, cite a repository-defined contract. Only 2 bugs, or 0.05%, relied solely on a repository-independent failure such as a crash, hang, or unintended data loss. These cases are discussed in detail in §B.7 and are found to be reasonable inclusions, as they involve a race condition and an out-of-bounds read.

Table 9 further separates the repository-defined contracts into detailed sources, which may overlap for the same bug. Appendix E.3.1 gives the annotation prompt.

Table 9: **Defect types and evidence for expected behavior.** Defect and headline evidence shares use completed post-filtering annotations. Detailed source shares use bugs with repository-defined support. Each bug has one primary observable failure and one broad evidence category. Detailed sources may overlap.

Observable failure		
Category	**Bugs**	**Share**
Incorrect output	1,662	41.3%
Runtime or resource failure	899	22.3%
Validation or diagnostic failure	1,129	28.0%
Incorrect state or side effect	209	5.2%
Missing or spurious behavior	126	3.1%
Other	0	0.0%

Evidence basis		
Category	**Bugs**	**Share**
Repository-defined contract	4,023	99.95%
Repository-independent failure only	2	0.05%
Detailed repository evidence source (may overlap)		
Documentation	1,918	47.7%
Docstring	2,329	57.9%
Type annotation	309	7.7%
Pre-existing test	3,544	88.1%
Caller or example	334	8.3%
Identifier or assertion	1,144	28.4%
Internal invariant	3,315	82.4%
Sibling implementation	1,581	39.3%
Referenced standard	447	11.1%
Mathematical or scientific definition	383	9.5%

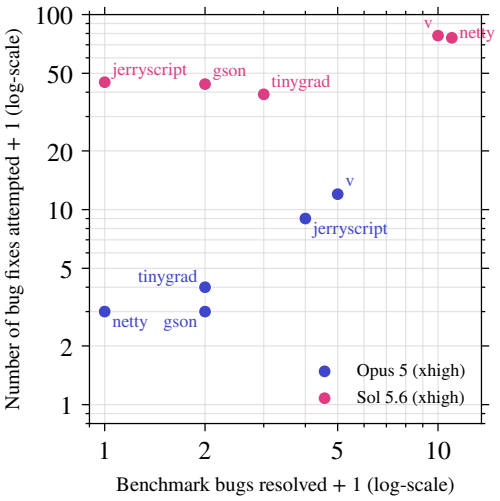


Figure 11: Benchmark bugs resolved and bug fixes attempted for several repositories.

Table 10: **SWE-sweep score vs bugs attempted, summed over all repositories.** *Bugs attempted* is the number of distinct bugs that an LM judge finds in the final patch, including bugs that are not part of SWE-sweep. Each of two judges (Opus 5 and Sol 5.6 (high)) is run three times; we report the range [min, max] over the six runs.

Model	Score	Bugs attempted
Sol 5.6 (xhigh)	4.7%	[3843, 4427]
Opus 5 (xhigh)	1.3%	[387, 393]

A.4.3 Using LM agents to ground all test expectations

This section conducts a validation study similar to that in the previous section (§A.4.2), but goes even further. Rather than only checking for evidence of the expected behavior that conflicts with the bug, the agent is tasked to check for evidence of every single assertion that is made by the tests of a bug instance. This addresses concerns not only about discoverability, but also about whether the hidden tests could be overly specific.

The construction-stage annotation (§B.8) sees the issue, test patch, and gold fix and asks whether any reasonable fix for the bug would pass the tests. Seeing the gold fix and original issue description could have introduced a bias. We therefore apply a stricter post-filtering validation study using the same agent setup from §A.4.2: The judging agent sees only the base repository and test patch (as opposed to seeing details about the issue or gold fix, as in the construction-stage annotation). The agent must decide whether every material test expectation is supported by the repository or by a repository-independent failure such as a crash or data loss (see our definitions of discoverability in §B.7).

Among bugs with unanimous acceptance from the original discoverability annotation, the stricter annotation flagged only about 12.4% as potentially requiring more evidence. However, this validation task is itself challenging for LM agents, and some flagged cases may therefore reflect limitations of the judge rather than genuine problems with the tests. We consequently interpret these results only as an upper bound on the share of potentially problematic test cases. The exact prompts used are shown in Appendix E.3.2.

A.4.4 Benchmark scores vs. number of bugs attempted

In this section, we provide data that helps address possible concerns mentioned in §A.3. Both concerns are related to whether the benchmark score reflects how many bugs a model attempts to fix in general, or only how well its fixes match the distribution of bugs in SWE-sweep. To answer this, we count all bugs that a model attempts to fix (based on rather loose filtering with LM judges), including bugs that are not part of SWE-sweep, and compare this count to the benchmark score.

Setup. We take the final patches of Opus 5 (xhigh) and Sol 5.6 (xhigh) on all 100 repositories. For each patch, an LM judge lists every distinct bug that the patch attempts to fix. This is an inherently noisy decision, as many edits can be related. We adopt a similar prompt to that given for the construction-stage annotation on *single unit of work* (§B.9; full prompt in §E.2.2), which focuses on the number of distinct root causes. The judge only sees the changes that the agent made to source files and the lines around them. We further instruct the LM to filter based on the definition of *bugs*

from the inference prompt (§E.1). In particular, changes that do not fix a bug are not counted, for example refactors, new features, formatting, or checks for conditions that cannot occur. However, the judge does not decide whether a fix is correct or complete. In contrast to the bugs in SWE-sweep, a counted bug does not need an upstream fix or a hidden test, and it is not checked by execution. We use two LMs as judges, Opus 5 (high) and Sol 5.6 (high), each run three times, and report the results of both.

The benchmark score and attempted-fix count give the same model ordering. Table 10 shows each model's SWE-sweep score next to the number of bugs that it attempts to fix, summed over all repositories. For bugs attempted, it reports the range over six runs of the two judges. The SWE-sweep score and count of attempted fixes give the same order of models for both judges. Sol 5.6 (xhigh) attempts around 4k bug fixes according to the LM judges, a similar order of magnitude to the total number of bugs tracked in SWE-sweep, while Opus 5 attempts significantly fewer fixes in the first place. Figure 11 further visualizes this relationship per repository five repositories.

B TASK COLLECTION

In this section, we describe benchmark construction in detail, expanding on §2.2.

B.1 HOW ARE TASKS DEFINED?

Before describing the construction pipeline, we define more formally what tasks contain and the requirements that make them valid.

The central unit of SWE-sweep is a *task*. A task represents a repository at a fixed point in its history, called the *base commit*. A SWE-sweep task consists of a repository at a particular commit, which we call the *base commit*, and a collection of *bug instances*.

What conditions must a bug instance fulfill? Every bug instance is formalized as a test patch and a *gold fix*. The tests introduced or changed by the test patch are referred to as the *hidden tests* of the bug instance. To be a valid bug instance for the purposes of SWE-sweep, it must fulfill the following criteria:

- R1 The test patch and gold fix independently apply to the base commit.
- R2 All hidden tests pass after the gold fix has been applied to the base commit, but at least one hidden test fails on the base commit (*F2P tests*). There might be additional tests that pass before and after the gold fix has been applied (*P2P tests*).
- R3 The F2P tests reveal a single *discoverable bug* in the base commit (see §B.7 for more information on how we define discoverable bugs).
- R4 The target tests are not *overly specific*; any reasonable fix to the discoverable bug will pass the target tests.
- R5 The target tests do not contradict the base commit tests.
- R6 Target tests of different bug instances do not contradict each other.

B.2 REPOSITORY SELECTION

We build SWE-sweep tasks from permissively licensed open-source GitHub repositories. We filter for repositories that link each bug fix to a bug report, e.g., projects that merge pull requests closing tracker issues, or that name tickets in their commit messages. Repositories must have sufficiently good build systems that we can build and test them at a revision years in the past. Building and executing tests must work on a 64-bit Linux system and complete in reasonable time (preferably less than 1 hour). The number of GitHub stars together with the number of contributors and the age of the repository is a good heuristic to select projects that are likely to have a good test suite and strong standards for issues and pull requests. A strong heuristic for bug instance yield is the number of issues in the project's tracker.

B.3 CONSTRUCTING BUG INSTANCE CANDIDATES

This stage turns a repository's history into a population of bug instance candidates.

We start with a large pool of merged pull requests that have an associated issue. Occasionally, to handle repositories that do not use pull requests as a main editing mechanism or are public mirrors of a private repository, we mine candidates from commit messages paired with reports from their custom issue trackers instead. Each bug instance begins with a commit or pull request linked to a bug report. We divide the associated changes into two artifacts: a gold fix containing changes outside the test suite, and a test patch containing changes to the repository's test suite.

In order to do so, we build two sets of regular expressions for path names: one for the graded source code and one for the test suite and the in-tree test infrastructure the tests import. Anything matching neither of these (e.g., documentation or CI configuration) is dropped. The glob sets must also cover build-system files for new tests; otherwise, a test patch may add a test that is never executed.

A bug instance candidate therefore is specified by the following three items:

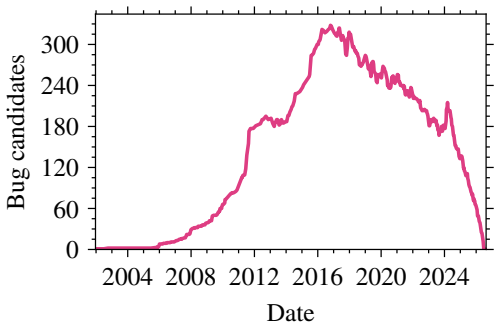


Figure 12: Number of concurrent bug instance candidates across the history of NumPy.

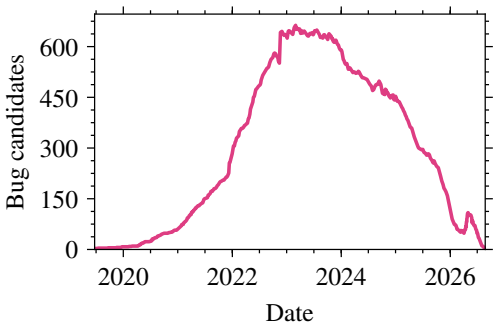


Figure 13: Number of concurrent bug instance candidates across the history of V.

- A *test patch*, which changes the repository's test suite. All tests that are changed are referred to as the *hidden tests* of the bug instance candidate.
- A *gold fix*, which changes code outside the test suite.
- Additional metadata: Description of the bug that the bug instance candidate fixes. This can be taken from the PR description or comments, from the issue or comments of any linked issue, or information directly from commit messages. We also collect additional metadata such as GitHub tags or labels. This additional information is not used for the evaluation or the main inference setting. However, it is used for some of the LM annotations, as well as some of the ablations.

We then apply a series of pre-filters to narrow down the number of bug instance candidates.

1. **Patches do not exceed size limits.** The gold fix and test patch together must touch fewer than 100 files and change fewer than 10.000 lines.
2. **Patches are not duplicates.** Fixes might be cherry-picked onto several branches. To exclude duplicates, we therefore remove candidates with an identical gold fix or an identical test patch.
3. **Adds tests.** The test patch must be non-empty.
4. **Has a source fix.** The gold fix must be non-empty.
5. **Is a bug fix.** We only consider changes that can reasonably be considered a bug fix. For this, we use Gemini 3.5 Flash Lite as a judge, chosen for throughput and price point due to the large number of candidates that need to be processed. The LM sees the PR/issue titles, labels, descriptions, and comments, and decides whether the change is a bug fix at all. This pre-filter maximizes recall measured against the stricter downstream filters while significantly reducing the number of candidates that need to pass through the more expensive later stages. In a limited study of repositories with well-maintained labeling of PR types, the results of this stage are largely identical to labeling based on said labels.

B.4 Determining the base commit for the repository

The goal of this stage is to find a repository commit containing many bug instance candidates at the same time. For each bug instance candidate, we estimate the period during which the bug was present as $[\text{introduction time},\ \text{fix time})$. We then count, for every commit, how many such intervals overlap it.

The fix time is given by the fix's mainline commit. When a fix appears only as a backport, we map it to the corresponding mainline change; if no mainline placement can be established, we discard the candidate.

To estimate when a bug was introduced, we apply the SZZ algorithm (Śliwerski et al., 2005). We blame the source lines that the fix removes or modifies after excluding tests, documentation, benchmarks, and examples, and select the latest commit identified by blame. When a fix consists only

of added lines, which leaves no removed lines to blame, we apply blame to the surrounding context. We locate interval boundaries using committer dates rather than positions along the first-parent history, so that introducing commits made on pull-request branches can still be represented.

The resulting curves (e.g., Figures 12 and 13) are only estimates. Around the peak of this curve, we select up to 10 base-commit candidates to test, focusing on release tags as they are typically more stable to build and spaced around larger changes.

For each shortlisted base commit candidate, we then build a Docker image and ensure that tests can be executed inside the image. This requires writing test runner/adapter code for each test framework. Base commit candidates that cannot be built or have tests that cannot be executed are excluded.

We then count the number of bug instances whose patches apply (R1) and whose bugs can be reproduced (R2). This leads to a more precise estimate of the number of concurrent bug instance candidates than the initial estimate based on the SZZ algorithm.

Based on these counts, we then select the final base commit.

B.5 Making more patches apply

Patch application is the single largest source of attrition in construction: at a base commit years older than most fixes, the great majority of candidate patches no longer apply. We therefore task `mini-SWE-agent` (Yang et al., 2024a) with rebasing them. The agent starts in the base-commit image with the repository checked out at the base commit, and receives the two original patches and the issue description. It writes a rebased gold fix and a rebased test patch. Repaired candidates are checked again for applying patches (R1) and reproduction (R2).

Two examples of rebased patches are shown below:

Gold fix (original)

```
   if (this.match(tt.name)) {
-    return this.state.value !== "async" && this
      .state.value !== "let";
+    const value = this.state.value;
+    if (value === "async" || value === "let") {
```

Original: babel PR:#11417. The upstream fix replaces a condition that also tests for `let`.

Gold fix (rebased)

```
   if (this.match(tt.name)) {
-    return this.state.value !== "async";
+    const value = this.state.value;
+    if (value === "async" || value === "let") {
```

Rebased: babel PR:#11417. The `let` check does not exist at the base commit, so only the replaced line differs. The inserted code is identical.

Test patch (original)

```
-       "interactive037", "interactive038"]
+       "interactive037", "interactive038", "
     interactive039"]
```

Original: Idris2 PR:#2010. The upstream patch registers a new test at the end of the list.

Test patch (rebased)

```
-       "interactive037"]
+       "interactive037", "interactive039"]
```

Rebased: Idris2 PR:#2010. The same test is registered. The base commit's list is shorter because `interactive038` does not exist yet.

For most rebasing attempts (59.2%), Claude Opus 4.8 was used, while Gemini 3.5 Flash Lite and GPT-5.6 Luna were substituted for later runs after they showed similar performance at lower cost. The median agent run takes 15 steps. The agent self-reports failure on 39.3% of attempts (indicating that the codebase has shifted in such a way that the patches cannot easily be adapted while being functionally equivalent).

Of the 24,268 bug instance candidates we attempt to repair, 59.1% yield a patch that applies, and 7.4% pass the remaining criteria and enter the benchmark. Out of these, 22.7% have both patches repaired, while 42.7% have only the test patch repaired.

Most rebased patches only differ in hunk position. We check whether the rebased patches are semantically identical to the originals in two ways. First, in 60.4% of rebased patches the added and removed lines are identical to the original and only the hunk position moved. Second, across the 1,806 bug instances in the benchmark that depend on a rebased patch, we ask three LMs to compare each rebased patch against its original and judge whether the same defect is repaired and still exercised by the test, using the panel and the unanimous-acceptance rule of the annotation stages

Table 11: **Rebasing a patch does not change how hard its bug instance is.** Share of bug instances that GPT-5.6 Sol resolved, split by whether or not the test patch or gold fix was rebased to make it apply to the base commit.

	Gold fix rebased	Not rebased	Any gold fix
Test patch rebased	5.0%	5.8%	5.5%
Test patch not rebased	4.0%	4.3%	4.3%
Any test patch	4.4%	4.7%	4.6%
Any patch rebased			5.0%
Neither patch rebased			4.3%

described below. Only 2.3% of patches do not pass this test. The corresponding bug instances were excluded from the benchmark.

Human validation. The authors manually reviewed 50 of the accepted rebased patches. This resulted in no further rejections of any bug candidates. Corroborating the above finding, most reviewed patches only differed in the hunk positions of the changes, while most of the remaining patches were adapted because of surrounding cosmetic changes or changed function signatures. Only four out of the 50 patches were materially changed. Three of these simply removed unrelated changes that were present in the original source patch but were irrelevant to the bug at hand. The one remaining materially changed patch adapted to a refactor of surrounding logic but applied the identical fix.

Resolve rates are not significantly different between rebased and non-rebased patches. Repair could also leave the bug intact but make it easier or harder to find and fix. To test this, we compare agent success on rebased and non-rebased bug instances (Table 11): 5.0% against 4.3%, a difference of +0.6 pp. This difference is not statistically significant, as the 95% confidence interval overlaps with zero ([-0.8, +2.2] pp). The confidence interval was computed using cluster bootstrapping over repositories. We therefore find no evidence that rebasing changes how hard a bug instance is.

B.6 Quality control

At this stage of the construction pipeline, we now have a collection of bug instance candidates that apply to the base commit (R1) and can be reproduced (R2).

To obtain a reliable final set of bug instances, we next filter out flaky tests and candidates whose fixes break behavior covered by the base test suite.

After the base commit has been chosen, we first run all base commit tests 10 times. We exclude all base commit tests that do not pass every single time (for simplicity, we use *base commit tests* to refer only to these passing tests below). We then ensure that R1 and R2 are *reliably* satisfied for all bug instance candidates by executing the hidden tests. For every bug instance candidate, we execute all hidden tests 10 times and exclude all bug instance candidates that have hidden tests that do not pass every single time with the gold fix applied. We further ensure that R5 is satisfied as follows: For every bug instance candidate, we apply its gold fix, then run all base commit tests. Any bug instance candidate that breaks a base commit test is excluded.

The remaining criteria (R3, R4, and R6) are checked using LM annotations as discussed in the following sections. For each of the annotation stages, we show real candidates drawn from the construction pipeline, paired with the verdict the annotation panel returned for them. Three models vote on each stage and a candidate advances only on unanimous acceptance, so a single rejection is decisive.

B.7 When is a bug discoverable?

Because agents receive no issue description, the expected behavior for each bug must be recoverable from the base repository or from a failure that is unambiguously wrong. We use the following two sources of evidence to make this distinction explicit. Table 12 summarizes acceptance and agreement for this review. Appendix E.2.3 gives the full prompt.

Repository-defined contracts. The repository's own user-visible statements establish expected behavior. These include documentation and repository content used to produce documentation, such as docstrings and type annotations. A repository can also incorporate an external source by reference: if a module states that it implements a named standard, that standard becomes part of the contract. Likewise, code presented as implementing a mathematical function or established scientific concept is expected to follow the corresponding mathematical or scientific definition.

Repository-independent failures. Some behavior is unambiguously undesirable without a repository-specific contract, including segmentation faults, unintended deletion of user data, UI freezes, and infinite loops. We apply this category conservatively: the expected behavior must be sufficiently clear across reasonable interpretations that documenting it explicitly would be superfluous. In fact, only two bugs in the end solely rely on this category and they are listed below.

Human validation. In a human study of 50 sampled bugs from the final dataset (§A.4.1), a 3-of-4 majority of reviewers found that all 50 bugs met the discoverability criterion and for all but four bugs, this vote was unanimous. Each of the four bugs received a dissenting vote on discoverability from only one reviewer (3 rejections and 1 unsure vote).

Agentic validation. In addition to the human validation, we also used agents to find repository contracts. The corresponding validation study is described in §A.4.2 and §A.4.3 and corroborates the reliability of the filtering process.

Annotation examples. The examples below pair each source of evidence with a real accepted candidate; each caption links to the evidence in the repository. We first show repository-defined contracts, then repository-independent failures, and close with rejected candidates.

Repository-defined contracts. The first set of examples consists of candidates that have clear repository-defined contracts.

The most direct evidence is documentation that users read. In the first two examples, the API documentation and a docstring state the expected behavior, and the code does not follow it.

Gold fix (truncated)

```
  raise ContentTypeError(
      self.request_info,
      self.history,
+     status=self.status,
```

Accepted: aiohttp PR:#8745. Documentation: the API docs define `status` as the HTTP response status.

Gold fix (excerpt)

```
+     mult = freq.n

  if periods is not None:
      periods = periods * mult
```

Accepted: pandas PR:#53709. Docstring: `period_range` takes `freq` from a Period `start`/`end`; the code dropped the multiplier.

The contract does not have to be written in prose. In sqlglot, a type annotation promises a return type. In SciPy, an existing test already asserts the expected result.

Gold fix (excerpt)

```
  @property
  def name(self) -> str:
-     if isinstance(self.this, Func):
+     if not self.this or isinstance(self.this,
      Func):
          return ""
      return self.this.name
```

Accepted: sqlglot PR:#4699. Type annotation: `Table.name` is declared `-> str` but can raise `AttributeError`.

Gold fix (excerpt)

```
  # If there are NaNs we apply _toint64()
  if np.isnan(np.sum(x)):
      x = _toint64(x)
- if np.isnan(np.sum(x)):
+ if np.isnan(np.sum(y)):
      y = _toint64(y)
```

Accepted: SciPy PR:#12955. Pre-existing test: `test_weightedtau` asserts the NaN-in-`y` result that the typo `isnan(sum(x))` breaks.

Expected behavior can also follow from how the code is used and named. In Netty, the normal code path shows that the response must be passed on unreleased. In click, the name `last_option` tells the reader which option the variable must hold.

Table 12: **Acceptance and agreement for the discoverable-bug review.** Each percentage is a bug-level average over all 9k bug candidates that pass all non-LM checks.

Agreement		Acceptance rate	
Unanimous agreement	**79%**	**Unanimous accept**	**60%**
Opus 5 (high) vs. Sol 5.6 (high)	84%	Opus 5 (high)	64%
Opus 5 (high) vs. Gemini 3.1 Pro	87%	Sol 5.6 (high)	73%
Sol 5.6 (high) vs. Gemini 3.1 Pro	87%	Gemini 3.1 Pro	74%

Gold fix (excerpt)

```
  } catch (Throwable t) {
-     release(response);
      ctx.fireExceptionCaught(t);
      removeThisHandler(ctx);
```

Accepted: Netty PR:#15602. Caller/example: this catch path releases a response that the normal path forwards unreleased.

Gold fix (excerpt)

```
      if _start_of_option(ctx, arg):
          last_option = arg
+         break

  return last_option is not None and last_option
      in param.opts
```

Accepted: click PR:#2816. Identifier: `last_option` should hold the last option, but a missing `break` lets an earlier one overwrite it.

Other code in the same repository can also define the expected behavior. In Jest, `fromJSON` must undo `toJSON`. In dask, sibling functions show how `eye` should handle `dtype=None`.

Gold fix (excerpt)

```
  private static mapFromArrayRecursive(
    arr: ReadonlyArray<[string, unknown]>,
  ): Map<string, unknown> {
-   if (arr[0] && Array.isArray(arr[1])) {
+   if (arr[0] && Array.isArray(arr[0][1])) {
      arr = arr.map(el => [
```

Accepted: Jest PR:#16353. Internal invariant: a wrong array index breaks the `fromJSON(toJSON(x))` round trip of the `duplicates` map.

Gold fix (excerpt)

```
  if M is None:
      M = N
+ if dtype is None:
+     dtype = float
```

Accepted: dask commit e25284d. Parallel implementation: `linspace` and `arange` replace `dtype=None` with a default; `eye` crashed on it.

Last, a repository can point to a definition outside of it. Luxon parses dates in the format of a named RFC, and the NumPy docstring gives the mathematical formula for the variance.

Gold fix (excerpt)

```
  const obsOffsets = {
    GMT: 0,
+   UT: 0,
    EDT: -4 * 60,
    EST: -5 * 60,
```

Accepted: luxon PR:#1782. Referenced standard: the RFC 2822/5322 grammar allows the `UT` zone, but `fromRFC2822` returns an invalid date for it.

Gold fix (excerpt)

```
  x = asanyarray(arr - arrmean)
- if issubclass(arr.dtype.type, nt.
      complexfloating):
-     x = um.multiply(x, um.conjugate(x), out=x)
      .real
- else:
+ if issubclass(arr.dtype.type, (nt.floating, nt
      .integer)):
      x = um.multiply(x, x, out=x)
+ else:
+     x = um.multiply(x, um.conjugate(x), out=x)
      .real
```

Accepted: NumPy PR:#13177. Math definition: the docstring defines `var = mean(abs(x - x.mean())**2)`; `x*x` drops the `abs` for complex input.

Repository-independent failures. A crash, hang, or data loss is unambiguously undesirable even with no repository contract, so such a failure grounds discoverability on its own. It is usually one of several sources of evidence for a bug; across the whole benchmark only two bug instances rest on

such a failure with no repository contract at all: the go-git race and OpenCV out-of-bounds read in the final examples below.

Gold fix (excerpt)

```
  func stringify(v any) string {
    switch s := v.(type) {
    ...
+   case nil:
+       return "<nil>"
    }
    return reflectlite.TypeOf(v).String()
```

Accepted: Go commit 70e453b. Crash: a `nil` value reaches `reflectlite.TypeOf(nil).String()`, which panics.

Gold fix (excerpt)

```
  currentItemState = ItemProcessingState.INITIAL
      ;
  requestToExecute = null;
  executionResult = null;
+ retryCounter++;
  assertInvariants(ItemProcessingState.INITIAL);
```

Accepted: OpenSearch PR:#11153. Hang: a retry loop never incremented `retryCounter`, so a failing operation was retried forever.

The next two examples show data loss in Airflow and the first of the two bugs that have no repository contract at all, but are still considered discoverable.

Gold fix (excerpt)

```
  )
- session.expunge_all()
+ if conn:
+     session.expunge(conn)
  return conn
```

Accepted: Airflow PR:#63080. Data loss: `session.expunge_all()` also detaches rows that the caller queued but did not commit, so they are never written.

Gold fix (excerpt)

```
- offsetHashIsFull bool
- mu                sync.RWMutex
+ offsetBuildOnce sync.Once
+ mu              sync.RWMutex
```

Accepted: go-git PR:#1729. Crash (no repository contract): concurrent `FindHash` calls race on `offsetHash` and hit Go's fatal concurrent-map error.

Like the go-git race, the OpenCV bug below has no repository contract. However, reading outside of a lookup table is a memory-safety error in any C++ program.

Gold fix (excerpt)

```
  int tz = cz >> (lab_base_shift - lab_lut_shift
      );
+ CV_DbgCheck(tx, tx >= 0 && tx < LAB_LUT_DIM, "
      ");
+ CV_DbgCheck(ty, ty >= 0 && ty < LAB_LUT_DIM, "
      ");
+ CV_DbgCheck(tz, tz >= 0 && tz < LAB_LUT_DIM, "
      ");
```

Accepted: OpenCV PR:#21257. Crash (no repository contract): an out-of-range value indexes the color LUT out of bounds.

Most non-discoverable candidates are dominated by changes that *add* a feature or capability rather than fix behavior the repository already implies; a minority are performance tweaks or changes mislabeled as fixes. The examples below were rejected at this stage. In the first one, the tests expect gson to turn a JSON string into a number, but nothing in the repository promises this. The quarkus change adds a new value to an enum, which is a new feature.

Gold fix (excerpt)

```
  case NUMBER:
+ case STRING:
    return new LazilyParsedNumber(
  default:
    throw new JsonSyntaxException(
```

Rejected: gson PR:#964. Not discoverable: no documentation promises that a JSON string becomes a Number.

Gold fix (excerpt)

```
  enum HttpMethod {
      GET,
      HEAD,
+     PATCH,
      POST,
      PUT,
```

Rejected: quarkus PR:#51729. New feature: adds `PATCH` to the `HttpMethod` enum.

A change can also look like a fix but only add something that was missing. The SymPy printer stated that `Min` and `Max` were not supported, and the OpenSearch PR is labeled as a bug fix but only adds a new getter.

Gold fix (excerpt)

```
  _known_functions = {
      'Abs': 'abs',
+     'Min': 'min',
+     'Max': 'max',
  }
```

Rejected: SymPy PR:#22914. New feature: adds printer support for `Min`/`Max`.

Gold fix (excerpt)

```
  public long getCreationTime() {
      return creationTime;
  }
+ public long getKeepAlive() {
+     return keepAlive;
+ }
```

Rejected: OpenSearch PR:#14495. Claim vs. code: only adds a `getKeepAlive()` getter.

The last two candidates do not change wrong behavior at all. The Rails change makes a correct query faster, and the React change adds support for a new HTML attribute.

Gold fix (excerpt)

```
+ if where_clause.contradiction?
+   return @async ? Promise::Complete.new({}) :
      {}
+ end
```

Rejected: Rails PR:#58058. Not a correctness bug: skips the query for a contradictory `WHERE`; the empty result was already correct.

Gold fix (excerpt)

```
  case 'autoPlay':
  case 'controls':
+ case 'credentialless':
  case 'default':
  case 'defer':
```

Rejected: React PR:#36148. New feature: adds `credentialless` to the known boolean HTML attributes.

B.8 Hidden test quality

Hidden tests must not be overly specific. The `appropriate_tests` annotation checks that the hidden tests are not overly specific: any reasonable fix for the bug should pass them. For example, a test should not check unrelated behavior or require the exact wording of an error message unless that wording is part of the repository's contract. Appendix E.2.4 gives the full prompt. Table 13 summarizes acceptance and agreement for this review.

Human validation. In a human study of 50 sampled bugs from the final dataset (§A.4.1), a 3-of-4 majority of reviewers found that all 50 bugs meet the appropriate-tests criterion, and all four reviewers found that 47 do. Each of the remaining 3 bugs drew an objection from only one of the four reviewers.

Table 13: **Acceptance and agreement for the appropriate-tests review.** Each percentage is a bug-level average over all 9k bug candidates that pass all non-LM checks.

Agreement	
Unanimous agreement	**86%**
Opus 5 (high) vs. Sol 5.6 (high)	88%
Opus 5 (high) vs. Gemini 3.1 Pro	93%
Sol 5.6 (high) vs. Gemini 3.1 Pro	90%

Acceptance rate	
Unanimous accept	**76%**
Opus 5 (high)	88%
Sol 5.6 (high)	80%
Gemini 3.1 Pro	84%

Test patch (excerpt)

```
+ def test_writeable(self):
+     a = np.arange(5)
+     a.flags.writeable = False
+
+     with pytest.raises(ValueError):
+         np.putmask(a, a >= 2, 3)
```

Accepted: NumPy PR:#17884. The sibling `np.put` already raises this `ValueError`, so any writeable-flag check in `putmask` passes.

Test patch (truncated)

```
+ message = "`a` must be 1-dimensional."
+ with pytest.raises(ValueError, match=message):
+     stats.percentileofscore(np.ones((3, 3)), 1
```

Rejected: SciPy PR:#23734. Pins the exact error text the pull request itself introduced; an equally correct fix worded differently would fail.

Over-specificity takes forms other than error-message wording. The next two rejections pin an internal artifact of one particular fix and an incidental output shape, respectively.

Test patch (excerpt)

```
+ from mlflow.rfunc.backend import RFuncBackend,
      _r_quote
+ assert _r_quote(value) == expected
```

Rejected: MLflow PR:#23179. Imports and asserts on `_r_quote`, a helper this fix happens to introduce; an equally correct inline escape would fail.

Test patch (excerpt)

```
+ @pytest.mark.parametrize(
+     ("version", "expected"),
+     [
+         ("git version 2.17", (2, 17)),
+         ("git version 2.18.1", (2, 18)),
+         ("git version 2.35.GIT", (2, 35)),
+         ("oh my git version 2.37.GIT", ()),
+         ("git version 2.GIT", ()),
+     ],
+ )
```

Rejected: pip PR:#12281. Pins each version string to an exact two-tuple; a fix that returned `(2, 18, 1)` for `2.18.1` would be equally correct and fail.

Changes to tests must preserve existing test expectations. This rubric checks that the test patch does not replace an expectation already asserted by the repository's test suite with a contradictory one. Both the evaluation rule and the inference prompt state that agents must not break existing base commit tests. A candidate whose gold fix breaks such a test would therefore not be solvable within the framework of the benchmark and is rejected. Similarly, bug instances that change test expectations might be more likely to conflict with each other. Appendix E.2.5 gives the full prompt. Table 14 summarizes acceptance and agreement for this review.

Human validation. In a human study of 50 sampled bugs from the final dataset (§A.4.1), all four reviewers found that all 50 bugs meet the criterion of preserving existing test expectations.

Additional validation. As described in §B.6, we also enforce R5 through a deterministic check: for each candidate, we apply its gold fix and run all base commit tests, rejecting the candidate if any test fails.

Table 14: **Acceptance and agreement for the preservation-of-expectations review.** Each percentage is a bug-level average over all 9k bug candidates that pass all non-LM checks.

Agreement		**Acceptance rate**	
Unanimous agreement	**98%**	**Unanimous accept**	**86%**
Opus 5 (high) vs. Sol 5.6 (high)	98%	Opus 5 (high)	87%
Opus 5 (high) vs. Gemini 3.1 Pro	98%	Sol 5.6 (high)	87%
Sol 5.6 (high) vs. Gemini 3.1 Pro	99%	Gemini 3.1 Pro	87%

Test patch (complete)

```
   {"RSA", testPrivateKey, SHA256WithRSA},
+  {"RSA-PSS-SHA256", testPrivateKey,
       SHA256WithRSAPSS},
   {"ECDSA-256", ecdsa256Priv, ECDSAWithSHA256},
   {"ECDSA-384", ecdsa384Priv, ECDSAWithSHA256},
```

Accepted: Go commit 1810462. One row appended to an existing test table, between untouched entries; no prior expectation is edited.

Test patch (truncated)

```
   for tensor in casted_tensors:
-      self.assertEqual(tensor.dtype, "float16")
+      self.assertEqual(tensor.dtype, "float32")
```

Rejected: Keras PR:#22137. An existing assertion is rewritten in place: the same inputs are now expected to cast to `float32` rather than `float16`.

Table 15: **Acceptance and agreement for the single-bug review.** Each percentage is a bug-level average over all 9k bug candidates that pass all non-LM checks.

Agreement		Acceptance rate	
Unanimous agreement	**93%**	**Unanimous accept**	**90%**
Opus 5 (high) vs. Sol 5.6 (high)	94%	Opus 5 (high)	93%
Opus 5 (high) vs. Gemini 3.1 Pro	96%	Sol 5.6 (high)	94%
Sol 5.6 (high) vs. Gemini 3.1 Pro	95%	Gemini 3.1 Pro	94%

Test patch (complete)

```
    assert_equal([], test.members)
    assert_equal({}, test.to_h)
+   assert_predicate(test, :frozen?)
  end
```

Accepted: Ruby PR:#15792. One assertion appended to an existing test, tightening it without revising any previous expectation.

Test patch (truncated)

```
  inheritComponent: ButtonBase,
  render,
- refInstanceof: window.HTMLDivElement,
+ refInstanceof: window.HTMLButtonElement,
  muiName: 'MuiAccordionSummary',
```

Rejected: MUI PR:#44408. The same conformance entry is edited: the rendered ref is now expected to be a button rather than a div.

B.9 Additional quality checks

Single unit of work. The bug instances are the smallest unit of scoring: an agent receives credit only when all of its hidden tests pass. Consequently, every instance must correspond to a single unit of work. The `is_single_bug` annotation checks that a bug instance's hidden tests correspond to a single root cause (that might or might not require multiple related changes to be fixed).

Appendix E.2.2 gives the full prompt. Table 15 summarizes acceptance and agreement for this review.

Shown below are four examples of accepted and rejected bug instances based on this criterion.

Gold fix (excerpt)

```
  config = {
      "num_tokens": self.num_tokens,
      "output_mode": self.output_mode,
+     "sparse": self.sparse,
  }
```

Accepted: Keras PR:#23375. One key missing from `get_config()`, so `sparse=True` is silently lost on a serialization round trip.

Gold fix (excerpt, two files)

```
- return value === 'NULL' ? `${key} IS NULL` : [
      key,
+ if (value === 'NULL') {
+   if (smth.comparator === '=') {
+     smth.comparator = 'IS';
- return arr.some(arg => _.isPlainObject(arg));
+ return arr.some(arg => _.isPlainObject(arg) ||
```

Rejected: Sequelize PR:#11454. Two unrelated fixes in one patch: `NULL` comparator handling in the query generator, and `Where` support in an array helper.

Gold fix (excerpt)

```
  func (r *Router) Find(method, path string, c
      Context) {
    ctx := c.(*context)
-   ctx.path = path
    currentNode := r.tree // Current node as
      root
```

Accepted: echo PR:#2385. One stray assignment copied the raw path into the context before matching; deleting it is the whole fix.

Gold fix (excerpt)

```
      'Max[*x]': 'Max(*x)',
      'Min[*x]': 'Min(*x)',
+     'Pochhammer[x,y]':'rf(x,y)',
+     'ArcTan[x,y]':'atan2(y,x)'
```

Rejected: SymPy PR:#19135. Two independent parser rules added at once; either could ship without the other.

Human validation. In a human study of 50 sampled bugs from the final dataset (§A.4.1), a 3-of-4 majority of reviewers found that all 50 bugs meet the single-bug criterion, and all four reviewers found that 49 do. The remaining bug drew an objection from only one of the four reviewers.

Compatibility of all bug instances within a repository. We need to establish that all tests in the same repository can be satisfied together. We therefore run one agent per repository using the same agent setup as described in §A.4.2. The judge receives the base repository and an index of all selected test patches. It is instructed to inspect every patch and report pairs that require mutually exclusive behavior, with exact quotes from both patches. We verify that every bug identifier is returned, then compute the largest compatible subset from the reported conflicts. This procedure did not report any conflicting pairs. Appendix E.3.3 gives the complete prompt.

C ADDITIONAL EXPERIMENTAL RESULTS

This appendix provides additional performance metrics (§C.1) that supplement the main results in §4. Furthermore, §C.2 significantly extends the trajectory findings in §5. Finally, §C.3 provides per-repository results for all model variants.

C.1 PERFORMANCE METRICS

This section provides additional breakdowns of the main results. We compare model performance with token usage, examine different ways of averaging scores, break down results by programming language, and compare performance across repositories of different sizes.

Token usage. Figure 14 shows the percentage of bugs resolved versus total output tokens, similar to the comparison in Figure 4. Bugs resolved rise with total output tokens, especially within a model family.

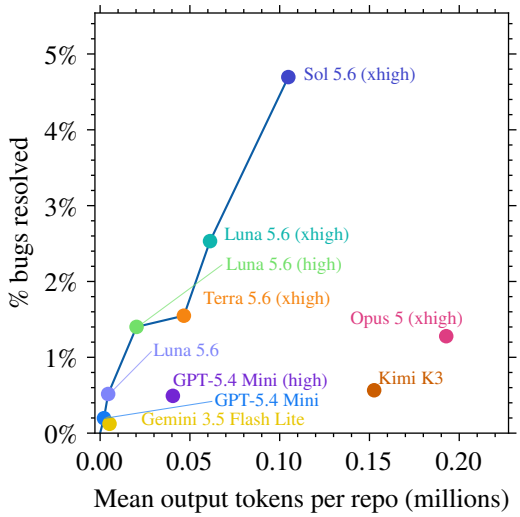


Figure 14: Percentage of bugs resolved versus mean total output tokens per repository; the line marks the Pareto frontier.

Performance metrics and averaging. Table 16 compares the percentage of bugs resolved across models. No repository is fully repaired by any model, with the best repaired repository having around half of its bugs resolved. The table also shows two different averaging methods to compute the same score: The bug-micro averaging gives each evaluated bug equal weight, while the repo-macro averaging gives each repository equal weight. Both metrics lead to similar rankings of the models. We choose the micro-average, because it is conceptually easier, and equivalent to weighing the different repositories by the number of bugs they contain, which correlates heavily with difficulty (see Table 5).

Results by language. Table 17 compares results by language. Performance is low across all language groups. Non-compiled repositories have slightly higher average and union results than compiled repositories, while the differences among individual language groups are larger. Results for C/C++ repositories are the lowest overall, while JavaScript/TypeScript shows the highest results, most clearly for GPT-5.6 Sol.

Performance across repositories of different sizes. Figure 15 compares resolved-bug counts across small, medium, and large repositories.

Table 16: Percentage of bugs resolved under bug-micro averaging, which gives every evaluated bug equal weight, and repo-macro averaging, which gives every repository equal weight; highest percentage resolved in one repository; and share of fully repaired repositories.

Model	% bugs resolved (bug-micro)	% bugs resolved (repo-macro)	Best repo	Fully repaired
Sol 5.6 (xhigh)	4.7%	10.3%	55.6%	0.0%
Luna 5.6 (xhigh)	2.5%	6.6%	44.4%	0.0%
Terra 5.6 (xhigh)	1.5%	4.3%	40.0%	0.0%
Luna 5.6 (high)	1.4%	4.8%	50.0%	0.0%
Opus 5 (xhigh)	1.3%	3.1%	33.3%	0.0%
Kimi K3	0.6%	1.3%	20.0%	0.0%
Luna 5.6	0.5%	1.1%	22.2%	0.0%
GPT-5.4 Mini (high)	0.5%	1.6%	25.0%	0.0%
GPT-5.4 Mini	0.2%	0.4%	11.1%	0.0%
Gemini 3.5 Flash Lite	0.1%	0.1%	5.3%	0.0%

Table 17: Results by language. Average is the mean across the three displayed agents; union counts each bug resolved by any included run of those agents.

Language	Repos	Bugs	GPT-5.6 Sol	Opus 5	Luna 5.6	Average	Union
Compiled	58	2,202	4.0%	1.2%	2.0%	2.4%	5.8%
Non-compiled	42	1,866	5.5%	1.4%	3.1%	3.3%	7.8%
Python	25	1,201	4.5%	1.6%	2.7%	2.9%	7.2%
JavaScript/TypeScript	17	413	11.6%	1.9%	6.3%	6.6%	13.8%
C/C++	14	488	2.0%	1.2%	0.8%	1.4%	3.1%
Java	10	363	6.9%	1.4%	3.3%	3.9%	8.5%
Other	34	1,603	3.4%	0.9%	1.8%	2.0%	5.2%

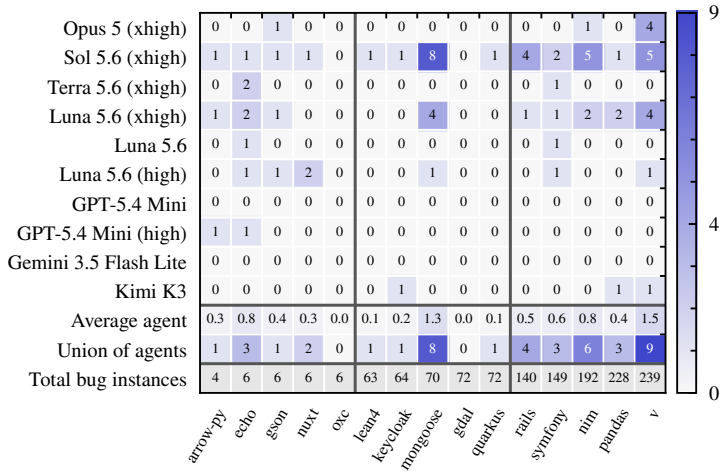


Figure 15: Number of bugs resolved for the five smallest repositories below 25 benchmark bugs, the five largest repositories with 25–74 bugs, and the five largest repositories with at least 75 bugs. Average agent is the mean across models; union of agents counts distinct bugs resolved by at least one model. The final row gives each repository's total benchmark bug instances.

C.2 AGENT BEHAVIOR

Expanding on §5, our evaluation across 4 models and 100 repositories reveals a substantial gap between reading the repository and actual bug discovery and repair: agents navigate to bug-relevant files far more frequently than they successfully generate complete fixes. This discrepancy highlights why current models resolve only a small fraction of bugs. Crucially, trajectory analysis indicates that agents frequently regress during execution, undoing previously correct edits in later steps. Furthermore, model performance exhibits strong diversity: different models resolve distinct populations of bugs, while scaling repeated attempts across longer action trajectories and multiple agentic inferences yields diminishing, rapidly plateauing returns.

Our trajectory dataset encompasses 400 runs across 4,165 bug instances (164 final outcomes were unavailable). Compared to the final benchmark dataset of 4,068 bugs used for all main results, the population used in this appendix included some bug instances that were later filtered out after all quality criteria were consistently applied. Since this corresponds to only around 2% of the benchmark population, we believe this does not significantly impact the findings of this section.

We define a *bug-relevant file* as any pre-existing source file modified by the gold fix. To track patch progression over time, evaluations are conducted at fixed checkpoints every 20 actions and upon run completion, totaling 5,173 checkpoints. When identical patches occur repeatedly within a trajectory, we record their last available evaluation.

Models differ in how broadly they explore the repository. Reading makes up 52.5% of Opus actions and 35.9% of Terra actions. Writes make up 25.5% of Opus actions and 37.8% of Terra actions, with Terra being the only model that writes more often than it reads (Table 18, Figure 16). Models also differ in file breadth: Opus reads a mean of 159.3 distinct files per repository, more than three times Terra's 48.9. Across models, agents often reach relevant code: Terra reads at least one bug-relevant file in 86 of its 100 runs, compared with 98 for Sol.

s

Table 18: How models explore the 100 repositories. Read, write, and execute are percentages of all actions; each action appears once. *Files read* is the mean number of different files read per repository, and *Reads per file* is the mean number of reads of each file. A run reaches a bug-relevant file when it reads any file changed by a gold fix.

Model	Read	Write	Execute	Files read	Reads per file	Runs reaching a bug-relevant file	First contact progress
Opus 5	52.5%	25.5%	18.8%	159.3	1.7	97/100	14.6%
Sol 5.6	42.5%	31.7%	19.5%	113.4	2.0	98/100	12.6%
Terra 5.6	35.9%	37.8%	19.4%	48.9	1.9	86/100	24.3%
Luna 5.6	48.6%	28.0%	17.7%	89.5	2.3	96/100	10.9%

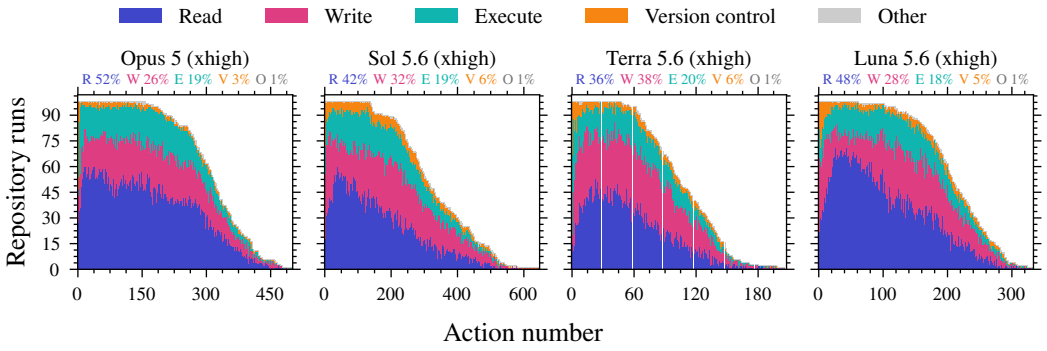


Figure 16: Distribution of observable actions for each model across 100 repositories. Actions are ordered from the start to the end of each run. At every action number, the figure shows the number of runs with a read, write, execute, version-control, or other action. Not every run is present at later actions because runs use different numbers of actions; the number does not measure performance. Labels show the percentage across all actions.

Agents read a small share of the repository, and how small depends on its size. The median run reads 2.6%–6.9% of a repository's tracked files (Table 19). Repository size spans three orders of magnitude, so we take the share within each repository rather than pooling. The distribution is skewed. Agents read most of a small tree; Opus reads 45 of `testify`'s 55 files. On a large tree they read only a few percent.

Table 19: **Agents read a small share of each repository.** The share is the number of distinct files an agent explicitly read divided by the number of tracked files at the base commit, computed per repository. Two outlier Opus runs (`rails`, `nim`) are excluded: in each, a single glob command matched and counted every file in a directory as read, reaching 10.7 and 4.5 files per action against a cohort median of 0.28–0.40. This inflation also affects the file counts reported above, which are therefore upper bounds.

Model	Runs	Median share of repository read	Largest share (repository)
Opus 5	98	3.8%	81.8% (`testify`)
Sol 5.6	100	6.9%	77.8% (`formulajs`)
Terra 5.6	100	2.6%	63.6% (`testify`)
Luna 5.6	100	4.4%	62.2% (`formulajs`)

Reading and editing bug-relevant files is far more common than resolving bugs. Across models, agents read at least one bug-relevant file for 23.4%–38.5% of bugs, edit one for 6.0%–21.5%, and resolve 1.3%–5.0% (Table 20). Changes to the same lines as a gold fix are less common still. Even when agents edit every bug-relevant file, most bugs are not resolved, showing that finding and changing relevant code does not always produce a correct repair.

Table 20: Reading, editing, and bug resolution across all bug instances. "Any bug-relevant file read" means that an agent reads at least one file changed by a gold fix; "Any bug-relevant file edited" means that an agent changes at least one. "Gold fix lines edited" requires an agent change to include a line changed by that fix. The final column is for bugs with every bug-relevant file edited and hidden-test results available.

Model	Any bug-relevant file read	Any bug-relevant file edited	Gold fix lines edited	Resolved	Resolved given all bug-relevant files edited
	36.2%	6.0%	0.6%	1.3%	22.8%
Opus 5	1,506/4,165	249/4,165	27/4,165	55/4,120	46/202
	38.5%	21.5%	3.7%	5.0%	24.4%
Sol 5.6	1,602/4,165	897/4,165	156/4,165	208/4,120	196/804
	23.4%	8.5%	1.2%	1.6%	19.9%
Terra 5.6	973/4,165	353/4,165	50/4,165	68/4,135	62/312
	37.9%	17.1%	2.0%	2.7%	16.4%
Luna 5.6	1,580/4,165	711/4,165	85/4,165	113/4,121	102/621

The edit-to-resolve gap holds regardless of what evidence a bug carries. Among 4,121 labeled bugs, at least one model edited a bug-relevant file for 1,175 (28.5%). Yet at least one model resolved only 252 of the 4,076 with available results (6.2%). This gap holds both for bugs with repository evidence only and for bugs with both sources (Table 21). The labels show which evidence is available for each bug, not which evidence an agent used.

Table 21: Evidence sources and results for labeled bugs. Each bug is present once. "Any edit" means that at least one of 4 models edited a file changed by a gold fix. The resolved percentage is for bugs resolved by at least one model. Bugs without available hidden-test results are not included in the resolved percentage, and bugs without evidence labels are not present.

Evidence available	Bugs	Any edit	Resolved
Repository evidence only	3,894	1,132/3,894 (29.1%)	243/3,849 (6.3%)
Repository-independent failure only	2	1/2 (50.0%)	0/2 (0.0%)
Both	225	42/225 (18.7%)	9/225 (4.0%)
All labeled bugs	4,121	1,175/4,121 (28.5%)	252/4,076 (6.2%)

Models read before they edit or resolve bugs. Models read before they make any file edits. By the halfway mark (measured in total actions performed), the number of bug-relevant files read is 65.6%–76.6% of the final number, the number edited is 12.8%–53.0%, and the number of bugs resolved is 20.5%–52.6% (Figures 17–19). Agents thus do more of their reading early in a run, while editing and bug recovery continue later.

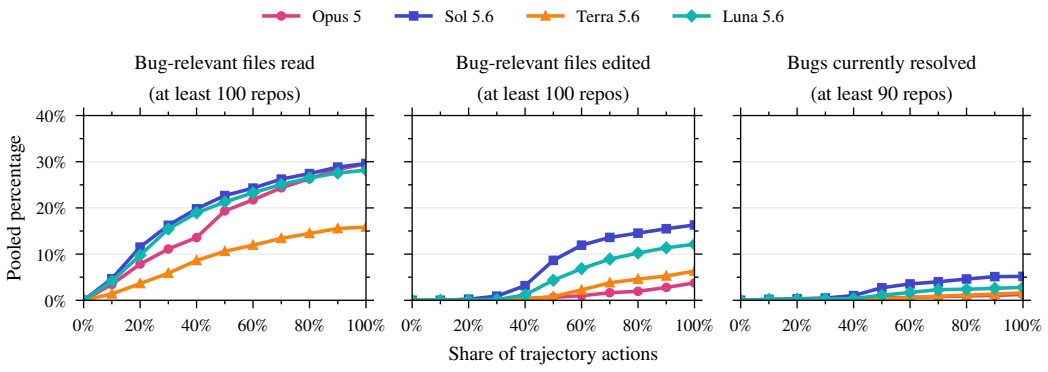


Figure 17: Bug-relevant files read, bug-relevant files edited, and bugs resolved over the percentage of actions in each run. All three results use the same range from 0 to 40%. Each curve is the percentage across all bugs in all repositories. Reading changes after each action; editing and resolution change only when a patch is evaluated. We do not count missing hidden-test results as failures.

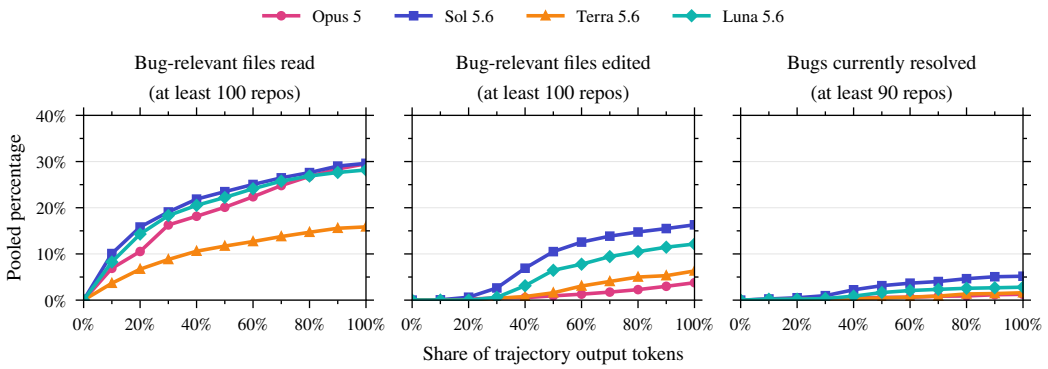


Figure 18: The same reading, editing, and resolved-bug results with progress measured by output tokens. Progress is from 0% to 100% for every run. Reading changes after each action; editing and resolution change only when a patch is evaluated. We do not count missing hidden-test results as failures.

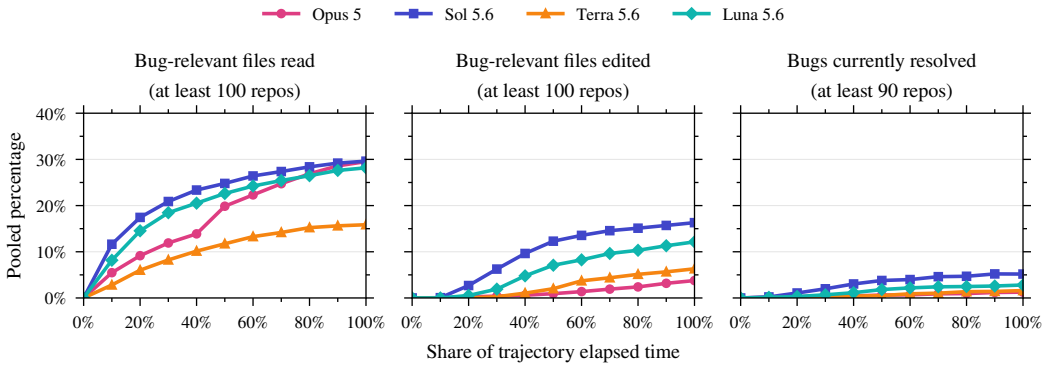


Figure 19: The same reading, editing, and resolved-bug results with progress measured by elapsed time. Progress is from 0% to 100% within each run. Repository counts report how many runs have available time and hidden-test results. We do not count missing results as failures.

Agents change more lines across a run than the final patch contains. Across models, the median end-state patch ranges from 14.0 to 507.5 lines, while the median total number of lines changed across the trajectory (i.e., total churn) is 29.0 to 635.0 lines (Figure 20, Table 23). Both numbers are agent changes; neither is the gold fix. We see that agents *churn* frequently, changing many lines that are eventually not present in the final patch. Furthermore, most agent changes are not in the locations changed by a gold fix. In the submitted patches, only 17.3%–27.9% of files changed by agents are bug-relevant. Of changed-line ranges, only 2.1%–7.1% include lines from a gold fix. These percentages use one gold fix per bug and do not measure correctness, because a valid repair may change other code: most changed files and line ranges are not part of that repair.

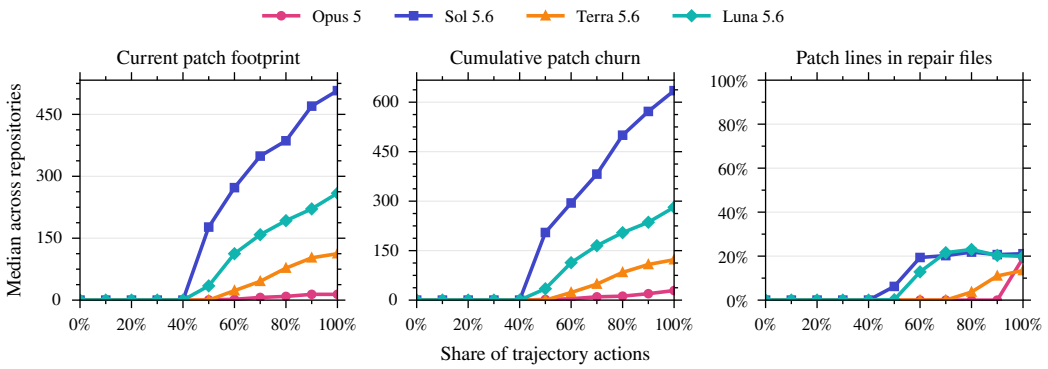


Figure 20: Patch changes over the percentage of actions in each run. The first result shows the number of lines changed in the current patch. The second shows all line changes from one patch to the next across the run. The third shows the percentage of current changed lines in bug-relevant files. Each result is the median across 100 repositories. At 100%, the patch is the submitted patch.

Later work can undo correct changes, but most runs finish at their best checkpoint. Across the 400 runs at increasing-time checkpoints, 42 lose at least one previously resolved bug after the patch changes (Table 22): these patch changes undo 93 previously resolved bugs. 28 model-bug repairs pass at some point but not at the end, but only 19/400 runs finish below their best observed checkpoint. A run's best checkpoint generally has the largest number of bugs resolved. Among runs with at least one bug resolved, models spend a median 24.8%–37.6% of their actions after the last increase in bugs resolved. We count a regression only when the patch changes; repeated evaluations of unchanged patches are reported below. Because we evaluate patches only at checkpoints, we know the first checkpoint at which a bug is resolved, not the exact action that resolved it.

Table 22: **Agents can undo correct repairs and continue working after their best checkpoint.** A regression is a bug resolved before a patch change but not resolved after it; this is about bugs resolved, not about the regression tests. "Runs with regression" counts runs with at least one regression. "Resolved before but not at end" counts bugs resolved at some checkpoint but not at the final checkpoint. "Runs below best" counts runs with more bugs resolved at a checkpoint than at the final checkpoint. "Actions after last increase" is the median remaining percentage of actions after the last increase in bugs resolved, among runs that resolve at least one bug.

Model	Runs with regression	Resolved before but not at end	Runs below best	Actions after last increase
Opus 5	6/100	5	4	25.9%
Sol 5.6	17/100	12	7	24.8%
Terra 5.6	5/100	5	3	25.9%
Luna 5.6	14/100	6	5	37.6%

Table 23: Additional results on patch changes and regressions by model. Patch lines and total changes show the median across repositories. For the same patch, we use its last available evaluation; we report a regression only when the patch changes. "Known checkpoints" is the number of checkpoints with an available evaluation out of all checkpoints for that model. "Final patch lines" is the size of the final patch, "Maximum patch lines" is the largest number of changed lines at any checkpoint, and "Total churn" is the total number of lines changed across all patch changes in a run. "Runs with regression" uses the definition above. "Transient repairs" is "Resolved before but not at end," and "Work after last improvement" is "Actions after last increase."

Model	Known checkpoints	Final patch lines	Maximum patch lines	Total churn	Runs with regression	Transient repairs	Work after last improvement
Opus 5	1,526/1,670	14.0	22.0	29.0	6/100	5	25.9%
Sol 5.6	1,590/1,756	507.5	557.5	635.0	17/100	12	24.8%
Terra 5.6	563/615	112.5	114.0	122.5	5/100	5	25.9%
Luna 5.6	1,030/1,132	259.0	267.0	281.5	14/100	6	37.6%

Agents stop because they judge their work done, not because they run out of budget. No run approaches either limit (Table 24). Against a 1000-action limit, the longest run uses 648. Against a 12-hour wall-clock limit, the longest takes 5.5 (45.6%). All 400/400 runs end by submitting. In their closing messages, 0 runs report an exhausted search or budget. Stopping is deliberate: 385 runs inspect their diff or status in their last five commands. The task prompt asks agents to fix as many bugs as they can, yet they stop well within their budget.

Table 24: **No run exhausts its action or time budget.** Medians and maxima are taken over the 100 repositories for each model. "At step limit" counts runs reaching the 1000-step limit and "at time limit" counts runs reaching the 12-hour wall-clock limit. Actions are shell commands; a single model turn may issue more than one.

Model	Median actions	Most actions	At step limit	Median hours	Most hours	At time limit
Opus 5	324	502	0/100	1.3	3.9	0/100
Sol 5.6	322	648	0/100	1.6	5.5	0/100
Terra 5.6	110	226	0/100	0.3	1.9	0/100
Luna 5.6	214	334	0/100	1.5	3.1	0/100

Agents search by writing throwaway programs, not only by reading files. For three of four models the second most common activity is piping a Python program into `python3 -`, used both to search (ad-hoc tree-walkers scanning for suspect patterns) and to edit (programs that rewrite source in place).

Agents test with their own programs more than with the project's test suite. On the 29 repositories where the graded code imports as a Python module, 79%–86% of runs write a program that imports it directly. This exercises the code rather than invoking the project as a user would. Running the project's own test suite is a minority of execution actions (5%–28%).

Different models resolve different bugs. Among bugs with results for all 4 models, 159 are resolved by one model only, while only 8 are resolved by all 4 models. 3,849 (93.4%) are not resolved by any model (Table 25). The final four rows show that every model resolves bugs that no other model resolves. No model therefore resolves all bugs resolved by the other models.

Table 25: Number of models resolving the same bug. Percentages use only bugs with available results for all 4 models. The last results show how many bugs are resolved by each model and no other model.

Resolving models	Bugs	Share
No model	3,849	93.4%
Exactly one model	159	3.9%
Exactly two models	60	1.5%
Exactly three models	44	1.1%
All four models	8	0.2%
Opus only	20	0.5%
Sol only	104	2.5%
Terra only	7	0.2%
Luna only	28	0.7%

Repeated attempts recover additional bugs, but gains diminish. On a 5-repository cohort, 10 sequential Luna xhigh attempts raise bug recovery from 1.6% to 3.5%, while the union of 10 independent attempts reaches 4.1% (Figure 7). With default reasoning, sequential attempts raise recovery from 0.1% to 0.9%, while independent attempts reach 0.6%. A single Sol xhigh run resolves 2.1% of bugs, between the first and final sequential Luna xhigh results. Independent runs resolve more with xhigh reasoning; sequential runs resolve more with default reasoning. Both recover additional bugs, but gains are smaller for later attempts.

Reading a file does not mean reading all of its lines. By action 140, Opus has read a mean of 35.8% of bug-relevant files per repository and a mean of 23.8% of all lines in those files. The corresponding Sol results are 39.9% and 9.0% (Figure 21). File breadth and line coverage therefore measure different parts of exploration. Exact line ranges are available for 77.6% of Opus read requests and 55.6% of Sol requests. The lower line result for Sol reflects both exploration and the smaller share of reads with exact ranges. Because some reads do not have exact ranges, we use files for the primary analysis and lines as an additional analysis.

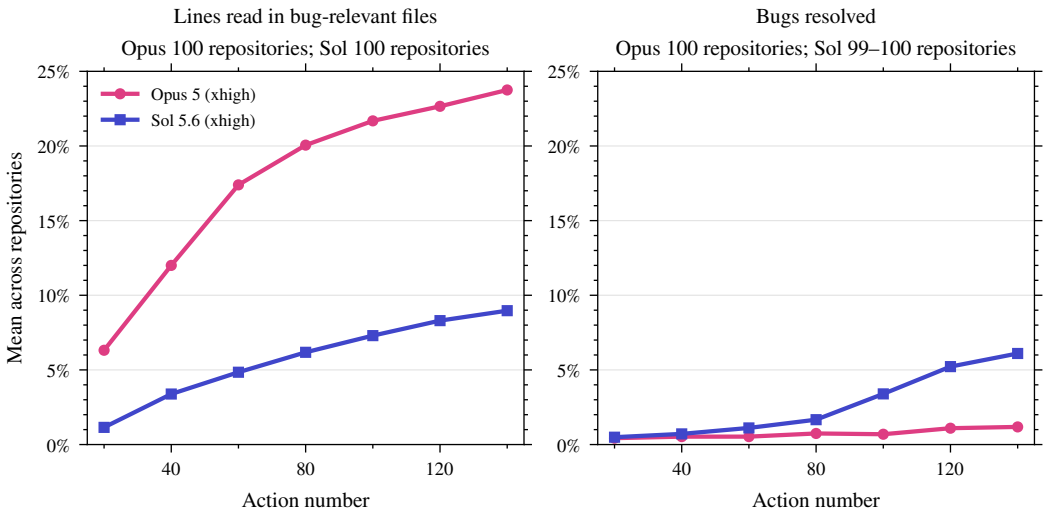


Figure 21: Lines read in bug-relevant files and bugs resolved by action number for Opus and Sol. Each result is the mean across repositories. Lines are included only for reads with exact line ranges. Bug resolution is from the most recent evaluated patch.

C.3 PER-REPOSITORY RESULTS

Table 26 reports the percentage of bugs resolved for all 10 measured model variants across all 100 benchmark repositories.

Table 26: Percentage of bugs resolved for all 10 measured model variants across all 100 benchmark repositories. Blank cells indicate unavailable evaluations.

Repo	Opus 5 (xhigh)	Sol 5.6 (xhigh)	Terra 5.6 (xhigh)	Luna 5.6 (xhigh)	Luna 5.6 (default)	Luna 5.6 (high)	GPT-5.4 Mini (default)	GPT-5.4 Mini (high)	Gemini 3.5 Flash Lite (default)	Kimi K3 (max)
agda	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
arrow	2.1%	2.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
arrow-py	0.0%	25.0%	0.0%	25.0%	0.0%	0.0%	0.0%	25.0%	0.0%	0.0%
astropy	1.2%	2.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
aws-cdk	2.7%	5.4%	2.7%	2.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
babel	0.0%	3.2%	0.0%	0.0%	3.2%	0.0%	0.0%	0.0%	0.0%	0.0%
black	0.0%	9.5%	9.5%	9.5%	4.8%	9.5%	0.0%	4.8%	0.0%	4.8%
click	0.0%	46.2%	23.1%	15.4%	7.7%	7.7%	0.0%	7.7%	0.0%	0.0%
clippy	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
conan	4.2%	0.0%	8.3%	0.0%	0.0%	4.2%	0.0%	0.0%	0.0%	0.0%
cvc5	0.0%	2.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dart-sdk	2.0%	2.0%	2.0%	2.0%	2.0%	2.0%	2.0%	2.0%	2.0%	0.0%
dask	0.0%	4.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dataframes-jl	14.3%	28.6%	14.3%	14.3%	0.0%	28.6%	0.0%	0.0%	0.0%	14.3%
datafusion	0.0%	0.0%	2.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dateutil	30.0%	40.0%	40.0%	10.0%	10.0%	10.0%	0.0%	20.0%	0.0%	20.0%
dayjs	18.2%	54.5%	27.3%	27.3%	0.0%	18.2%	0.0%	0.0%	0.0%	18.2%
distributions-jl	0.0%	0.0%	0.0%	11.1%	0.0%	0.0%	0.0%	11.1%	0.0%	0.0%
dmd	0.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
dolphinscheduler	10.0%	30.0%	0.0%	10.0%	0.0%	30.0%	0.0%	0.0%	0.0%	0.0%
duckdb	1.8%	1.8%	0.9%	0.9%	0.9%	0.9%	0.9%	0.0%	0.9%	0.0%
echo	0.0%	16.7%	33.3%	33.3%	16.7%	16.7%	0.0%	16.7%	0.0%	0.0%
elixir	0.0%	17.0%	4.3%	0.0%	2.1%	2.1%	0.0%	0.0%	0.0%	0.0%
fasthttp	0.0%	0.0%	11.1%	22.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
formulajs	0.0%	55.6%	33.3%	33.3%	22.2%	33.3%	11.1%	11.1%	0.0%	0.0%
fsharp	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
gdal	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
gin	27.3%	27.3%	27.3%	9.1%	0.0%	9.1%	9.1%	9.1%	0.0%	0.0%
gitoxide	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
go	0.0%	3.1%	1.0%	0.0%	1.0%	0.0%	1.0%	0.0%	1.0%	0.0%
go-git	0.0%	21.7%	8.7%	13.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
goawk	7.1%	35.7%	14.3%	35.7%	0.0%	28.6%	0.0%	0.0%	0.0%	14.3%
godot	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
gorm	0.0%	31.6%	0.0%	10.5%	5.3%	0.0%	0.0%	0.0%	0.0%	0.0%
gson	16.7%	16.7%	0.0%	16.7%	0.0%	16.7%	0.0%	0.0%	0.0%	0.0%
hdf5	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
idris2	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
isomorphic-git	0.0%	38.5%	15.4%	15.4%	0.0%	7.7%	0.0%	0.0%	0.0%	0.0%
isort	0.0%	27.3%	9.1%	9.1%	0.0%	18.2%	0.0%	0.0%	0.0%	0.0%
jerryscript	20.0%	0.0%	6.7%	6.7%	0.0%	0.0%	0.0%	0.0%	0.0%	13.3%
jest	0.0%	10.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
keycloak	0.0%	1.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	1.6%
lean4	0.0%	1.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
libexpat	0.0%	11.1%	0.0%	11.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
luxon	28.6%	42.9%	21.4%	21.4%	0.0%	7.1%	0.0%	0.0%	0.0%	0.0%
material-ui	0.0%	5.1%	0.0%	2.6%	0.0%	2.6%	0.0%	0.0%	0.0%	0.0%
mathlib4	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
matplotlib	0.0%	1.3%	1.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
micropython	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
modelingtoolkit-jl	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
mongoose	0.0%	11.4%	0.0%	5.7%	0.0%	1.4%	0.0%	0.0%	0.0%	0.0%
mpmath	0.0%	18.8%	0.0%	12.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
mypy	0.0%	0.0%	0.0%	1.1%	0.0%	0.0%	0.0%	0.0%	0.0%	1.1%
nacos	0.0%	16.7%	0.0%	8.3%	8.3%	8.3%	0.0%	0.0%	0.0%	0.0%
netty	0.0%	10.4%	2.1%	5.2%	1.0%	1.0%	0.0%	1.0%	1.0%	0.0%
networkx	0.0%	5.6%	0.0%	11.1%	5.6%	0.0%	0.0%	0.0%	0.0%	0.0%
nim	0.5%	2.6%	0.0%	1.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
numpy	0.0%	0.8%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
nuxt	0.0%	16.7%	0.0%	0.0%	0.0%	33.3%	0.0%	0.0%	0.0%	0.0%
openapi-generator	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
opencv	0.0%	0.0%	0.0%	2.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
optuna	14.3%	0.0%	0.0%	14.3%	0.0%	14.3%	0.0%	0.0%	0.0%	0.0%
oxc	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
pandas	0.0%	0.4%	0.0%	0.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.4%
pendulum	33.3%	55.6%	33.3%	44.4%	0.0%	44.4%	0.0%	22.2%	0.0%	0.0%
php-src	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
phpspreadsheet	0.0%	0.0%	0.0%	11.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
pint	7.7%	23.1%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
prettier	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.9%
proj	0.0%	9.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
pulsar	0.0%	5.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Continued on next page

Repo	Opus 5 (xhigh)	Sol 5.6 (xhigh)	Terra 5.6 (xhigh)	Luna 5.6 (xhigh)	Luna 5.6 (default)	Luna 5.6 (high)	GPT-5.4 Mini (default)	GPT-5.4 Mini (high)	Gemini 3.5 Flash Lite (default)	Kimi K3 (max)
quarkus	0.0%	1.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
qutip	10.0%	10.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rails	0.0%	2.9%	0.0%	0.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
react-router	0.0%	0.0%	0.0%	14.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
roslyn	3.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
ruff	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
rxjava	7.7%	3.8%	15.4%	7.7%	7.7%	3.8%	7.7%	7.7%	0.0%	7.7%
scikit-learn	3.8%	3.8%	0.0%	2.5%	0.0%	1.2%	0.0%	1.2%	0.0%	0.0%
scipy	0.7%	2.2%	0.7%	0.7%	0.0%	0.0%	0.0%	0.7%	0.0%	0.7%
shapely	0.0%	55.6%	0.0%	44.4%	0.0%	33.3%	0.0%	0.0%	0.0%	11.1%
shardingsphere	0.0%	33.3%	0.0%	13.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
sqlglot	0.0%	3.8%	3.1%	3.8%	0.8%	2.3%	0.0%	0.8%	0.0%	0.0%
stylelint	0.0%	3.7%	0.0%	1.9%	0.0%	1.9%	0.0%	0.0%	0.0%	0.0%
sunpy	6.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	6.7%
svgo	9.1%	36.4%	27.3%	18.2%	0.0%	9.1%	0.0%	9.1%	0.0%	9.1%
swc	5.3%	5.3%	0.0%	5.3%	5.3%	5.3%	5.3%	5.3%	5.3%	5.3%
symfony	0.0%	1.3%	0.7%	0.7%	0.7%	0.7%	0.0%	0.0%	0.0%	0.0%
syn	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tarantool	0.0%	7.5%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
terser	0.0%	16.7%	33.3%	0.0%	0.0%	50.0%	0.0%	0.0%	0.0%	0.0%
testify	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
three-js	0.0%	11.1%	0.0%	44.4%	0.0%	11.1%	0.0%	0.0%	0.0%	0.0%
tinygrad	6.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
tokio	0.0%	16.7%	0.0%	8.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
trino	2.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.3%
v	1.7%	2.1%	0.0%	1.7%	0.0%	0.4%	0.0%	0.0%	0.0%	0.4%
webpack	0.0%	5.7%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
xarray	6.5%	9.7%	0.0%	3.2%	3.2%	0.0%	0.0%	0.0%	0.0%	0.0%
z3	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

D REPOSITORY INDEX AND STATISTICS

This appendix gives more details about the repositories in SWE-sweep and adds to the overview in §2.3. §D.1 gives more statistics about the repositories and their bugs. §D.2 lists all repositories. §D.3 provides more plots related to the SZZ algorithm that we use to estimate bug lifetimes.

D.1 ADDITIONAL DATASET STATISTICS

This section gives the full distributions behind the summary statistics in §2.3. It shows how large the repositories and bugs are, which programming languages appear in the files changed by the gold fixes, and how often the gold fixes of two bugs change the same file.

Figure 22 gives binned distributions of repository scale. Figures 23 and 24 give cumulative distributions of repository scale and the bug-level statistics in Table 3. Their means are 1.6 F2P tests, 50.4 P2P tests, 16.3 gold fix lines, and 37.0 test-patch lines per bug.

Table 27: **Gold fix overlap and separation.** The first row includes all benchmark bugs. The last two rows include pairs of bugs whose gold fixes each change exactly one existing source file, and both change the same file. Different content compares changed source lines after removing diff metadata and unchanged context. Distance is the minimum distance between the changed line ranges.

Measure	Share
Bugs sharing a bug-relevant file	56.3%
Same-file repair pairs with different changed content	96.7%
Same-file repair pairs more than 100 lines apart	80.7%

Table 28: Complete source-language breakdown for gold fixes underlying Figure 3. Bug counts can be fractional because a single gold fix can modify source files in multiple languages, resulting in fractional assignment of the bug to languages.

Language	File endings	Bugs	Share
Python	`.py`, `.pyi`	1,058.7	26.02%
C++	`.cc`, `.cpp`, `.cxx`, `.h`, `.hh`, `.hpp`, `.hxx`	385.8	9.48%
Java	`.java`, `.template`	381.5	9.38%
JavaScript	`.cjs`, `.js`, `.jsx`, `.mjs`	326.8	8.03%
V	`.v`	239.0	5.88%
Rust	`.rs`	226.5	5.57%
Nim	`.nim`	192.0	4.72%
Go	`.go`	189.0	4.65%
C	`.c`, `.h`	187.3	4.60%
PHP	`.php`	158.0	3.88%
Ruby	`.rb`	140.0	3.44%
D	`.d`	110.1	2.71%
TypeScript	`.cts`, `.mts`, `.ts`, `.tsx`	86.2	2.12%
Lean	`.lean`	71.0	1.75%
Cython	`.in`, `.pxd`, `.pxi`, `.pyf`, `.pyx`, `.tp`	58.7	1.44%
Haskell	`.hs`, `.hs-boot`	44.0	1.08%
Dart	`.dart`	43.8	1.08%
Elixir	`.ex`	42.0	1.03%
C#	`.cs`	29.5	0.73%
F#	`.fs`, `.fsi`	28.0	0.69%
Julia	`.jl`	26.0	0.64%
Idris	`.idr`	13.0	0.32%
Mustache	`.mustache`	12.5	0.31%
Erlang	`.erl`, `.yrl`	5.0	0.12%
Lua	`.lua`	5.0	0.12%
Fortran	`.f`, `.f77`, `.f90`, `.for`	4.0	0.10%
Visual Basic	`.vb`	1.5	0.04%
Alex	`.x`	1.0	0.02%
re	`.re`	1.0	0.02%
CMake	`CMakeLists.txt`	0.7	0.02%
Shell	`.sh`	0.5	0.01%
Total		4,068.0	100.00%

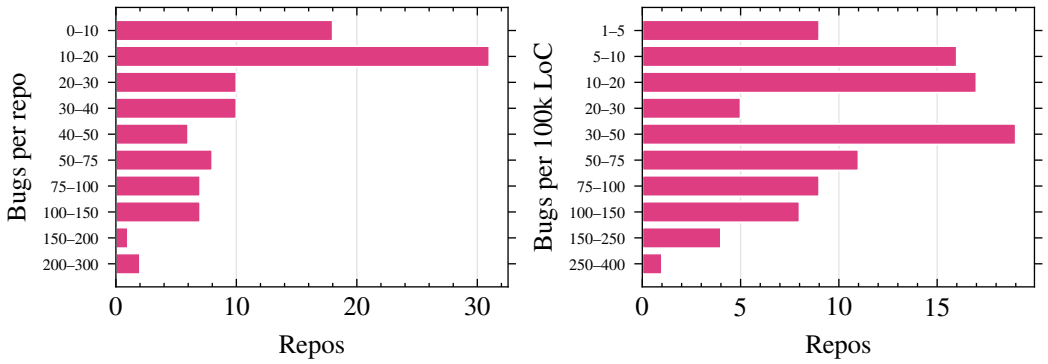
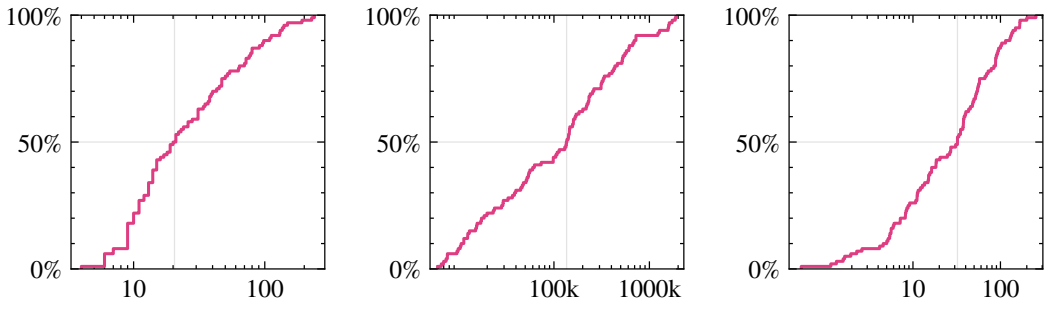
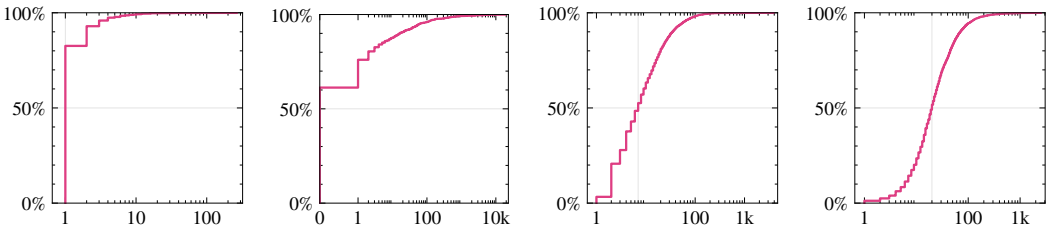


Figure 22: Binned repository-level distributions.



(a) Bugs per repo (b) Graded-source LoC (c) Bugs per 100k LoC

Figure 23: Empirical cumulative distributions of repository-level benchmark statistics. Faint-gray guides mark the median value and 50th percentile.



(a) F2P tests (b) P2P tests (c) Gold fix size (d) Test patch size

Figure 24: Empirical cumulative distributions of bug-level benchmark statistics. Faint-gray guides mark the median value and 50th percentile.

D.2 REPOSITORY INDEX

Table 29 expands the benchmark overview in §2.3. Repository names link to their upstream projects. License identifiers link to each project's license declaration; multiple identifiers indicate alternative or file-scoped terms.

Table 29: Repository index.

Repository	Language	Description	License	Bugs	LoC
agda	hs	Dependently typed programming language and interactive theorem prover	MIT	45	147k
arrow	cpp	Columnar data format and multi-language analytics libraries	Apache-2.0	47	311k
arrow-py	py	Date and time library for Python	Apache-2.0	4	8k
astropy	py	Core astronomy and astrophysics library for Python	BSD-3-Clause	80	229k
aws-cdk	ts	Framework for defining cloud infrastructure in code	Apache-2.0	37	200k
babel	js	JavaScript compiler and transformation toolkit	MIT	31	62k
black	py	Opinionated Python code formatter	MIT	21	15k
click	py	Composable command-line interface toolkit for Python	BSD-3-Clause	13	10k
clippy	rs	Linter collection for Rust	Apache-2.0;MIT	76	135k
conan	py	Package manager for C and C++	MIT	24	45k
cvc5	cpp	Automated SMT theorem prover	BSD-3-Clause	38	462k
dart-sdk	dart	Dart language SDK, runtime, compilers, and libraries	BSD-3-Clause	50	1,913k
dask	py	Parallel and distributed computing library for Python	BSD-3-Clause	21	56k
dataframes-jl	jl	In-memory tabular data library for Julia	MIT	7	17k
datafusion	rs	Extensible SQL query engine written in Rust	Apache-2.0	38	530k
dateutil	py	Extensions to Python date and time handling	Apache-2.0; BSD-3-Clause	10	6k
dayjs	js	Lightweight immutable date and time library for JavaScript	MIT	11	7k
distributions-jl	jl	Probability distributions and statistical functions for Julia	MIT	9	16k
dmd	d	Compiler for the D programming language	BSL-1.0	112	609k
dolphinscheduler	java	Distributed workflow orchestration platform	Apache-2.0	10	183k
duckdb	cpp	In-process analytical SQL database	MIT	109	723k
echo	go	High-performance web framework for Go	MIT	6	8k
elixir	ex	Functional programming language for the Erlang VM	Apache-2.0	47	146k
fasthttp	go	High-performance HTTP library for Go	MIT	9	24k
formulajs	js	JavaScript implementations of spreadsheet functions	MIT	9	13k
fsharp	fs	F# compiler, core library, and development tools	MIT	28	336k
gdal	cpp	Raster and vector geospatial data translation library	MIT	72	1,731k
gin	go	HTTP web framework for Go	MIT	11	7k
gitoxide	rs	Git implementation written in Rust	Apache-2.0;MIT	9	160k
go	go	Go programming language and toolchain	BSD-3-Clause	98	1,612k
go-git	go	Extensible Git implementation written in Go	Apache-2.0	23	40k
goawk	go	AWK interpreter with CSV support written in Go	MIT	14	10k
godot	cpp	Cross-platform 2D and 3D game engine	MIT	17	1,275k
gorm	go	Object-relational mapper for Go	MIT	19	11k
gson	java	JSON serialization library for Java	Apache-2.0	6	17k
hdf5	c	Library and file format for hierarchical scientific data	BSD-3-Clause	14	615k
idris2	idr	Dependently typed functional programming language	BSD-3-Clause	13	113k
isomorphic-git	js	Git implementation for JavaScript runtimes	MIT	13	15k
isort	py	Import sorter for Python	MIT	11	11k
jerryscript	c	Lightweight JavaScript engine for embedded systems	Apache-2.0	15	134k
jest	ts	JavaScript testing framework	MIT	20	52k
keycloak	java	Identity and access management server	Apache-2.0	64	573k
lean4	lean	Programming language and interactive theorem prover	Apache-2.0	63	233k
libexpat	c	Streaming XML parser written in C	MIT	9	19k
luxon	js	Date and time library for JavaScript	MIT	14	7k
material-ui	js	React component library implementing Material Design	MIT	39	105k
mathlib4	lean	Mathematics library for the Lean theorem prover	Apache-2.0	14	1,865k
matplotlib	py	Plotting library for Python	PSF-2.0	78	142k
micropython	c	Python implementation for microcontrollers	MIT	22	423k
modelingtoolkit-jl	jl	Symbolic modeling system for scientific computing in Julia	MIT	10	23k
mongoose	js	Object data modeling library for MongoDB and Node.js	MIT	70	55k
mpmath	py	Arbitrary-precision numerical computing library for Python	BSD-3-Clause	16	48k
mypy	py	Static type checker for Python	MIT	91	137k
nacos	java	Service discovery and configuration management platform	Apache-2.0	12	222k
netty	java	Event-driven network application framework for Java	Apache-2.0	96	254k
networkx	py	Graph analysis library for Python	BSD-3-Clause	18	98k
nim	nim	Statically typed systems programming language	MIT	192	218k
numpy	py	N-dimensional array library for numerical computing in Python	BSD-3-Clause	131	406k
nuxt	ts	Full-stack web framework for Vue	MIT	6	37k
openapi-generator	mustache	Generator for API clients, servers, and documentation from OpenAPI specs	Apache-2.0	31	434k
opencv	cpp	Computer vision and image processing library	Apache-2.0	47	553k
optuna	py	Hyperparameter optimization framework	MIT	7	28k
oxc	rs	High-performance JavaScript and TypeScript toolchain	MIT	6	521k
pandas	py	Data analysis library with labeled tabular structures	BSD-3-Clause	228	239k
pendulum	py	Date and time library for Python	MIT	9	12k
php-src	c	PHP language interpreter	BSD-3-Clause	26	1,570k
phpspreadsheet	php	Spreadsheet reader and writer for PHP	MIT	9	98k
pint	py	Physical quantities and unit conversion library for Python	BSD-3-Clause	13	11k

Continued on next page

Repository	Language	Description	License	Bugs	LoC
prettier	js	Opinionated code formatter for web languages	MIT	34	39k
proj	cpp	Cartographic projection and coordinate transformation library	MIT	21	141k
pulsar	java	Distributed publish-subscribe messaging platform	Apache-2.0	19	379k
quarkus	java	Kubernetes-native Java application framework	Apache-2.0	72	670k
qutip	py	Quantum dynamics simulation library for Python	BSD-3-Clause	10	74k
rails	rb	Web application framework for Ruby	MIT	140	157k
react-router	ts	Routing library for React	MIT	14	29k
roslyn	cs	C# and Visual Basic compilers and code analysis APIs	MIT	31	1,591k
ruff	rs	Python linter and code formatter written in Rust	MIT	52	318k
rxjava	java	Reactive programming library for the JVM	Apache-2.0	26	166k
scikit-learn	py	Machine learning library for Python	BSD-3-Clause	80	145k
scipy	py	Scientific computing library for Python	BSD-3-Clause	136	517k
shapely	py	Geometric object manipulation and analysis library for Python	BSD-3-Clause	9	20k
shardingsphere	java	Distributed SQL middleware for data sharding and governance	Apache-2.0	15	328k
sqlglot	py	SQL parser, optimizer, and transpiler for Python	MIT	130	51k
stylelint	js	Configurable linter for CSS and related languages	MIT	54	54k
sunpy	py	Solar physics data analysis library for Python	BSD-2-Clause	15	46k
svgo	js	SVG optimizer	MIT	11	13k
swc	rs	JavaScript and TypeScript compiler written in Rust	Apache-2.0	19	233k
symfony	php	Web application framework for PHP	MIT	149	1,228k
syn	rs	Parser for Rust source code	Apache-2.0	13	33k
tarantool	c	In-memory database and application server	BSD-2-Clause	40	312k
terser	js	JavaScript parser, mangler, and compressor	BSD-2-Clause	6	29k
testify	go	Assertions and mocking toolkit for Go tests	MIT	9	10k
three-js	js	3D graphics library for JavaScript	MIT	9	159k
tinygrad	py	Lightweight deep learning framework	MIT	15	170k
tokio	rs	Asynchronous runtime for Rust	MIT	12	108k
trino	java	Distributed SQL query engine	Apache-2.0	43	724k
v	v	Compiled programming language focused on simplicity and speed	MIT	239	262k
webpack	js	Module bundler for JavaScript and web assets	MIT	35	130k
xarray	py	Labeled multidimensional array library for Python	Apache-2.0	31	60k
z3	cpp	SMT theorem prover	MIT	11	691k

D.3 SZZ COVERAGE CURVES FOR ALL REPOSITORIES

Figures 25 and 26 show the coverage curves from the SZZ algorithm (§2.2) for all benchmark repositories.

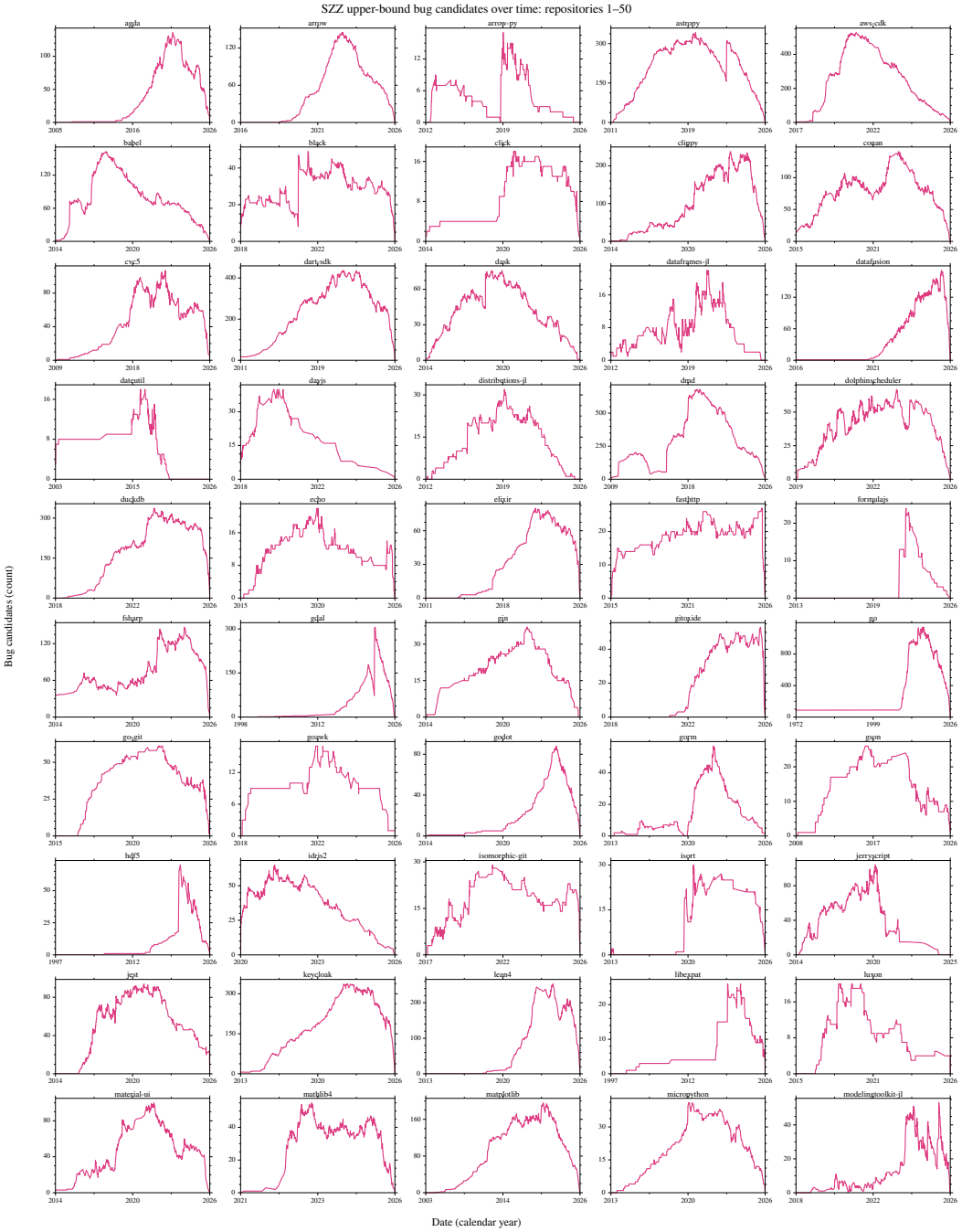


Figure 25: Estimates of the number of concurrent bug instance candidates across the histories of benchmark repositories 1–50, in the order listed in Table 29.

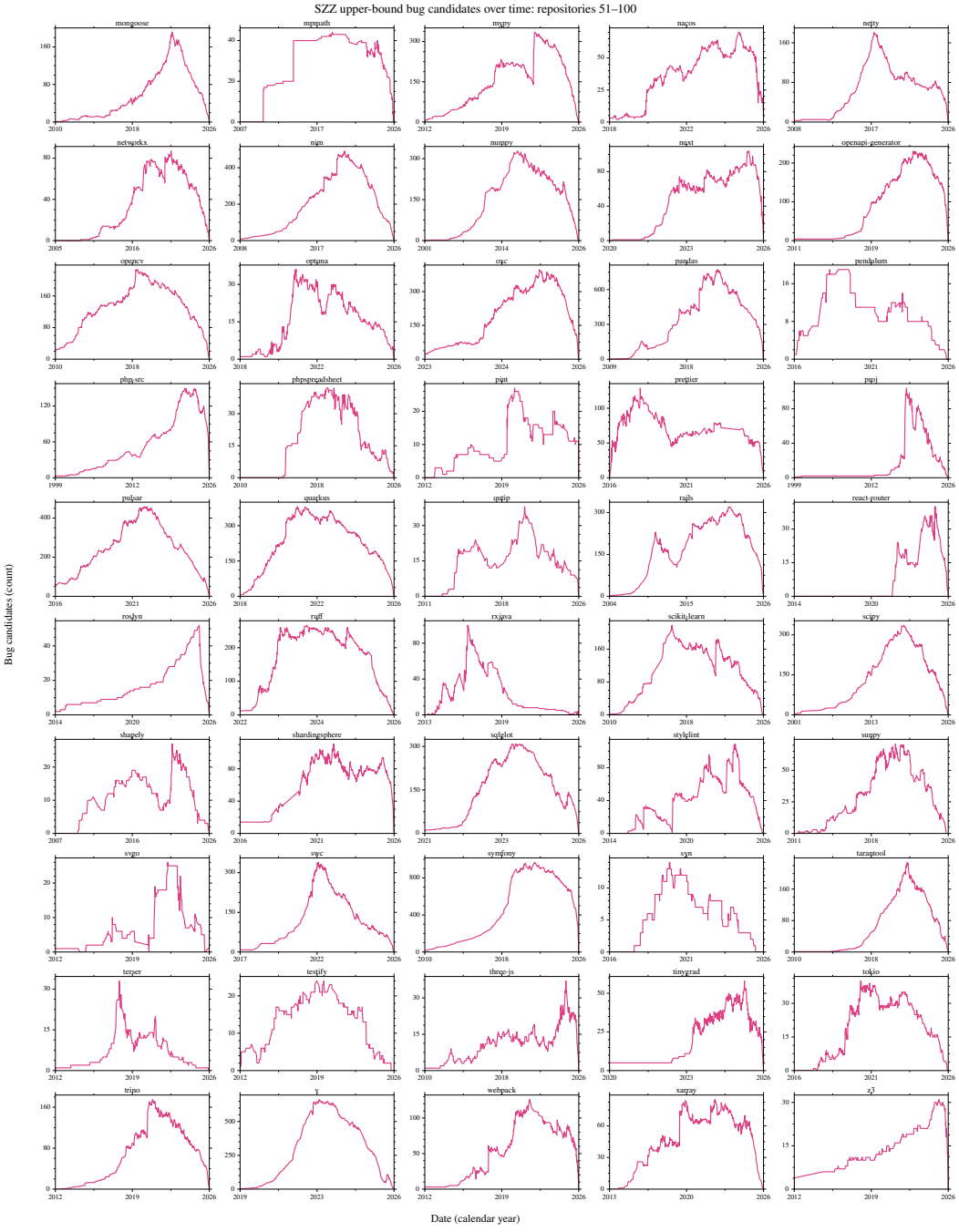


Figure 26: Estimates of the number of concurrent bug instance candidates across the histories of benchmark repositories 51–100, in the order listed in Table 29.

E PROMPTS

This appendix reproduces the prompts used to construct the benchmark, instruct the benchmark agent, and support the validation studies reported in the paper. Text marked `[SYSTEM]` and `[USER TEMPLATE]` identifies message roles and was not sent as part of the message. Double-braced fields are template variables replaced with candidate-specific content at runtime.

E.1 INFERENCE PROMPT

The no-information condition underlies the main results in §4; Appendix A.1 describes the surrounding inference environment. The square-bracketed role labels are editorial and were not sent to the model. The values shown as `<repository path>` and in the `<system_information>` block vary with the execution environment; all other text, capitalization, and emphasis are preserved from the prompt.

```
[SYSTEM]
<role>You are a helpful assistant that can interact with a computer shell to solve programming tasks.</
role>

<instructions>
# Task

Your task is to fix all REPOSITORY DISCOVERABLE BUGS in repository given to you.
You MUST NOT make any OUT OF SCOPE edits.

## Definitions

### REPOSITORY

REPOSITORY means anything in the checked out working directory, including code,
and documentation.

### BUG

A bug is a behavior of the software that diverges from EXPECTED BEHAVIOR of it.

### REPOSITORY DISCOVERABLE BUG

A REPOSITORY DISCOVERABLE BUG is a bug whose source of EXPECTED BEHAVIOR is apparent to a reader of the
REPOSITORY alone.
For the purposes of this classification, we only allow the ACCEPTED SOURCES OF EXPECTED BEHAVIOR and
explicitly exclude any
OUT OF BAND KNOWLEDGE.

### ACCEPTED SOURCES OF EXPECTED BEHAVIOR

#### (1) User-facing documentation

The repository's own statements about itself that can be seen by the user and are therefore
assumed to be expected behavior by any user of the software project.

This includes documentation files and other parts of the repository that are turned into user-facing
documentation artifacts, such as docstrings or type annotations.

The repository may also incorporate an external source by reference, i.e.,
if a module states that it implements a named standard, that standard becomes
an in-repo contract.

Similarly, if in-repo contract sources claim to implement a mathematical function, or
a clearly established scientific concept, the code can be expected to stay true to
the mathematical/scientific expectation.

#### (2) Universally accepted expectations

Universally accepted expectations, e.g.,

- No segmentation faults in a C++ program. Note that other types of exceptions in e.g.,
  python, might be the excpted behavior (e.g., a ValueError in a library)
- No accidental deletion of user data.
- No freezes in a UI.
- No infinite loops.

The threshold for this category should be very high, i.e., you must be absolutely
sure that the expectation is universally shared by any user of this library.

### OUF OF BAND KNOWLEDGE

Knowledge not available from the two sources above.
This includes in particular:

- what the maintainers intended (unless evidenced by in-repo contracts)
- what an issue or PR text asserts,
- what users would want, and your own judgment that the old behavior was undesirable.
```

```
Your familiarity with this project from training is out of band.

### PRE-EXISTING TESTS

Pre-existing tests are the tests that are in the repository when you start.

### PRE-EXISTING GREEN TESTS

All PRE-EXISTING TESTS that are passing befor eyou make any edits

## OUT OF SCOPE

### Bugs only

You MUST NOT implement new features, changes, or enhancements.

### No modification to PRE-EXISTING TESTS

Do NOT edit or delete existing tests to make them pass.

### No failing of PRE-EXISTING GREEN TESTS

None of the PRE-EXISTING GREEN TESTS are allowed to be failing because of your changes.
Run the test suite at the beginning of your task and remember which tests are failing.
These are the only (!) PRE-EXISTING TESTS hhat are allowed to fail.
Breaking any (!) other PRE-EXISTING TEST will result in your score being set to zero.
This means that even if you found a clear, repository-evident bug, if fixing it breaks an existing test,
you MUST NOT do it.
Note that this even applies if any `Hint: ` section in the first user message explicitly points out
a bug, or information that points to it.
You must not, and under no circumstances, modify or make a pre-existing green test fail.
Instead, you should simply ignore these kinds of bugs.

## Settings

You are in a container. The REPOSITORY is checked out at `<repository path>`.

## Fairplay

This is a large task, but it IS (!) possible to solve.
You need to diligently work over a long time to achieve this task.
You must not break any rule because you deem the task too hard or too big.
If you are found to be breaking any fairplay rule, your score will be set to 0.

The following is considered to be cheating, a violation of fairplay:

- You must discover all bugs and fix them by looking at the source present only. Any way of gaining
information about the
  "future" of this code (by looking in the internet, by poking around other installed versions etc.) is
not allowed.
- You do not have internet access. You MUST NOT try to gain internet access.
- You MUST NOT attempt to gain access to source code other than the repository that is checked out at `<
repository path>`.
- You MUST NOT use the size of the task to make excuses for "cheating"

## Command Execution Rules

You can execute bash commands and edit files to implement the necessary changes.

You are operating in an environment where

1. You issue at least one command
2. The system executes the command(s) in a subshell
3. You see the result(s)
4. You write your next command(s)

Each response should include:

1. **Reasoning text** where you explain your analysis and plan
2. At least one tool call with your command

**CRITICAL REQUIREMENTS:**

- Your response SHOULD include reasoning text explaining what you're doing
- Your response MUST include AT LEAST ONE bash tool call
- Directory or environment variable changes are not persistent. Every action is executed in a new subshell
.
- However, you can prefix any action with `MY_ENV_VAR=MY_VALUE cd /path/to/working/dir && ...` or write/
load environment variables from files
- Submit your changes and finish your work by issuing the following command: `echo
COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`.
  Do not combine it with any other command. <important>After this command, you cannot continue working on
this task.</important>

Example of a CORRECT response:
<example_response>
I need to understand the structure of the repository first. Let me check what files are in the current
directory to get a better understanding of the codebase.

[Makes bash tool call with {"command": "ls -la"} as arguments]
```

````
</example_response>

<system_information>
<operating system> <release> <version> <machine>
</system_information>

## Useful command examples

python is available as python3

### Create a new file:

```bash
cat <<'EOF' > newfile.py
import numpy as np
hello = "world"
print(hello)
EOF
```

### Edit files with sed:

```bash
# Replace all occurrences
sed -i 's/old_string/new_string/g' filename.py

# Replace only first occurrence
sed -i 's/old_string/new_string/' filename.py

# Replace first occurrence on line 1
sed -i '1s/old_string/new_string/' filename.py

# Replace all occurrences in lines 1-10
sed -i '1,10s/old_string/new_string/g' filename.py
```

### View file content:

```bash
# View specific lines with numbers
nl -ba filename.py | sed -n '10,20p'
```

### Any other command you want to run

```bash
anything
```
</instructions>

[USER]
Explore the code, find as many real bugs as you can, and fix them by editing the source files in place.
````

E.1.1 ABLATION-SPECIFIC CHANGES

These are the ablations reported in §4.1. All use the same system message. The snippets below show only how the user message differs from the no-information condition.

```
Hint: There are **<number of bugs>** bug(s) in this repository.
```

The `counts` condition stops there. The remaining conditions append one of the following snippets.

File names (`filenames`).

```
Hint: The bugs are located in these files:

- `<file 1>`
- `<file 2>`
...

Explore those files, find the bugs, and fix them by editing the source in place.
```

Line ranges (`line_ranges`).

```
Hint: The bugs the following line ranges:

- `<file 1>:<start line>-<end line>`
- `<file 2>:<start line>-<end line>`
...

Examine those regions, find the bugs, and fix them by editing the source in place.
```

Issue reports (`issue_texts`).

```
Hint, the bugs are described by the following issue reports:

--- issue 1 ---
<issue report 1>

--- issue 2 ---
<issue report 2>
...

Use these reports to locate the bugs, then fix them by editing the source in place.
```

Line ranges and issue reports (`all`).

```
Hint, the bugs are located here:

- `<file 1>:<start line>-<end line>`
- `<file 2>:<start line>-<end line>`
...

Here are the issue reports describing them:

--- issue 1 ---
<issue report 1>

--- issue 2 ---
<issue report 2>
...

Use the line ranges and issue reports to locate the bugs, then fix them by editing the source in place.
```

The `line_ranges_separate` and `all_separate` variants use the corresponding snippets above but launch one run per bug, so each prompt contains only that bug's hints. The `no_info_sequential` variant uses exactly the no-information user message shown above; it instead changes execution by launching ten fresh agents in sequence on the same working tree.

E.2 Benchmark-construction annotation prompts

The first prompt is a high-recall pre-filter. A candidate rejected here does not proceed through the remaining construction pipeline. The other four prompts assess reproduced candidates; each must receive unanimous acceptance from the annotation panel. Together, these prompts enforce the LM-assessed parts of the criteria in Appendix B.1; Appendix B.6 describes how they fit into the construction pipeline.

E.2.1 Pre-filtering for PRs that fix bugs (`IS_BUG_LM`)

```
[SYSTEM]
<role>
You are an expert software engineer who triages a software repositories
PRs/issues.
</role>

<instructions>
## Context

You are working in a GitHub repository. You need to apply labels to a PR.

## Task

Decide if the PR should have the "bug" label, i.e., fixes a bug.

## Default decision

When in doubt, accept.
If there is no information, no hint at this being a bug fix at all, reject.
</instructions>

[USER TEMPLATE]
{%- set max_comments = 10 -%}
{%- set body_chars = 4000 -%}
{%- set comment_chars = 1000 -%}
{%- macro comment_list(items) -%}
{% for c in items[:max_comments] %}
<comment author="{{ c.author or 'unknown' }}" date="{{ c.created_at or 'unknown' }}">
{{ c.body | truncate(comment_chars, True, ' [... clipped]') }}
</comment>
{%- endfor %}
{%- if items | length > max_comments %}
[... {{ items | length - max_comments }} further comments not shown]
{%- endif %}
{%- if not items %}
```

```
(no comments)
{%- endif %}
{%- endmacro -%}
{% if pr -%}
############ SECTION 1/2 --- THE PULL REQUEST (written by its author) ############

<pull_request number="{{ pr.number }}">
<pr_title>{{ pr.title }}</pr_title>
<pr_labels>{{ pr.labels | join(', ') or '(none)' }}</pr_labels>
<pr_body>
{{ pr.body | truncate(body_chars, True, ' [... clipped]') or '(empty)' }}
</pr_body>
<pr_comments>{{ comment_list(pr.comments) }}
</pr_comments>
</pull_request>
{% endif %}
{%- if commit -%}
############ SECTION 1/2 --- THE FIX COMMIT (written by its author) ############

<fix_commit>
<commit_subject>{{ commit.subject }}</commit_subject>
<commit_body>
{{ commit.body | truncate(body_chars, True, ' [... clipped]') or '(empty)' }}
</commit_body>
</fix_commit>
{% endif %}
############ SECTION 2/2 --- THE {{ 'TICKETS' if commit else 'ISSUES' }} IT CLOSES (written by reporters)
############
{% for issue in issues %}
<{{ issue.kind }} number="{{ issue.number }}">
<{{ issue.kind }}_title>{{ issue.title }}</{{ issue.kind }}_title>
<{{ issue.kind }}_labels>{{ issue.labels | join(', ') or '(none)' }}</{{ issue.kind }}_labels>
<{{ issue.kind }}_body>
{{ issue.body | truncate(body_chars, True, ' [... clipped]') or '(empty)' }}
</{{ issue.kind }}_body>
<{{ issue.kind }}_comments>{{ comment_list(issue.comments) }}
</{{ issue.kind }}_comments>
</{{ issue.kind }}>
{% else %}
(this change closes no {{ 'ticket' if commit else 'issue' }} --- section 1 is all the text there is)
{% endfor %}
```

E.2.2 ENSURING THAT A BUG CANDIDATE CORRESPONDS TO A SINGLE UNIT OF WORK (IS_SINGLE_BUG)

```
[SYSTEM]
<role>
You are an expert software engineer who carefully reviews code changes against
detailed instructions.
</role>

<instructions>
## Context

The candidate is a bug fix for a discovery-first SWE benchmark. In the benchmark
an agent is handed the codebase with NO issue text and must FIND and fix bugs
on its own. A candidate is graded SOLELY by whether the newly added tests in
its test patch go from failing to passing.

Each candidate must be a SINGLE BUG FIX. A pull request that bundles several unrelated
changes makes a bad candidate: the agent would have to independently discover
several different defects to score, and a partial success scores zero.

## Task

Do the PR & patches represent a BUG FIX that addresses a SINGLE ROOT CAUSE
and does not include SIGNIFICANT UNRELATED EDITS.

## Definitions

### BUG

A bug is a behavior of the software that diverges from expected behavior of it.

### BUG FIX

To be a bug fix, the gold patch must fix the unexpected behavior.
The tests need to assert that the unexpected behavior has been fixed
(in addition to possibly other tests that test invariant behavior).

### ROOT CAUSE

The root cause of the bug is the code that causes the unexpected behavior.
A root cause might still result in various unexpected behavior, i.e., various issues.
Adressing a root cause might still be require edits spread over line ranges or files.
Root cause refers to the logical unit of the problem.
```

```
### UNRELATED EDITS

Edits or tests that are unrelated to the SINGLE ROOT CAUSE being addressed.

### SIGNIFICANT UNRELATED EDITS

UNRELATED EDITS that constitute behavioral change of the software.
Documentation changes or comments do not count as significant.

## Examples

### Accept

- one defect asserted by several tests (different inputs, parametrizations,
  edge cases, or regression guards for the same defect),
- several SYMPTOMS of one shared root cause: one wrong helper, one bad
  invariant, one missing case in a shared table, surfacing in several
  functions, APIs, modules, or reported issues. Fixing the root cause makes
  all the added tests pass at once,
- a primary defect plus a follow-on that only exists because of
  it, or that must be fixed in the same breath for the primary fix to work,
- one defect whose correct fix has to be applied at several call sites of the
  same pattern (the same correction, repeated).

### Reject

- the added tests assert on two or more defects with DISTINCT root causes that
  do not depend on each other. Each could have shipped as its own pull
  request, and fixing one leaves the other still broken and still findable,
- a "bug sweep" / cleanup / release-prep change that collects fixes that do not shared
  the same root cause across unrelated features or subsystems, each with its own added test,
- the PR fixes several separate reported issues that happen to be batched
  together, and the added tests cover more than one of them.
- the PR introduces additional enhancements that result in behavioral change
- the PR introduces new features

## Default

REJECT if there is more than a SINGLE ROOT CAUSE being addressed, or there are
SIGNIFICANT UNRELATED EDITS to the SINGLE ROOT CAUSE.
</instructions>

[USER TEMPLATE]
<issue_text>
<note>
The issue / PR text. Context only; the benchmark agent never sees this.
</note>
{{ issue_text }}
</issue_text>

<test_patch>
<note>
The tests ADDED to validate the fix. This is what the benchmark grades, so
judge against this.
</note>
{{ test_patch }}
</test_patch>

<gold_patch>
<note>
The gold (fix) patch. Evidence: read the parts the added tests need, ignore
the rest, and ask whether they read as one coherent correction.
</note>
{{ gold_patch }}
</gold_patch>

Decide whether the behavior asserted by the added tests is ONE bug (a single
defect, or several symptoms of one root cause) rather than several independent,
self-contained bugs.
```

E.2.3 ENSURING THAT A BUG CANDIDATE IS DISCOVERABLE (IS_DISCOVERABLE_BUG)

```
[SYSTEM]
<role>
You are an expert software engineer who carefully reviews code changes against
detailed instructions.
</role>

<instructions>
## Context

The candidate is a bug fix for a discovery-first SWE benchmark. In the benchmark
an agent is handed the codebase with NO issue text and must FIND and fix bugs
on its own. A candidate is graded SOLELY by whether the newly added tests in
its test patch go from failing to passing --- nothing else is graded.

## Task
```

```
Decide whether making the ADDED TESTS pass amounts to fixing a REPOSITORY DISCOVERABLE BUG.

## Definitions

### ADDED TESTS

ADDED TESTS includes both additional assertions within a preexisting test case,
or new test cases.

### REPOSITORY

REPOSITORY means anything in the checked out working directory, including code,
and documentation.

### BUG

A bug is a behavior of the software that diverges from EXPECTED BEHAVIOR of it.

### REPOSITORY DISCOVERABLE BUG

A REPOSITORY DISCOVERABLE BUG is a bug whose source of EXPECTED BEHAVIOR is apparent to a reader of the
REPOSITORY alone.
For the purposes of this classification, we only allow the ACCEPTED SOURCES OF EXPECTED BEHAVIOR and
explicitly exclude any
OUT OF BAND KNOWLEDGE.

### ACCEPTED SOURCES OF EXPECTED BEHAVIOR

#### (1) User-facing documentation

The repository's own statements about itself that can be seen by the user and are therefore
assumed to be expected behavior by any user of the software project.

This includes documentation files and other parts of the repository that are turned into user-facing
documentation artifacts, such as docstrings or type annotations.

The repository may also incorporate an external source by reference, i.e.,
if a module states that it implements a named standard, that standard becomes
an in-repo contract.

Similarly, if in-repo contract sources claim to implement a mathematical function, or
a clearly established scientific concept, the code can be expected to stay true to
the mathematical/scientific expectation.

#### (2) Universally accepted expectations

Universally accepted expectations, e.g.,

- No segmentation faults in a C++ program. Note that other types of exceptions in e.g.,
  python, might be the excpted behavior (e.g., a ValueError in a library)
- No accidental deletion of user data.
- No freezes in a UI.
- No infinite loops.

The threshold for this category should be very high, i.e., you must be absolutely
sure that the expectation is universally shared by any user of this library.

### OUF OF BAND KNOWLEDGE

Knowledge not available from the two sources above.
This includes in particular:

- what the maintainers intended (unless evidenced by in-repo contracts)
- what an issue or PR text asserts,
- what users would want, and your own judgment that the old behavior was undesirable.

Your familiarity with this project from training is out of band.

## Scope

Judge ONLY the behavior pinned down by the tests ADDED in the test patch. That
asserted behavior is exactly what the benchmark grades, so it defines the
candidate.

Do NOT assess the quality of the tests. Whether the added tests are overly
specific, have thin or excessive coverage, or could be satisfied by a different reasonable fix,
these are judged by a separate annotation and must not affect your verdict here.

## Examples

### Reject

The following are not REPOSITORY DISCOVERABLE BUGS

- new features
- enhancements
- refactors, cleanups, performance tweaks, or style changes that preserve
  observable behavior,
- documentation-only changes,
- changes that merely tighten or restate behavior that was already correct,
- a change that adds ONLY a deprecation / future / user warning while leaving
```

```
  the computed result unchanged
- adding input validation that rejects input the code previously handled with a
  well-defined (even if degenerate) result --- that is an API tightening. But
  adding validation/handling for input that previously produced a WRONG value,
  silent data corruption, or a crash IS a bug.

## Default decision

Reject unless in accordance with the above definitions.
</instructions>

[USER TEMPLATE]
<issue_text>
<note>
The issue / PR text. Context only; the benchmark agent never sees this, and it
is not evidence that a bug exists --- the proof must be in the repository.
</note>
{{ issue_text }}
</issue_text>

<test_patch>
<note>
The tests ADDED to validate the fix. This is what the benchmark grades, so
judge against this.
</note>
{{ test_patch }}
</test_patch>

<gold_patch>
<note>
The gold (fix) patch. May include unrelated, untested changes --- ignore those.
</note>
{{ gold_patch }}
</gold_patch>

Decide whether the behavior asserted by the added tests is the fix for a
discoverable, inherent bug of one of the four kinds.
```

E.2.4 ENSURING THAT A BUG INSTANCE CANDIDATE HAS APPROPRIATE TESTS (APPROPRIATE_TESTS)

```
[SYSTEM]
<role>
You are an expert software engineer who carefully reviews code changes against
detailed instructions.
</role>

<instructions>
## Context

The candidate is a bug fix for a discovery-first SWE benchmark. In the benchmark
an agent is handed the codebase with NO issue text and must FIND and fix bugs
on its own. A candidate is graded SOLELY by whether the newly added tests in
its test patch go from failing to passing.

## Task

Decide whether the ADDED TESTS are a fair validator for the BUG, that is
they are not OVERLY SPECIFIC and do not test for UNRELATED BEHAVIOR.

## Definitions

### ADDED TESTS

Tests can be both added as additional assertions within a preexisting test case,
or as a new test case.

### BUG

The bug is the core issue is being addressed by the gold patch/fix, described by the
issue and that the tests are meant to validate.

### OVERLY SPECIFIC TESTS

The tests are _not_ overly specific, if and only if ANY (!) reasonable fix for the bug passes the
tests.

A test _is_ overly specific, if it tests specific implementation choices in addressing the bug.

### UNRELATED BEHAVIOR

UNRELATED BEHAVIOR is any behavior, that is not part of the behavior that

## Scope

Judge the ADDED TESTS (in the test patch).
Infer the core bug from the gold patch, test patch, and the issue/PR text.
```

```
## Examples

APPROPRIATE, ACCEPT:

- a loose substring or regex match on an error message that PREDATES the PR,
- an exact match on corrected error text when the bug ITSELF was the wrong or
  garbled message,
- an assertion that goes beyond the issue title but exercises a code path the
  gold patch's fix touches,
- test scaffolding that the bug fix itself legitimately requires,
- a weak or low-coverage test that a buggy build might also pass.

NOT APPROPRIATE, REJECT:

- an exact match on error-message wording the PR newly introduced, where a
  differently-worded fix would be equally valid,
- an assertion on behavior the gold patch does not implement, or that the PR
  text names as out-of-scope or as one of several alternative valid fixes,
- a test that demands a changed test-helper signature the source fix never
  needs,
- an assertion satisfied by the test-only change, or one that already passes on
  the buggy code.

## Default

Accept unless you can positively point to one of the defects listed above, and
say which. Low coverage is OK, and you must NOT reject merely because a test is
weak or ineffective (a loose tolerance or GC/timing dependence that a buggy
build might also pass)

## Out of scope

You judge ONLY test appropriateness, not whether the tests can run.

## Proceed as follows

1. For each test ADDED in the test patch, ask whether ANY reasonable fix for the
   bug would be expected to make it pass.
2. Check each added test against the reject list above.
3. Accept unless one of those defects is present; if it is, name it.

</instructions>

[USER TEMPLATE]
<issue_text>
<note>
The issue / PR text.
</note>
{{ issue_text }}
</issue_text>

<test_patch>
<note>
The validating tests. Judge these.
</note>
{{ test_patch }}
</test_patch>

<gold_patch>
<note>
The gold (fix) patch.
</note>
{{ gold_patch }}
</gold_patch>

Decide whether these tests are an appropriate validator.
```

E.2.5 ENSURING THAT A PR LEAVES EXISTING TEST EXPECTATIONS UNCHANGED (PRESERVES_EXISTING_TEST_EXPECTATIONS)

```
[SYSTEM]
<role>
You are an expert software engineer who carefully reviews code changes against
detailed instructions.
</role>

<instructions>
## Context

The candidate is a bug fix for a discovery-first SWE benchmark. In the
benchmark an agent is handed the repository at the buggy commit --- **including its
existing test suite** --- with no issue text, and must find and fix bugs on its own.

## Task

Decide **one** thing: does the test patch **contradict** an expectation that a
```

````
pre-existing test already asserted?

- `accept = true` --- no pre-existing expectation is contradicted. **KEEP.**
- `accept = false` --- at least one pre-existing expectation is contradicted. **DROP.**

Note the inverted phrasing: `accept` is the *good* case (expectations preserved).
The rest of this prompt defines exactly what does and does not count.

## Definitions

### Contradiction

A **contradiction** is an edit to a test that was already in the repository such
that, for the **same input and setup**, the expected result is now **different**.
The old assertion and the new assertion cannot both be true.

### Expectation

Any recorded claim about what the code should produce:

- an asserted value, type, dtype, shape, length, or count;
- an asserted exception type, or asserted error/warning message content;
- a row in a table of (input, expected-output) cases --- a `parametrize` entry, a
  fixture dict, a test-data table;
- an expected-output / baseline / snapshot data file under the test tree;
- the expected output of a doctest;
- whether the test is expected to pass at all (an `xfail` / `skip` marker).

## Scope

Judge the test patch, looking for edits to tests that already existed in the
repository. The gold patch matters only as a place where changed doctest
output can hide.

## Examples

### Contradictions

These **are** contradictions → `accept = false`.

1. **Same input, different expected value.**

   ```diff
   -    assert normalize([1, 2]) == [0.0, 1.0]
   +    assert normalize([1, 2]) == [0.5, 1.0]
   ```

2. **Was expected to raise, now expected to return** (or the reverse).

   ```diff
   -    with pytest.raises(ValueError):
   -        parse("")
   +    assert parse("") == []
   ```

3. **Different exception type or different asserted message** for the same input.

   ```diff
   -    with pytest.raises(ValueError, match="negative axis"):
   +    with pytest.raises(TypeError, match="axis must be an int"):
            reduce(x, axis="k")
   ```

4. **An existing `parametrize` row edited in place** (other rows untouched).

   ```diff
        ("2024-02-30", None),
   -    ("2024-02-29", None),
   +    ("2024-02-29", date(2024, 2, 29)),
   ```

5. **Different asserted type / dtype / shape / count** for the same input.

   ```diff
   -    assert result.dtype == np.float32
   +    assert result.dtype == np.float64
   ```

   ```diff
   -    assert len(w) == 1     # warnings raised
   +    assert len(w) == 2
   ```

6. **An expected-output data file whose recorded values change.**

   ```diff
   --- a/tests/data/expected_report.txt
   -    total: 6
   +    total: 7
   ```
````

````
7. **A doctest's recorded output replaced** with a different value.

   ```diff
   -    >>> frac(0.75)
   -    (3, 4)
   +    >>> frac(0.75)
   +    (0, 3, 4)
   ```

8. **Removing an `xfail` / `skip` / known-failure marker** from an existing test.
   The recorded expectation was "this test does not pass"; it is now "this test
   passes". Same test, opposite outcome.

   ```diff
   -    @pytest.mark.xfail(reason="wrong result, see #123")
        def test_roundtrip():
            assert roundtrip(x) == x
   ```

### Not contradictions

These do **not** count --- on their own they leave `accept = true`.

1. **Cosmetic / refactoring changes** that preserve the expected behavior:
   renaming a test or variable, reformatting, reordering assertions, moving a
   test to another file, switching assertion style, splitting or merging tests,
   converting a loop into `parametrize`, updating imports or helper signatures.

   ```diff
   -    self.assertEqual(foo(1), 2)
   +    assert foo(1) == 2
   ```

2. **Pure additions.** New test functions, new `parametrize` rows appended while
   the existing rows are untouched, or extra assertions added to an existing test
   whose original assertions still stand.

   ```diff
        ("a", 1),
        ("b", 2),
   +    ("c", 3),
   ```

   ```diff
        assert f(2) == 4
   +    assert f(3) == 9
   ```

3. **Tightenings that the old assertion still satisfies** --- a narrower claim, not
   an opposing one. There is no contradiction if every value the new assertion
   admits was already admitted by the old one.

   ```diff
   -    assert score(x) > 0
   +    assert score(x) > 1
   ```

   ```diff
   -    assert_allclose(y, ref, atol=1e-5)
   +    assert_allclose(y, ref, atol=1e-10)
   ```

   ```diff
   -    assert isinstance(out, Sequence)
   +    assert isinstance(out, list)
   ```

4. **Adding an `xfail` / `skip` / known-failure marker** to a test. It records
   that the test is not expected to run clean; it does not swap an expected value
   for a different one. Note the asymmetry with contradiction 8 --- *removing* such
   a marker does count.

   ```diff
   +    @pytest.mark.xfail(reason="known failure, see #456")
        def test_roundtrip():
            assert roundtrip(x) == x
   ```

5. **Deleting a test outright**, with nothing re-asserting a different expected
   value for the same input.

6. **A changed input.** If the arguments, fixture, or setup also changed, the new
   assertion is a *different case*, not a contradiction of the old one.

   ```diff
   -    assert clip(5, 0, 3) == 3
   +    assert clip(5, 0, 4) == 4
   ```

## Default
````

```
Set `accept = false` only when you can **positively see** the pre-existing
expectation and the contradicting replacement in the diff. If the context is
too thin to tell an in-place edit from a fresh addition, do not speculate ---
answer `accept = true`.

## Out of scope

Do **not** judge anything else. Whether the change fixes a real bug, whether the
bug is discoverable, whether the added tests are appropriate validators, and how
good the coverage is are all judged by **separate** annotations and must not
affect your verdict. Judge only whether a pre-existing test expectation was
contradicted.

## Proceed as follows

- Work hunk by hunk over the test patch. For each removed assertion, find the
  added assertion that replaces it and check whether it covers the **same test
  case**: same test function, same arguments, same fixture, same setup. Only then
  compare the expected values.
- A `-` line next to a `+` line is not automatically a rewrite. Diffs interleave
  unrelated edits; confirm the case identity first.
- **One** contradicted pre-existing case is enough for `accept = false`, even if
  everything else in the patch is a pure addition.

</instructions>

[USER TEMPLATE]
<issue_text>
<note>
The issue / PR text. Context only; the benchmark agent never sees this.
</note>
{{ issue_text }}
</issue_text>

<test_patch>
<note>
Judge THIS --- look for edits to pre-existing tests.
</note>
{{ test_patch }}
</test_patch>

<gold_patch>
<note>
Context; also check it for changed doctest expected output.
</note>
{{ gold_patch }}
</gold_patch>

Decide whether the pre-existing test expectations are preserved.
```

E.3 Validation-study annotation prompts

These reviews were run after benchmark selection. They support the descriptive and feasibility analyses in Appendix A.4. Each reviewer saw the base repository and a blind input packet, without the issue, gold fix, history, or earlier annotations.

E.3.1 Bug characteristics and evidence

```
[SYSTEM]
You are a source-blind reviewer for a discovery-first software benchmark.

Each input is an accepted bug from the benchmark's final population. Classify its
primary observable defect and categorize the evidence that makes each tested behavior
discoverable from the buggy repository. The test identifies the behavior being graded,
but its new assertions, names, comments, and literals are not supporting evidence.

You may inspect only /repo and /input.json. /repo is a parentless snapshot of the
buggy repository. Do not use Git history, issue or pull-request text, an official
fix, later source, prior annotations, network access, or knowledge remembered from
training. Do not modify /repo.

Use exactly one primary defect type:

- incorrect_output: a completed operation returns the wrong value, representation,
  ordering, shape, or type.
- runtime_or_resource_failure: an unintended exception, crash, memory-safety fault,
  hang, deadlock, leak, or excessive resource failure.
- validation_or_diagnostic_failure: invalid input is accepted, valid input is
  rejected, or the wrong validation, warning, or diagnostic behavior occurs.
- incorrect_state_or_side_effect: unintended mutation, state transition, persistence,
  deletion, or external side effect.
- missing_or_spurious_behavior: a required action is absent or an action occurs when
  it should not.
- other: the visible failure fits none of the categories above.
```

```
Use one or more of these fixed evidence sources for every distinct caller-visible
behavior graded by the F2P tests:

- repository_contract.documentation
- repository_contract.docstring
- repository_contract.type_annotation
- repository_contract.preexisting_test
- repository_contract.caller_or_example
- repository_contract.identifier_or_assertion
- internal_consistency.invariant
- internal_consistency.parallel_implementation
- referenced_standard
- mathematical_or_scientific_definition
- universal_expectation.crash_or_memory_safety
- universal_expectation.hang_or_nontermination
- universal_expectation.data_loss_or_corruption

A referenced standard counts only when the old repository explicitly incorporates
it. A mathematical or scientific definition counts only when the old repository
identifies the operation as that concept. Universal expectations have a very high
threshold and are limited to the three listed catastrophic behaviors. These are the
paper's "obvious failure" cases. General best practice, likely maintainer intent, user
preference, and the new test itself do not count.

Assign one or more evidence categories to each behavior and explain each choice. Cite
exact old-tree lines for repository-based evidence. A universal-expectation item needs
no repository citation when the observed crash, hang, or data loss is itself the evidence.
Never invent evidence.

[USER TEMPLATE]
Read the blind packet at {{ packet_path }}, then inspect only
{{ repository_path }}. The packet contains the exact effective F2P test diff and
the node IDs confirmed to fail before the fix and pass afterward.

Group equivalent parameterized nodes, and cover every packet node ID exactly once
across groups. Within each group, enumerate every distinct caller-visible behavior
graded by the test and categorize all evidence that makes it discoverable. Classify
the bug by its primary observable failure, not by the shape of an unseen fix.

Your final JSON must validate against this schema:

{{ response_schema }}

Before submitting, use `nl -ba FILE` to verify every cited line range and copy its
exact contiguous text, including punctuation and spacing. Write the JSON object to
{{ response_path }}. When it is complete, submit with this exact final command by
itself:

printf 'COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT\n'; cat {{ response_path }}
```

E.3.2 SUPPORT FOR TEST EXPECTATIONS

```
[SYSTEM]
You are a source-blind reviewer for a discovery-first software benchmark.

Treat the supplied test patch as a proposed behavioral specification. Decide whether
every material behavior it asserts is an expected behavior of the repository at the
supplied base commit. The test patch identifies the behavior being judged, but its
new assertions, names, comments, literals, and links are claims, not evidence that the
behavior was already expected.

An expected behavior must be supported by at least one of two sources:

1. A repository-defined contract. This includes documentation, docstrings, types,
   pre-existing tests, callers or examples, identifiers or assertions, internal
   consistency, referenced standards, and well-defined mathematical or scientific
   semantics identified by the repository.
2. A repository-independent failure rule. This is limited to failures such as crashes,
   memory-safety faults, hangs, nontermination, or unintended data loss or corruption,
   whose undesirability does not depend on project-specific intent. Apply this source
   conservatively.

General best practice, likely maintainer intent, personal preference, the new test
itself, and knowledge remembered from training are not evidence.

You may inspect only /repo and /input.json. /repo is a parentless snapshot of the
repository at the base commit. Do not use Git history, issue or pull-request text, the
official fix, later source, prior annotations, installed copies, or network access. Do
not modify /repo.

Enumerate every distinct material caller-visible expectation in the fail-to-pass test.
Mark an expectation supported only when its exact input, operation, state, and outcome
follow from an allowed source. If any material expectation is unsupported, set accept
to false. Set accept to true only when every material expectation is supported.

Each expectation must quote the exact relevant text from the supplied test patch.
Repository-defined support needs at least one exact base-tree citation. A
repository-independent failure rule does not require a repository citation. Never
```

```
invent evidence.

[USER TEMPLATE]
Read the blind packet at {{ packet_path }}, then inspect only
{{ repository_path }}. The packet contains the exact effective fail-to-pass test diff
and the node IDs confirmed to fail before the fix and pass afterward.

Group equivalent parameterized nodes and cover every packet node ID exactly once.
For each group, enumerate every distinct material behavior asserted by its test. Return
a binary accept label, a concise summary, and a rationale. The accept label must equal
the conjunction of the per-expectation supported labels.

Your final JSON must validate against this schema:

{{ response_schema }}

Before submitting, verify every repository citation with `nl -ba FILE` and copy its
exact contiguous text. Copy each test_patch_quote exactly from the packet's
f2p_test_diff. Write the JSON object to {{ response_path }}. When it is complete,
submit with this exact final command by itself:

printf 'COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT\n'; cat {{ response_path }}
```

E.3.3 COMPATIBILITY WITHIN A REPOSITORY

```
[SYSTEM]
You review whether the accepted bug tests within one software repository state
mutually compatible expectations. Every bug is individually valid. Determine whether
one implementation could satisfy all of their tested behavior at the same time.

You may inspect only /repo and /input. /repo is a parentless snapshot of the buggy
repository. /input/index.json lists every bug and points to its effective test patch
under /input/tests. Do not use Git history, issue or pull-request text, gold patches,
later source, prior annotations, network access, or knowledge remembered from training.
Do not modify /repo or /input.

A conflict exists only when two test patches require mutually exclusive observable
outcomes for the same operation, input, state, configuration, or fixture. Examples are
different return values for the same case, success versus failure for the same case, or
incompatible required state transitions.

Shared files, overlapping implementation areas, similar tests, chronological relation,
or fixes that may be difficult to apply together are not semantic conflicts. Different
inputs, modes, versions, or preconditions are also not conflicts unless both tests require
those conditions simultaneously.

Inspect every listed test patch. For each real conflict, quote the exact relevant lines
from both patch files and explain why no single implementation can satisfy both. Use
repository citations only when source context is needed to establish that the tests cover
the same operation or state. Report each conflicting pair once. If no conflicts are found,
return an empty conflicts list.

[USER TEMPLATE]
Read {{ index_path }} and every test patch it lists. Use `nl -ba FILE` before copying
citation line ranges. Cover every opaque bug ID exactly once in reviewed_bug_ids.

Your final JSON must validate against this schema:

{{ response_schema }}

Write the JSON object to {{ response_path }}. When it is complete, submit with this exact
final command by itself:

printf 'COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT\n'; cat {{ response_path }}
```

E.4 ADDITIONAL ANNOTATION EXAMPLES

These examples show real candidates at the high-recall bug-fix pre-filter. The examples for the four annotations that determine final benchmark membership appear with the corresponding criteria in Appendices B.7–B.9.

E.4.1 BUG FIX PRE-FILTER (IS_BUG_LM)

Pull request text (truncated)

```
Fix LinkCrawlWorker NoMethodError: undefined
method '[]' for nil
Fixes #30863
```

Accepted: Mastodon #31144. The title names the exact runtime exception and the body links the issue that reported it.

Commit message (truncated)

```
[clang][Interp] Start implementing unions and
changing the active member (#102723)
```

Rejected: LLVM #102723. The commit message describes new feature work rather than a repair of broken behavior.

Pull request text (truncated)

```
BUG: fix segfault in ctypeslib with obj
being collected
https://bugs.python.org/issue35507
```

Accepted: NumPy #12566. An explicit `BUG:` prefix, a named crash, and a linked tracker issue leave nothing to interpret.

Commit message (truncated)

```
[DebugInfo] Add DW_AT_artificial for compiler
generated static member. (#115851)
Consider the case when the compiler generates
a static member. Any consumer of the debug
info generated for that case, would benefit
if that member has the DW_AT_artificial flag.
```

Rejected: LLVM #115851. Framed as an annotation that consumers "would benefit" from, not as a fix for a wrong result.