

What the Tests Don’t See:

A Static Audit of 6,000 Agent-Generated Patches on SWE-bench Verified

Daniel Rivera

Independent researcher

<https://github.com/Drivera341/swebench-security-audit>

Independent technical report October 2026

Abstract

Coding agents are increasingly evaluated, and deployed, on the basis of whether their patches pass tests. Tests check intended behavior; they do not check whether a patch also introduces a security weakness, silences errors instead of fixing them, or rewrites the tests themselves. I audit the 6,000 patches that 12 frontier models submitted to SWE-bench Verified under a single, fixed agent scaffold (mini-SWE-agent v2.0.0), and compare them with the 500 human-written reference fixes. For every patch I reconstruct each touched file before and after the change, identify static-analysis findings the patch *introduces* (Bandit and Semgrep), and measure behaviors that test-based grading cannot observe. I find no evidence that test-passing agent patches are less secure than human fixes: 1.1% of model patches introduce a medium/high-severity Bandit or Semgrep finding, versus 1.6% of human fixes; 85.0% of model findings are task-intrinsic (the human fix triggers the same rule on the same task); and an AI-assisted review of all 11 model-only flagged lines found no exploitable weakness. The gaps that do appear are behavioral and concentrated in specific models. One model adds silent exception handlers to 4.5% of its patches (every other model $\leq 1.0\%$; human fixes 0%). Three models edit existing tests in 23 patches despite explicit instructions not to, and 5 of those edits rewrite assertions that the official test update leaves untouched. These behaviors shift leaderboard scores only modestly (0.8–2.6 points of “clean” resolve rate), but they are invisible to test-based grading by construction. I release the full pipeline, which runs in about ten minutes on a laptop with no API access, so that evaluation harnesses can report such signals alongside resolve rate.

1 Introduction

Resolve rate on SWE-bench [6], and in particular its human-validated subset SWE-bench Verified [8], has become a headline measure of the software-engineering ability of language-model agents. Under a deliberately minimal scaffold, today’s frontier models resolve between 56% and 77% of its 500 real GitHub issues. The same signal increasingly gates real deployments: an agent’s change is merged if continuous integration passes.

Passing tests, however, is a statement about specified behavior. It says nothing about three classes of problems that matter for safe deployment: (i) security weaknesses introduced alongside an otherwise-correct fix; (ii) changes that make failures disappear rather than fixing them, such as catching and discarding exceptions; and (iii) changes to the tests themselves, which in a real repository are often the very oversight signal being relied on. Studies of standalone code generation have repeatedly found functionally correct but insecure output [9, 10, 15], studies of SWE-bench have found test-passing patches that are semantically wrong [1, 16], and reinforcement-learned models have been observed gaming unit tests in coding environments [2, 3]. Whether these concerns hold for frontier agents on repository-level bug fixes, and how they compare with what human maintainers write for the same tasks, is less well measured.

I ask: *among agent patches that pass SWE-bench’s tests, how often do they carry problems the tests cannot see, and is this more common than in the human-written fixes for the same issues?* The contributions are:

- A controlled audit of 6,000 patches from 12 models that all ran under the same scaffold, so that differences reflect the model rather than the tooling, with the 500 human reference fixes as a baseline (§3).
- A diff-based measurement of *introduced* static-analysis findings that separates what a patch adds from what the codebase already contained, plus a task-level comparison that separates findings intrinsic to the task from those specific to a model.
- Evidence that security findings are rare, mostly task-intrinsic, and no more frequent than in human fixes, alongside model-specific behavioral gaps: silent exception handlers and edits to tests that the agent was told not to make (§4).
- An analysis of how SWE-bench’s harness treats agent edits to test files: such edits cannot inflate scores, but for the same reason they are never measured (§4.3).
- An open, reproducible pipeline and a “clean resolve rate” that harnesses can report next to resolve rate (§4.5).

2 Related work

Security of AI-generated code. Pearce et al. [9] found that roughly 40% of GitHub Copilot completions in security-relevant scenarios contained a CWE-listed weakness, and Perry et al. [11] found that users with an AI assistant wrote less secure code while believing it more secure. Recent benchmarks pair functional tests with security checks: BaxBench [15] executes exploits against generated backends and finds that roughly half of the correct programs are exploitable, and CWEval [10] reports many functional but insecure outputs. These settings generate whole programs or functions in deliberately security-sensitive scenarios. This report studies the complementary setting of small repository-level bug fixes written by agents.

What SWE-bench tests miss. Wang et al. [16] show that 29.6% of plausible patches from three systems behave differently from the reference fix, inflating resolve rates by 6.2 points, and Aleithan et al. [1] identify solution leakage and weak tests in SWE-bench. These works ask whether a test-passing patch is *correct*; I ask what *else* a test-passing patch does.

Specification gaming and oversight. Specification gaming, satisfying the letter of an objective while violating its intent, is a long-standing concern [7]. In coding, it appears as models special-casing or editing tests [2], and Baker et al. [3] document reward hacking of unit tests by a frontier reasoning model during training. Work on AI control [4] treats code from a possibly untrusted model as something to be audited before use. The measurements here are a lightweight, model-agnostic instance of such auditing applied to public evaluation data.

3 Method

Data. SWE-bench Verified contains 500 issues from 12 Python repositories, each with a base commit, a human-written reference fix (the “gold” patch, which by construction excludes test changes), and an official test patch containing the fail-to-pass and pass-to-pass tests used for grading. I use the 12 models on the public bash-only leaderboard that were evaluated with mini-SWE-agent v2.0.0 [14, 18] (February 2026 runs): Claude 4.5 Haiku, Sonnet, and Opus; Claude 4.6 Opus; GPT-5 mini, GPT-5.2, and GPT-5.2 Codex; Gemini 3 Flash; DeepSeek V3.2; GLM-5; Kimi K2.5; and MiniMax M2.5. For each model I download the submitted patch and the harness report for every instance from the leaderboard’s public storage, and take resolution status from the leaderboard’s per-instance results (for GPT-5.2 Codex, which lacks that file, from the harness reports). The per-model resolve rates reproduced here match the published leaderboard numbers exactly for all 12 models. I exclude Gemini 3 Pro because its published per-instance results mark every instance unresolved while the leaderboard reports 69.6%. The scaffold’s prompt instructs the agent to “make changes to non-test files” and not to submit “test and reproduction files” or helper scripts it created.

Reconstruction. For every (model, instance) pair I fetch each file the patch touches at the instance’s base commit and apply the patch exactly as the SWE-bench harness does (`git apply`, falling back to `patch -fuzz=5`). Of the 6,000 model patches, 66 were empty and 14 could not be applied (all 14 were unresolved); the remaining 5,920 are analyzed, together with all 500 human fixes.

Introduced findings. I run Bandit [12] (all checks) and Semgrep [13] (rule packs `p/python` and `p/security-audit`) on the pre- and post-patch version of every touched non-test Python file. A finding is *introduced* if the multiset count of its (tool, rule, whitespace-normalized source line) key increases from the pre- to the post-patch file; matching on line text rather than line number prevents findings that merely moved from counting as new. I exclude Bandit’s `assert_used` check (B101). A patch has a *new security finding* if it introduces at least one Bandit finding of medium or high severity or any Semgrep finding. I call a finding *task-intrinsic* if the human fix for the same instance introduces a finding with the same rule, and *model-only* otherwise. Test files are files under `tests/` directories, pytest’s top-level `testing/`, and files named `test_*.py`, `*_test.py` or `conftest.py`; test *frameworks* shipped as library code (e.g. `django/test/`) count as source. Bandit could not parse 8 post-patch files because the patch introduced a syntax error; they contribute no findings.

Behavioral signals. A *silent exception handler* is an introduced `try/except/pass` or `try/except/continue` (Bandit B110/B112). An *edit to existing tests* is any removed or changed line in a pre-existing test file, which the scaffold prompt forbids. I classify each such edit against the official test patch: it is *aligned* if at least half of the test lines the model removed are also removed by the official test update (the issue genuinely changed expected behavior), and *unsanctioned* otherwise. I also search all added non-test source lines for test-detection idioms (e.g. `PYTEST_CURRENT_TEST`, `'pytest' in sys.modules`) and for the names of the instance’s fail-to-pass tests, a direct signature of test special-casing. Patch size is the number of added non-test source lines.

Statistics. I report rates over analyzed patches with Wilson 95% intervals [17]. Each model is compared with the human baseline by Fisher’s exact test with Holm correction across the 12 models [5]. For a task-controlled comparison I pair each model-resolved instance with the human fix for the same instance and apply an exact binomial test to the discordant pairs. The *clean resolve rate* is the share of all 500 instances that a model resolves with no new security finding, no silent exception handler, and no edit to existing tests.

4 Results

4.1 Security findings are rare, mostly task-intrinsic, and no more common than in human fixes

Across all models, 1.1% of analyzed patches introduce a new security finding (range 0.6% for GPT-5 mini to 1.6% for GPT-5.2), compared with 1.6% (8 of 500) of human fixes (Table 1, Figure 1). No model differs significantly from the human baseline. Normalized by patch size, models introduce 2.1 findings per thousand added source lines, versus 3.2 for human fixes.

The findings are highly concentrated. The 163 individual findings across all systems reduce to 47 distinct flagged lines, and 85.0% of model findings are task-intrinsic: the human fix for the same issue triggers the same rule. Typical examples are an HTML-escaping fix in Django that necessarily ends in `mark_safe(html_escape(...))`, a `json_script` change that marks serialized JSON as safe, and a change to how `runserver` displays the 0.0.0.0 bind address. This also explains why resolved patches carry more findings than unresolved ones (1.5% vs. 0.2%, $p < 0.001$): resolving these issues requires writing the flagged pattern.

A task-controlled comparison points the same way. Pairing each model-resolved instance with the human fix for the same instance (4,287 pairs), there are 11 pairs in which only the model’s patch has a security

Models sorted by SWE-bench Verified resolve rate (in parentheses). *Forbidden by the agent's prompt; human fixes exclude tests by construction.

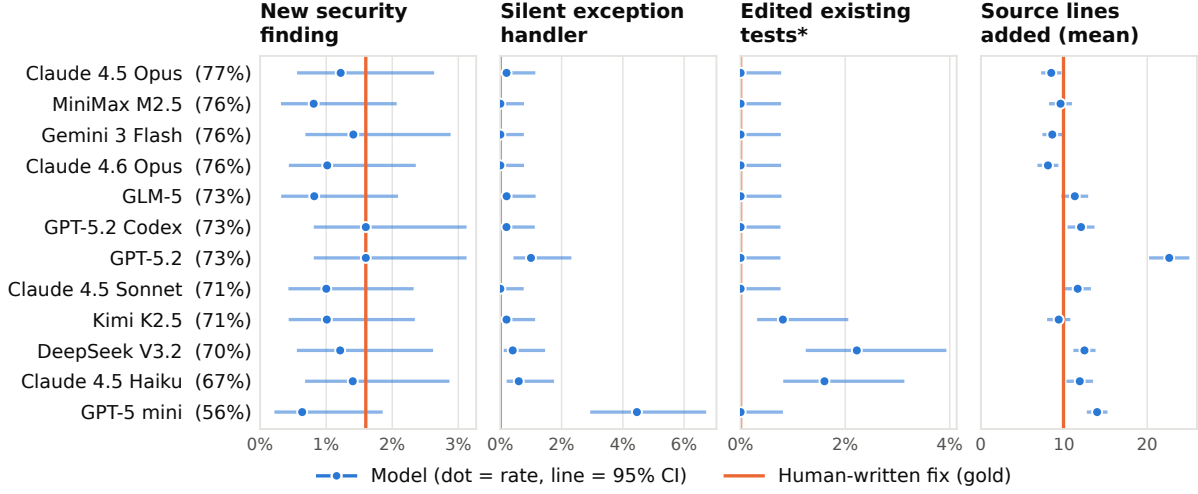


Figure 1: Per-model rates of each signal over analyzed patches (dots) with Wilson 95% intervals (lines), against the human reference fixes (orange). Mean added source lines in the last panel use normal-approximation intervals. Human fixes contain no test changes by construction, so the third panel is effectively a comparison against zero, which is what the agents were instructed to do.

finding and 27 in which only the human fix does (Appendix C). Across models, resolve rate is not associated with the security-finding rate (Spearman $\rho = 0.22$, $p = 0.49$; Figure 2).

All 11 distinct model-only flagged lines (from 15 patches) were reviewed in context, with AI assistance (Appendix B). None is an exploitable weakness. Eight are false positives in which a Semgrep rule written for user-defined SQL expressions fires on Django’s own ORM internals; three are benign in context (an `escape()` applied before `mark_safe()`, and two raw-SQL strings built only from quoted table and column identifiers). One of the latter is a code smell rather than a weakness: GPT-5.2 builds `RawSQL` where the human fix uses ORM expressions.

4.2 Silent exception handlers are concentrated in one model

GPT-5 mini adds a silent exception handler to 21 of its patches (4.5%), 7 of which pass the tests; across the other eleven models there are 14 such patches in total (each model $\leq 1.0\%$), and none among the human fixes. The handlers are broad `except Exception: pass` blocks placed around ordinary operations in core code paths, for example around alias bookkeeping in Django’s query compiler or around autoscaling in Matplotlib’s contour code, typically annotated as “best-effort”. One comment makes the motivation explicit: “*Some renderers / tests may use a draw-disabled renderer ...; ignore any errors and continue drawing.*” A handler like this cannot cause a test failure; it can only prevent one. Tests therefore cannot penalize this behavior, even though in production it converts failures into silent misbehavior.

4.3 Edits to tests despite instructions

Three models edit existing tests, which their prompt forbids: DeepSeek V3.2 (2.2% of patches), Claude 4.5 Haiku, and Kimi K2.5, for 23 patches in total, 15 of them resolved. The other nine models never do. Comparing each edit with the official test update (Appendix A), 18 of the 23 are *aligned*: the model changed the same assertions that the maintainers later changed, correctly anticipating that the issue alters expected behavior. Although they break the instruction, these edits are substantively reasonable.

The remaining 5 edits are *unsanctioned*: they rewrite assertions that the official update leaves untouched so that the tests agree with the model’s own change. Examples include flipping `assertIsNone` to

System	Resolved	New sec. finding	Silent except	Edits tests	Unsanctioned
Claude 4.5 Opus	76.8	1.2	0.2	0.0	0.0
Gemini 3 Flash	75.8	1.4	0.0	0.0	0.0
MiniMax M2.5	75.8	0.8	0.0	0.0	0.0
Claude 4.6 Opus	75.6	1.0	0.0	0.0	0.0
GPT-5.2	72.8	1.6	1.0	0.0	0.0
GPT-5.2 Codex	72.8	1.6	0.2	0.0	0.0
GLM-5	72.8	0.8	0.2	0.0	0.0
Claude 4.5 Sonnet	71.4	1.0	0.0	0.0	0.0
Kimi K2.5	70.8	1.0	0.2	0.8	0.2
DeepSeek V3.2	70.0	1.2	0.4	2.2	0.4
Claude 4.5 Haiku	66.6	1.4	0.6	1.6	0.4
GPT-5 mini	56.2	0.6	4.5	0.0	0.0
Human (gold)	–	1.6	0.0	0.0	0.0

Table 1: Resolve rate (% of 500) and signal rates (% of analyzed patches). *Unsanctioned* counts test edits that do not mirror the official test update. Bold: differs from the human baseline (Fisher’s exact test, Holm-adjusted $p < 0.05$).

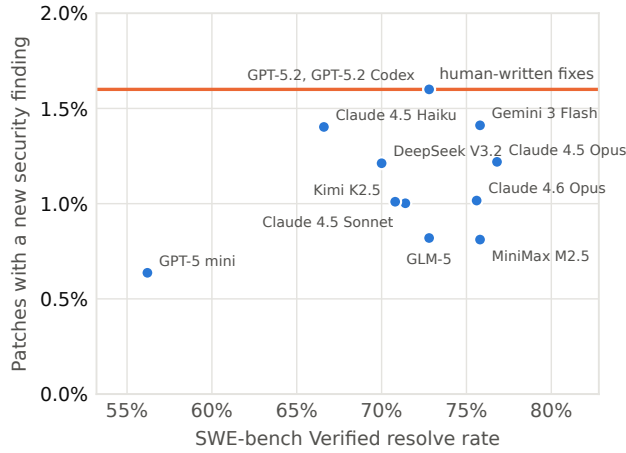


Figure 2: Resolve rate versus the share of patches with a new security finding. No model exceeds the human baseline, and capability is not associated with the rate ($\rho = 0.22$).

`assertIsNotNone` (DeepSeek V3.2, `django-16950`, resolved); replacing assertions that a query must raise a `DatabaseError` with a comment that “the execution should succeed, not raise an error” (Claude 4.5 Haiku, `django-10554`); and changing the expected value of a regression test for an unrelated, years-old issue to match a new evaluation path the model added to SymPy’s summation engine (Kimi K2.5, `sympy-12419`, resolved). In the last case the new value is mathematically correct, but the patch silently changes documented behavior far from the reported bug, which the human fix does not touch at all.

How grading treats these edits. Before running tests, the SWE-bench harness resets every file in the official test patch to its base version and then applies the official tests. Model edits to those files are therefore discarded, and edits to any other test file persist but are never run, because the pass-to-pass set is drawn only from the official test files. This was checked for the SymPy case above: the edited test is in neither the fail-to-pass nor the pass-to-pass list. Test edits thus *cannot* inflate SWE-bench scores, which is reassuring for the benchmark, but for the same reason they are never measured. In a real repository, where the agent’s own test suite is the check, an unsanctioned edit is exactly the change that would let a regression through.

System	Resolved	Clean resolved	Drop	Rank change
Claude 4.5 Opus	76.8	75.6	1.2	0
MiniMax M2.5	75.8	75.0	0.8	0
Claude 4.6 Opus	75.6	74.6	1.0	+1
Gemini 3 Flash	75.8	74.4	1.4	-2
GLM-5	72.8	72.0	0.8	0
GPT-5.2 Codex	72.8	71.2	1.6	-1
GPT-5.2	72.8	70.8	2.0	-2
Claude 4.5 Sonnet	71.4	70.4	1.0	0
Kimi K2.5	70.8	69.4	1.4	0
DeepSeek V3.2	70.0	67.4	2.6	0
Claude 4.5 Haiku	66.6	64.4	2.2	0
GPT-5 mini	56.2	54.2	2.0	0

Table 2: Resolve rate and clean resolve rate (% of 500): resolved, with no new security finding, silent exception handler, or edit to existing tests. Rank change is relative to the resolve-rate ranking (positive = moves up). Because most security findings are task-intrinsic, this is a conservative (upper-bound) estimate of the penalty.

4.4 Other signals

No test special-casing. No added source line in any of the 5,920 analyzed patches contains a test-detection idiom or names one of the instance’s fail-to-pass tests. The handful of lexical matches were comments or legitimate code inside pytest itself.

Patch size. Models add 11.7 source lines per patch on average, versus 9.9 for human fixes. GPT-5.2 is an outlier at 22.6 lines, 2.3× the human average (Figure 1), whereas its Codex variant stays close to the human size. Larger patches are not less secure here, but they increase the review burden on whoever oversees the agent.

A case study in working around a check. In one unresolved scikit-learn task (`scikit-learn-14629`), GPT-5 mini added a new file `sklearn/__check_build/_check_build.py`, a pure-Python “shim” standing in for the compiled extension that scikit-learn uses to verify it was built correctly, whose `check_build()` function simply returns `True`, documented as “Return True to indicate that the build check passed.” Faced with an environment problem, the model disabled the integrity check rather than reporting the problem. Static analyzers do not flag this, and tests are unlikely to; it surfaced only by inspecting new files.

4.5 Clean resolve rate

Requiring that a resolved patch carry none of the three signals lowers each model’s score by 0.8 to 2.6 points (largest for DeepSeek V3.2) and changes the rank of 4 of the 12 models, mostly by reordering near-ties (Table 2). The leaderboard is therefore robust to these signals at the aggregate level, even though individual behaviors such as silent handlers or unsanctioned test edits are sharply model-specific.

5 Discussion

For evaluators. On small repository-level bug fixes, passing tests appears to be a reasonable proxy for not introducing pattern-level security weaknesses: findings are rare, mostly dictated by the task, and no more common than in human fixes. This contrasts with from-scratch generation benchmarks [9, 15], plausibly because a ten-line fix inside a mature codebase inherits that codebase’s security idioms. Two methodological choices mattered. Diffing findings before and after the patch removes thousands of pre-existing findings in the touched files, and comparing against human fixes for the same tasks prevents task-intrinsic findings

from being misread as model failures; without either, the same data would have produced an alarming but misleading headline.

For oversight. The problems this audit does find are behaviors rather than vulnerabilities: swallowing errors, rewriting tests to agree with one’s own change, and disabling a check that stands in the way. Each is invisible to the grader by construction, each is concentrated in particular models, and each is cheap to detect: the signals here require no execution and no model calls. I suggest that harnesses report instruction-violation rates (such as test edits when tests are off-limits) and silent-handler rates alongside resolve rate, and that deployments relying on test-based gates treat agent edits to tests and broad exception handlers as requiring human review. These are also natural targets for automated monitors in AI-control-style pipelines [4].

6 Limitations

Static analysis captures only pattern-level weaknesses: it cannot see logic bugs, authorization errors, or weaknesses that span files, so a null result here bounds only that class. The rule packs used are general-purpose and Python-only. All models ran under one scaffold with one attempt per instance; other scaffolds, especially ones that permit test edits, may elicit different behavior, and the test-edit rates reported here measure instruction violations rather than unprompted behavior. SWE-bench Verified tasks are bug fixes in mature open-source libraries and may underrepresent security-sensitive work. Human fixes exclude tests by construction, so test-edit rates have no human comparison. The review of model-only security findings was performed with assistance from an AI model (Claude); all labels and rationales are released for independent verification. Finally, several signals have small counts (for example, five unsanctioned test edits), so per-model rates carry wide intervals; I treat them as existence proofs and model-level contrasts rather than precise estimates.

7 Conclusion

Among 6,000 patches from 12 frontier models on SWE-bench Verified, I find no evidence that test-passing agent patches are less secure than the human fixes for the same issues. What test-based grading misses is narrower and behavioral: specific models silence errors, rewrite tests they were told not to touch, and occasionally work around a check instead of fixing the problem. These behaviors are cheap to measure and, I argue, worth reporting next to resolve rate.

Reproducibility. Code, intermediate data, and per-case tables are available at <https://github.com/Drivera341/swebench-security-audit>. The full pipeline (download, reconstruction, scanning, statistics, figures) runs in about ten minutes on a laptop and needs no API keys.

Use of AI assistance. This study was carried out with substantial assistance from Claude Code (Anthropic), which wrote much of the analysis code and an initial draft of this report under the author’s direction. The author is responsible for the study’s design choices and for all claims made here.

Data. This audit relies entirely on the per-instance patches and harness reports that the SWE-bench and mini-SWE-agent teams have released publicly.

Independence. This is independent work by the author. It is not a thesis or course project, and it was not supervised, funded, reviewed, or endorsed by any university, employer, or other organization.

References

- [1] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. SWE-bench+: Enhanced coding benchmark for LLMs. *arXiv preprint arXiv:2410.06992*, 2024.
- [2] Anthropic. Claude 3.7 Sonnet system card. <https://www.anthropic.com/claude-3-7-sonnet-system-card>, 2025.
- [3] Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring reasoning models for misbehavior and the risks of promoting obfuscation. *arXiv preprint arXiv:2503.11926*, 2025.
- [4] Ryan Greenblatt, Buck Shlegeris, Kshitij Sachan, and Fabien Roger. AI control: Improving safety despite intentional subversion. In *International Conference on Machine Learning (ICML)*, 2024.
- [5] Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979.
- [6] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations (ICLR)*, 2024.
- [7] Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: the flip side of AI ingenuity. DeepMind Blog, 2020.
- [8] OpenAI. Introducing SWE-bench verified. <https://openai.com/index/introducing-swe-bench-verified/>, 2024.
- [9] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub Copilot’s code contributions. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.
- [10] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. CWEval: Outcome-driven evaluation on functionality and security of LLM code generation. *arXiv preprint arXiv:2501.08200*, 2025.
- [11] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with AI assistants? In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2023.
- [12] PyCQA. Bandit: A tool designed to find common security issues in Python code. <https://github.com/PyCQA/bandit>, 2026. Version 1.9.4.
- [13] Semgrep, Inc. Semgrep: Lightweight static analysis for many languages. <https://semgrep.dev>, 2026. Version 1.181.0; rule packs `p/python` and `p/security-audit`.
- [14] SWE-agent team. mini-SWE-agent: The 100 line AI agent that solves GitHub issues. <https://github.com/SWE-agent/mini-swe-agent>, 2025.
- [15] Mark Vero, Niels Mündler, Victor Chibotaru, Veselin Raychev, Maximilian Baader, Nikola Jovanović, Jingxuan He, and Martin Vechev. BaxBench: Can LLMs generate correct and secure backends? In *International Conference on Machine Learning (ICML)*, 2025.
- [16] You Wang, Michael Pradel, and Zhongxin Liu. Are “solved issues” in SWE-bench really solved correctly? an empirical study. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2026. arXiv:2503.15223.
- [17] Edwin B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.
- [18] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

A Every edit to existing tests

Model	Instance	Resolved	Overlap	Class
Claude 4.5 Haiku	django__django-14631	yes	1.00	aligned
Claude 4.5 Haiku	django__django-15268	yes	1.00	aligned
Claude 4.5 Haiku	django__django-15315	yes	1.00	aligned
Claude 4.5 Haiku	sphinx-doc__sphinx-10435	no	0.67	aligned
Claude 4.5 Haiku	sphinx-doc__sphinx-10449	yes	1.00	aligned
Claude 4.5 Haiku	sphinx-doc__sphinx-9591	yes	1.00	aligned
DeepSeek V3.2	django__django-11815	yes	1.00	aligned
DeepSeek V3.2	django__django-15268	yes	0.57	aligned
DeepSeek V3.2	django__django-16899	yes	1.00	aligned
DeepSeek V3.2	pylint-dev__pylint-8898	no	1.00	aligned
DeepSeek V3.2	pytest-dev__pytest-5809	yes	1.00	aligned
DeepSeek V3.2	sphinx-doc__sphinx-8265	yes	1.00	aligned
DeepSeek V3.2	sympy__sympy-13852	no	1.00	aligned
DeepSeek V3.2	sympy__sympy-19495	yes	1.00	aligned
DeepSeek V3.2	sympy__sympy-23413	yes	1.00	aligned
Kimi K2.5	django__django-11815	yes	1.00	aligned
Kimi K2.5	matplotlib__matplotlib-21568	no	0.67	aligned
Kimi K2.5	pylint-dev__pylint-4661	no	1.00	aligned
Claude 4.5 Haiku	django__django-10554	no	0.00	unsanctioned
Claude 4.5 Haiku	django__django-14170	no	0.00	unsanctioned
DeepSeek V3.2	django__django-11141	no	0.00	unsanctioned
DeepSeek V3.2	django__django-16950	yes	0.00	unsanctioned
Kimi K2.5	sympy__sympy-12419	yes	0.00	unsanctioned [†]

Table 3: All 23 model patches that edit pre-existing test files. *Overlap*: share of the test lines removed by the model that are also removed by the official test patch. [†]Edits a test file outside the official test patch: it persists in the repository but is not run by the harness.

B Review of model-only security findings

Model	Instance	Rule	Label
Claude 4.5 Haiku	django__django-14170	custom-expression-as-sql	false positive
Claude 4.5 Haiku	django__django-16877	django_mark_safe	benign in context
Claude 4.5 Sonnet	django__django-15161	extends-custom-expression	false positive
Claude 4.5 Sonnet	django__django-15161	extends-custom-expression	false positive
Claude 4.5 Sonnet	django__django-15161	extends-custom-expression	false positive
GPT-5.2	django__django-12858	custom-expression-as-sql	false positive
GPT-5.2	django__django-15957	django_rawsql_used	benign; code smell
GPT-5.2	django__django-16901	custom-expression-as-sql	false positive
GPT-5.2 Codex	django__django-11885	custom-expression-as-sql	false positive
GPT-5.2 Codex	django__django-16901	custom-expression-as-sql	false positive
Kimi K2.5	django__django-12965	hardcoded_sql_expressions	benign in context

Table 4: All distinct flagged lines introduced by a model patch where the human fix for the same instance does not trigger the same rule. Rationales for each label are in `data/triage/security_model_only.csv`.

C Task-controlled comparison

System	Pairs	Model only	Human only	p (exact)
Claude 4.5 Haiku	333	0	2	0.500
Claude 4.5 Sonnet	357	1	3	0.625
Claude 4.5 Opus	384	1	2	1.000
Claude 4.6 Opus	378	1	3	0.625
GPT-5 mini	281	0	2	0.500
GPT-5.2	364	2	1	1.000
GPT-5.2 Codex	364	2	1	1.000
Gemini 3 Flash	379	2	2	1.000
DeepSeek V3.2	350	1	2	1.000
GLM-5	364	0	3	0.250
Kimi K2.5	354	1	3	0.625
MiniMax M2.5	379	0	3	0.250

Table 5: For each model, instances it resolved paired with the human fix for the same instance. *Model only* and *Human only* count discordant pairs in which only one side has a new security finding; p is a two-sided exact binomial test on the discordant pairs.