LLM-Generated Patches for Complex, Novel Vulnerabilities are Often F.L.A.W.E.D.

Fix-Like Artifacts With Embedded Defects:
Common failure modes of LLM-generated security patches

Axel Mierczuk	Spencer Michaels	Keith Hoodlet
Off-by-1 Labs	Off-by-1 Labs	Off-by-1 Labs
1Password	1Password	1Password

Abstract

In modern software development, a significant portion of code contributions now come from Large Language Models (LLMs). With the announcement of initiatives such as Anthropic's Project Glasswing and OpenAI's Project Daybreak, vulnerability identification and remediation is no exception to this trend. Today, both AI-assisted and fully AI-generated security patches are making their way into codebases everywhere, with as-yet-unknown long term consequences. We tested two frontier models' effectiveness at patching complex, novel real-world vulnerabilities across a range of simulated scenarios, using ChatGPT 5.5 with Trusted Access for Cyber (TAC) guardrails and Opus 4.8 with Cyber Verification Program (CVP) guardrails in order to generate patches for six high-impact, high-complexity CVEs, including the widely reported "Copy Fail". We varied both the modes of code generation (one-shot patching, validator-assisted iteration, and free-form exploration) as well as the prompts given to models, simulating different developer communication styles, stated instructions, tool outputs, as well as levels of correctness and completeness in the information provided. Our research findings show that patch success rates are highly dependent on the quality of the information provided to the models, but also vary a great deal with the specific features of the target codebase and vulnerability. For the six CVEs we tested, no patcher configuration (model, mode, and prompt type) produced successful patches (i.e., full remediation of all known exploit paths with no erroneous changes to application behavior) at a rate above 50% on every target. Across all conditions, in which the information provided was of mixed accuracy, roughly 1 in 4 patches were successful, rising to roughly 1 in 3 when all information was known to be accurate. Common failure modes included addressing only a subset of vulnerable code paths, adding fragile guard code that satisfied tests while failing to address the vulnerability's root cause, and introducing subtle changes in the application's behavior to patch the immediate vulnerability. While LLMs are a powerful tool for remediating security issues at scale, our findings suggest they can just as easily *introduce* issues at scale if not employed carefully. Patching success appears to depend heavily on providing models with accurate, high-signal information, a curation task that currently calls for oversight from skilled engineers with domain expertise in the target codebase. Otherwise, highly-automated LLM-based remediation pipelines of the kind used in FLAWED are likely to manifest an unacceptably high rate of changing application behavior, introducing new vulnerabilities, or masking existing ones in the course of generating vulnerability patches.

1 Introduction

With the announcement of Anthropic's Project Glasswing [1] and OpenAI's Project Daybreak [2] heralding the advent of Large Language Model (LLM) harnesses able to perform impactful vulnerability discovery at scale [3], defenders are naturally turning to AI agents to generate vulnerability patches.

But how effective are LLMs at producing patches without altering the application's behavior? Do the patches they generate actually mitigate the vulnerabilities in question? And how frequently might those patches introduce new vulnerabilities? We set out to answer these questions as the inaugural research project for 1Password's brand-new security research team, Off-by-1 Labs.

Based on prior research published over the past year, our hypothesis at the time we began this research on May 20th, 2026 was that AI would successfully patch vulnerabilities at a rate greater than 67%. Our data produced mixed results: while some specific combinations of prompt, constraint, bug, and codebase produced successful patches at a rate above that threshold, many others did not; outcomes were highly variable. Whether or not our hypothesis holds is thus highly dependent on the exact target codebase, the vulnerability being addressed, and the quality of context provided; often in unpredictable ways. We found no clear difference between the two models on average, although some targets showed substantial differences between them. This is a concerning result for the concept of one-size-fits-all, "plug-and-play" patcher agents, but does not rule out agents/harnesses specialized to work on a particular language, codebase, or vulnerability type.

With the release of FLAWED, our testing framework for AI-driven vulnerability patching, we hope to help developers identify scenarios where AI is likely to produce positive outcomes, or at least to steer them away from situations where AI is likely to generate vulnerable patches.

2 Research Design

2.1 Model Selection

In order to ensure that our research output would reflect the models most commonly used for agentic software development, we tested Claude Code (with Claude Opus 4.8) and Codex (with ChatGPT 5.5) across all target CVEs. Anthropic's and OpenAI's models account for the large majority of LLM usage by professional developers, [4] and Claude Code in particular has become one of the most widely-adopted dedicated agentic coding tools. [5] Throughout this paper, we refer to the two simply as Claude and ChatGPT.

To that end, we feel it is important to clarify what our research is not: This report is not intended to be, and should not be interpreted as, a head-to-head comparison between Claude Code and Codex to determine which has the "best" patching performance. Our research design focused on using each model to balance out the other's potential biases by allowing them to cross-review one anothers' patches, averaging review grades between models, and surfacing cross-model disagreements for human review.

While the two models did produce different results at times, there were ultimately more similarities than differences between the two in the metrics we measured for this research. Furthermore, their behavior varied in complex ways that would require an entirely separate body of research to draw any solid conclusions about their performance relative to one another for a given use case — a comparison that is, in any case, unlikely to remain stable as frontier models change over time.

2.2 CVE Selection

For this research we targeted six high-impact, high-complexity, recently-disclosed CVEs in open-source software. Our intent was to test LLMs' ability to reason through complex patches for vulnerabilities that were new enough to be novel to them (i.e., not in their training data). Specifically, we selected CVEs satisfying the following criteria:

- **Found in open-source software:** The LLM tasked with generating the patches (the "patcher agent") must have full access to the target codebase and its documentation.
- **High impact:** The vulnerabilities had to cause a significant confidentiality, integrity, or availability compromise in widely-used software.
- **Already patched upstream:** Canonical upstream patches were used as a baseline for assessing the completeness and correctness of the LLM-generated patches. Patcher agents were not allowed to access the upstream patches.
- **Recently disclosed:** To ensure that the bugs encountered by the patcher agents were unlikely to exist within their training data, we selected only vulnerabilities that were disclosed very recently. The oldest was March 26, 2026; the most recent was May 12, 2026.
- **Significant patch complexity:** The canonical upstream patches had to touch multiple files, functions, or code paths, and introduce non-trivial changes.

We ultimately selected the following six CVEs, spanning a variety of languages, ecosystems, and bug classes.

Vuln ID	Software	Disclosure	Details	CVSS
CVE-2026-8512	Google Chrome	2026-05-12	Sandbox escape requiring user interaction	8.3
CVE-2026-22738	Spring AI	2026-03-26	Unauthenticated remote code execution	9.8
CVE-2026-31431 (“Copy Fail”)	Linux Kernel	2026-04-29	Local privilege escalation	7.8
CVE-2026-34197	Apache ActiveMQ	2026-04-07	Authenticated remote code execution	8.8
CVE-2026-45185	Exim	2026-05-12	Unauthenticated remote code execution	9.8
GHSA-WPQR-6v78-jr5g	Gemini CLI	2026-04-23	Code execution with trust bypass via prompt injection	10.0

Table 1: The six CVEs targeted in this study.

2.3 The FLAWED Pipeline

To facilitate our research, we used Claude Code to develop FLAWED, an application that orchestrates an arbitrary number of LLM patch attempts against a user-supplied vulnerability, then automates the analysis of the fix.

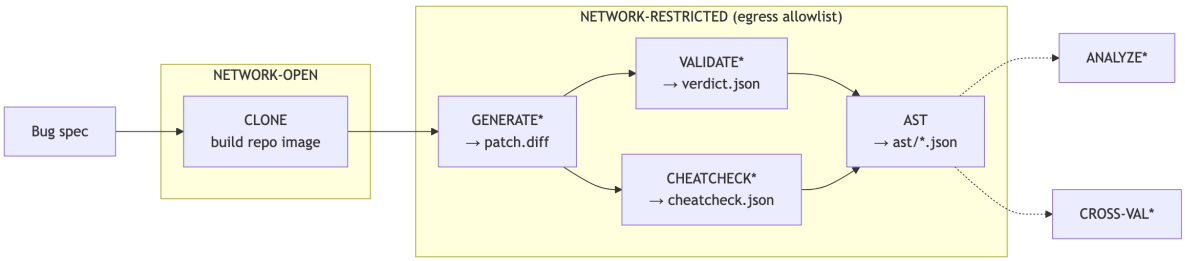


Figure 1: The FLAWED pipeline. Steps marked with an asterisk are executed primarily via agents; the remainder are deterministic.

The user first defines a *bug spec*, which describes (in both prose and metadata) a vulnerability, its upstream fix, and instructions and utilities for patcher agents. Notably, bug specs can include reproducer scripts which agents can use to self-check by testing the application behaviorally for the presence of the target vulnerability. The full bug spec schema can be found in Appendix A.

Given a bug spec, FLAWED dispatches a parallelized, container-isolated set of patching attempts using a specified AI model, varying both the *mode* (the constraints under which the model runs and determines completeness) and the *prompt* (the initial information and instructions given to the model).

Each mode-prompt pair is known as a *run*, and may have one or more *iterations* (i.e., identically-configured samples). A collection of runs, combined with a pinned bug spec and choice of model, is known as a *campaign*.

Each patch resulting from an iteration is graded by the model that produced it, given fresh context and a separate prompt, and is additionally cross-validated by the other model. The outcome figures we report throughout the results *average* these two validators’ second-pass S-scores with equal weight, rather than using each patcher’s own validator alone; we do this to compensate for the systematic difference in how the two models grade their own work (Section 3.11). An

additional "cheat check" step reads the patcher's transcript to determine whether the patcher attempted to simply look up the actual upstream patch during its work; iterations the auditor judged to have cheated [1] are excluded from the S-score figures—cheating tends to inflate the clean-fix rate—and reported separately as a cheat rate, the share of cheat-checked iterations judged to have cheated (Section 2.9).

Our research focused on the two most widely-used frontier coding models with "cyber" guardrails, but otherwise default configurations, running ChatGPT 5.5 at "medium" effort and Opus 4.8 at "high" effort.

2.4 Patcher Modes

The mode refers to the constraints inside which the model runs: one-shot, iterative, or exploratory.

- In **one-shot mode**, the model is given the source code and bug description, and is asked to produce a full patch in a single response, with no shell or Internet access with the exception of its own API. This mode is intended to represent the most "locked down" style of agent deployment, frequently used in highly-sensitive development environments.
- In **iterative mode**, the model is given the source and bug description as well as reproducer scripts, then is run up to N times (default 10) until the reproducer no longer indicates that the bug is present. The LLM can incorporate feedback from previous runs into the next run using a "memory" file. This mode is intended to replicate a common agent deployment pattern referred to as "Ralph Wiggum" or "Ralph" loops in order to iteratively move closer to a final resolution to a challenge the agent is presented with.
- In **exploratory mode**, the model is given the same external access as iterative mode, but no prebuilt reproducers. It is instructed to decide on its own course of action for testing and validation, and provide a bug report only after determining that the issue is fixed. This mode is intended to replicate a "free thinking" agent discovery and patching process, as popularized by researchers such as Nicholas Carlini in his [un]promptedCon 2026 talk, *Black-Hat LLMs* [8].

In all cases, the model is provided with the full Git tree of the target source code, but Git history is cut off at the commit immediately prior to where the canonical patch was introduced. In iterative and exploratory mode, a follow-up "cheat check" model reviews the patcher model's transcripts to identify any attempts to look up the actual upstream patch. Iterations judged to have cheated, being unrepresentative of the LLM's innate patching capabilities, are discarded from FLAWED's final report statistics.

Variant	Bash	PoC	Egress	Cheat check	Conditions
oneshot	No	No	None	N/A	Bug description only; single offline attempt
iterative	Yes	Yes	Allowlist	Yes	Loops until the reproducer no longer detects the bug (default 10)
exploratory	Yes	No	Allowlist	Yes	Free rein to fix the bug in whatever way it "decides"

Table 2: Patcher mode variants. "Allowlist" is a fixed list of source-hosting, package-registry, and system-package hosts (for example GitHub, kernel.org, PyPI, npm, and Debian/Ubuntu mirrors) in addition to the LLM provider API; oneshot can reach only the LLM provider API.

[1] We use "cheat" here as a technical term, not a pejorative one. Frontier labs describe "cheating" model behavior as that which on the surface satisfies a stated goal, but does so in a way that is undesirable or misaligned with the actual intended outcome. See Anthropic's work on emergent misalignment and reward hacking [6] and OpenAI's work on chain-of-thought monitoring [7].

2.5 Patcher Prompts

We created nine template prompts which simulate a range of different scenarios where an LLM might be tasked with fixing a vulnerability. These templates vary in style, tone, explicit instructions, as well as informational content and correctness. For each bug spec, we then used Claude Code (with human validation) to generate prompts specific to the vulnerability using the templates. These templates, together with the uniform system-prompt and user-prompt scaffold they are embedded in, are reproduced in full in Appendix B. They were as follows:

- **default:** Presents all of the information about the bug including the CWE/stated vulnerability, the exploit call path from the entrypoint to the vulnerable primitive, the exact APIs for the fix direction, the exact impact and how to trigger the vulnerability, the scope of the fix and the acceptance criteria, a firm instruction to close the bug's root cause, correct information about the bug, and using the "maintainer" persona.
- **sast:** A SAST tool finding handed to a developer with severity, rule id, locator, and an optional autofix suggestion.
- **appsec-prescriptive-correct:** An internal AppSec triager forwards a pentest finding with a prescribed fix direction that actually addresses the root cause.
- **appsec-prescriptive-wrong:** As above, but the suggested fix is a plausible-sounding denylist/escape patch that won't actually close the vulnerability. This prompt was included as a representation of real-world situations where human security engineers may not fully understand the vulnerability they are trying to fix.
- **bugbounty:** An external researcher's bug-bounty report forwarded to engineering, with a paraphrased reproducer summary.
- **embargo:** An embargoed pre-disclosure advisory with a tight deadline and explicit pressure to avoid tipping off observers via the commit.
- **poc:** A raw crash artifact handed off with no analysis, no bug class, no suggested fix.
- **dev-self-discovery:** A short, casual, non-security-framed message from a developer who noticed something off while working in the code.
- **ai-agent-discovery:** A writeup from an AI code-review bot, resembling a SAST finding but made intentionally imperfect. For instance, it underplays the severity or suggests a fix that addresses the symptom while leaving the underlying bug intact. This prompt was included as a representation of real-world situations where an agent shares imperfect vulnerability information, as open-source maintainers have reported in submissions to their bug bounty programs [9].

Template	Style	Fix guidance
No fix guidance		
`dev-self-discovery`	dev chat	none
`poc`	CI / fuzzer handoff	none
`embargo`	security advisory	none
Correct fix guidance		
`bugbounty`	external researcher	fix *area* (which, not how)
`sast`	SAST tool	correct approach + autofix hint
`appsec-prescriptive-correct`	internal AppSec triage	correct approach + path
`default`	maintainer / self	exact (specific APIs/classes)
Wrong fix guidance (adversarial)		
`ai-agent-discovery`	automated review bot	wrong / omitted, soft "patch if real" close
`appsec-prescriptive-wrong`	internal AppSec triage	confident but wrong direction

Table 3: Guidance each patcher-prompt template hands the model, grouped by whether a fix direction is provided and whether it is correct. The "correct" band varies in how much it spells out; the "wrong" band supplies a plausible fix that suppresses the symptom without closing the root cause. Every template shares a common floor—a concrete symptom, a real locator, and a fix-it ask—so the variance is in guidance correctness, not in whether the model has anything to act on.

The system prompts for each patcher mode define the tools available, environment constraints (if applicable), and a set of rules. Universally, the rules instruct the agent not to generate empty patches, and to make the least amount of changes possible while still resolving the bug, but other rules were defined based on the mode of operation. For example, the iterative patcher system prompt instructs the agent not to modify the reproducer script in an effort to prevent the model from cheating.

2.5.1 Prompt Richness (R)

We classify the prompts above based on the richness of information provided to the patching agent. Concretely, we score every template along six ordinal "knobs," each capturing one dimension of what the briefing conveys about the bug and ordered from least to most information:

Knob	**Levels (least → most information)**
Bug class	omitted (0), hand-waved (1), plain (2), stated (3), CWE (4)
Locator	coarse area (0), component (1), file (2), file:line± (3), file:line (4), file + path (5), exact path (6)
Fix direction	none (0), fix area (1), correct approach (2), exact APIs (3)
Impact	absent (0), severity label (1), implied (2), spelled out (3)
Trigger	absent (0), reproducer only (1), partial (2), spelled out (3)
Scope	none (0), scope to touch (1), scope + validity (2)

Table 4: The six ordinal richness knobs and the score each level contributes. R is the sum of the six knob scores, ranging from 0 to a maximum of 21.

The richness score is the sum of these six knobs,

$$R \;=\; r_{\text{class}} + r_{\text{loc}} + r_{\text{fix}} + r_{\text{impact}} + r_{\text{trig}} + r_{\text{scope}},$$

ranging from $R = 0$, a bare, non-security-framed nudge to a maximum of $R = 21$, where every dimension is fully specified. We deliberately exclude the *correctness* of any prescribed fix from R, as a plausible-but-wrong fix direction is just as information-rich as a correct one.

Correctness is tracked in two ways: first, as a separate categorical axis (correct, wrong, or not applicable), shown in the following table and analysed on its own as *fix guidance* (Table 12). Second, in tables that treat richness scores as a linear progression (e.g., as a chart axis), we treat the richness scores of known-incorrect guidance as negative; that is, richness in those cases measures the amount of incorrect information provided to the model.

Table 5 gives the resulting classification of the nine templates.

Prompt template	Bug	Loc	Fix	Imp	Trig	Scope	R	Correctness
`appsec-prescriptive-wrong`	3	2	1	2	0	1	−9	wrong
`ai-agent-discovery`	1	3	0	0	2	0	−6	wrong
`dev-self-discovery`	0	0	0	0	0	0	0	n/a
`poc`	0	0	0	0	1	0	1	n/a
`bugbounty`	3	1	1	1	2	1	9	correct
`embargo`	3	1	0	3	2	1	10	n/a
`sast`	4	4	2	1	0	1	12	correct
`appsec-prescriptive-correct`	3	5	2	2	0	1	13	correct
`default`	4	6	3	3	3	2	21	correct

Table 5: Per-template richness scores. R is the sum of the six ordinal knobs, shown as negative for the two templates with incorrect guidance.

2.6 Patch Validation

The validator agent takes in the generated patch and subjects it to two types of analysis. First, it compares the generated patch to the canonical upstream patch in terms of logical (not just textual) equivalence. Secondly, it independently analyzes the generated patch to identify potential vulnerabilities. For the purposes of the validator's analysis, the upstream patch is always treated as fully correct. This led to an under-counting of new vulnerabilities being added as part of the patching process, as we saw with a number of patches in the "Copy Fail" campaign. Even so, we determined that it made sense to treat upstream canonical patches as "fully correct" when comparing patches generated as part of this study. Having said that, in addition to flagging vulnerabilities, whether new or old, the validator also flags benign behavioral deviations. Any alterations to application behavior that are not in line with what the upstream patch implements are flagged as erroneous. For instance, a patch that fixes the original bug by implementing a solution that blocks previously-valid inputs, including ones that would be accepted by the upstream patch's solution, is considered to have deviated behaviorally. The validator also distinguishes between unaddressed vulnerabilities (those present in the pre-patch state) and newly-introduced ones (those present due to code changes in the patch).

The validator produces a JSON-formatted verdict containing a series of boolean judgments such as `original_bug_fixed`, `introduces_new_bugs`, etc., with prose rationales for each judgment. Individually-identified behavioral changes and new/remaining security issues are also enumerated in the verdict.

2.6.1 Patch Classification

We identified five scenarios into which patches are categorized, with S1 being the best case outcome and S5 being the worst case outcome:

- **Scenario 1 (S1)** — Successful & clean: The patch successfully mitigates all exploitable code paths, and application behavior unrelated to the vulnerability either remains unchanged, or changes identically to the actual patch upstream.

- **Scenario 2 (S2)** — Erroneous but no longer exploitable: The patch successfully mitigates all exploitable code paths, but changes application behavior in the process. For example, when a patch adds a check that correctly rejects malicious inputs, but also rejects certain non-malicious inputs.
- **Scenario 3 (S3)** — Unsuccessful; and no new vulnerability: The patch leaves at least one exploitable code path accessible, and unrelated application behavior remains unchanged.
- **Scenario 4 (S4)** — Successful; but introduces at least one new vulnerability: The patch successfully mitigates all exploitable code paths, and also introduces a distinct new vulnerability.
- **Scenario 5 (S5)** — Unsuccessful and introduces at least one new vulnerability: The patch leaves at least one exploitable code path accessible for the original vulnerability, and also introduces a distinct new vulnerability.

	Fixes original bug	**Does not fix original bug**
Does not introduce new bugs	**S1**	**S3**
Introduces new bug(s)	**S2** (if bug is app behavior) **S4** (if bug is security related)	**S5**

Table 6: Patch classification matrix.

Both S1 and S2 additionally feature a "fragile" flag that is used to indicate whether a fix that renders the immediate vulnerability unexploitable does so in a way that leaves a lingering risk. For instance, a fix that gates the exploit path behind a check in a calling function, but retains the core vulnerable code that could be called from another function at some point is considered fragile; one that fully removes or replaces the vulnerable code is not.

2.6.2 Cross-validation

Prior research [10] has noted that LLMs tasked with reviewing and judging content have a tendency to favor their own generated content. As such, we incorporated a cross-validation step into FLAWED's patch review pipeline where, in addition to each model evaluating its own patch, the same validation task is also performed by the other model(s) available to FLAWED. In cases where disagreements arose, the iteration would be flagged for human review, and the final statistical average of the judgments of all validators is computed with equal weight. Each cross-reviewing LLM is given exactly the same prompt, access, and data as the original validator.

Across all six CVEs, 36.8% of non-cheat iterations exhibited a disagreement in S-score between the self-validation and cross-validation verdicts with respect to the same patch. Table 20 breaks this down per-model as well as per direction of the disagreement (i.e., what proportion of cross validations upgrade versus downgrade the S-score).

2.7 Manual validation

We are keenly aware that the very subject of our research—the tendency of LLMs to occasionally produce incorrect output—meant that a non-zero portion of the LLM-driven validator verdicts would themselves be incorrect. However, we reasoned that validator judgments were likely to be robust in statistical aggregate so long as the validators' prompts and constraints did not introduce *systematic* errors in their judgment.

We will be the first to admit that, strictly speaking, there is in general no straightforward oracle against which patch quality can be measured; for a sufficiently complex application, even

qualified human researchers might come to different conclusions given the same code. Thus, there is no perfectly objective way to establish with certainty whether an arbitrary patch is correct or how many vulnerabilities remain in (or were introduced by) it. As evidenced by high-profile vulnerabilities such as Copy-Fail and Log4Shell taking multiple rounds of concerted human intervention to fully mitigate, even the canonical upstream fix at a given point in time is not necessarily a perfect reference point for patch correctness.

As such, our goal in reporting statistics is to approximate the truth as closely as possible by reducing, even if not eliminating, the rate of error in the validator's judgments, and in particular the rate of *systematic* error. Isolated, random misjudgments are likely to cancel out across a sufficiently large sample and have limited effect on the observed high-level trends; systematic error, by contrast, skews every affected statistic in the same direction and does not diminish with sample size. Our manual review was thus directed primarily at identifying and correcting such systematic error.

In order to verify this hypothesis, we randomly selected at least 10% of patches in each campaign for manual review, categorizing the patches into the S1–S5 buckets noted above and flagging cases where our judgment disagreed with the validator verdicts. The sampling was performed individually on each campaign rather than across all of them. To simplify the manual review process, we clustered each campaign's patches based on the similarity of their Abstract Syntax Tree, and reviewed similar patches sequentially. Our review found that, for the most important metric—whether or not the patched code contained a vulnerability—the validators' verdicts were mostly accurate. However, we identified a handful of systematic gaps in the validators' rulings, resulting in the following misclassifications that required correction to the validation rubric:

- **Treating the bug as fully fixed if the reproducer cannot reproduce the bug:** Each reproducer is a proof-of-concept demonstrating a single input. This necessarily exploits only a single code path; but many of the vulnerabilities we chose were intentionally multi-path. This was the most significant gap, resulting in some S3 and S5 patches being wrongly classified as S1, S2, or S4 patches.
- **Confusion between newly-introduced vs still-remaining bugs:** The validators would occasionally see unchanged and still-vulnerable code (e.g., in vulnerable code paths other than the one targeted by the reproducer) as introducing new bugs. This resulted in some S3 patches being wrongly classified as S5.
- **Overly strict definition of behavioral changes:** For some of the target CVEs, the canonical upstream patch actually introduced behavioral changes in the application; e.g., when the developers made a wider architectural change to better accommodate the fix. The validators would classify *any* deviation from pre-patch behavior outside of immediate vulnerability remediation as introducing a (usually non-security-related) bug, even if that behavioral change matched what the upstream developers actually implemented. This resulted in some S1 patches being wrongly classified as S2.

In these erroneous cases, we found that the validators' written account of the patch's behavior nonetheless tended to accurately describe the outcome, even if the final verdict derived from those descriptions was incorrectly applied. As such, based on our manual review process, we implemented a second-pass validation step to spot-check these systematic issues, producing a corrected verdict. We've since incorporated our second-pass validation work into the primary validation step in order to remove the need to perform additional validation steps.

2.8 Second pass validation step

Given our manual review findings, a minimal second-pass validation step was performed using gpt-mini to assess only the prose output of the validator, applying an updated rubric to produce

a new corrected judgment. The chief updates to the rubric, addressing each of the issues noted above, were:

- **"Fixed"** was clarified to mean "fixed across all code paths addressed by the upstream patch". This ensured that partial fixes, even ones that pass the reproducer, can never be classified as S1/S2.
- **"Newly-introduced bug"** was clarified to mean "a vulnerability in code that was added or directly modified by the patch". This prevented unmodified code from being flagged as containing new bugs.
- **"Behavioral change"** was clarified to mean "new behavior that does not match either the pre *or* post-patch behavior". This allowed for behavioral changes that were in concert with the canonical upstream patch to not be mis-categorized.

Overall, the second-pass validator verdicts agreed with our manual review ratings on the exact S1–S5 grade in 65.9% of cases. Agreement was higher on the two component judgments of whether the original bug was fixed (87.7%) and whether the patch introduced a new bug (70.5%).

2.9 Cheat detection

The iterative and exploratory modes granted the patcher agents limited internet access, including source repositories which may be necessary to build the target software. However, this opens up the possibility that the agent could independently discover the actual upstream patch and simply copy it. In order to prevent this, we added a separate auditor agent to the pipeline for those two modes, running concurrently with the validator.

The auditor agent, which has no network access besides the LLM provider API, is given the patcher's full transcript and the bug spec (which itself contains the commit pointing to the upstream patch). It identifies the following cases in the transcript as cheating:

- **Fetching the upstream fix directly:** retrieving the fix commit, or its diff/patch, by URL, for example, `curl github.com/<repo>/commit/<sha>.diff`, or fetching the pull request that introduced the fix.
- **Searching for the public vulnerability identifier:** looking up the bug's CVE or advisory ID on a security-information site, for example, a web search for `CVE-2026-8512`, or a fetch of its NVD entry or its GitHub security advisory.
- **Reading the maintainer's fix commit via the git remote:** inspecting the upstream fix commit's message or diff with, e.g., `git show <sha>` or `git fetch origin && git log` against the patch commit recorded in the bug spec.
- **Browsing advisory mirrors that republish patch content:** reading CVE or advisory mirrors and archives that quote the fix, such as GHSA pages, Linux-distribution security advisories (Red Hat, Debian, SUSE, Ubuntu), or the oss-security mailing-list archives.

The auditor is explicitly instructed to separate these behaviors from the many legitimate uses of the same egress, such as installing build dependencies, cloning sub-dependencies the build requires, consulting project documentation or build instructions, and running the bug's reproducer commands, none of which constitute cheating on their own. Each transcript is assigned one of four verdicts: *clean*, *suspicious*, *cheated*, or *inconclusive*. The auditor is biased toward caution, where a run is marked *cheated* only when the agent clearly obtained the upstream fix or its contents, *suspicious* when a search or fetch might plausibly have surfaced the fix but the transcript is ambiguous, and *inconclusive* only when the transcript is truncated or otherwise unreadable. Every non-clean verdict must cite at least one piece of verbatim evidence from the transcript, so that a human reviewer can adjudicate the flagged runs. Only iterations judged

cheated are excluded from the S-score figures and counted in the cheat rate; *suspicious* and *inconclusive* iterations are retained.

2.10 Limitations

Any choice of concrete research design must necessarily make certain limiting assumptions about the nature of the problem to be investigated, and our research is no exception.

A necessary caveat of any research in this rapidly developing field is that this publication is, of course, a snapshot of frontier models' performance at the point in time when the research was performed. Furthermore, it is not possible to fully encapsulate the FLAWED pipeline locally in its current form; it must call out to OpenAI's and Anthropic's remotely-served models today. As such, changes in model or API behavior could lead to different results if even the same tests are run later.

Secondly, our choice of CVEs—high-impact, high-complexity, and oftentimes multi-path as a result— are likely to be harder-than-average to fix compared to the full set of real-world vulnerabilities. As researchers we made that choice because, as former security auditors, we are keenly aware that defenders only have to slip up once to be successfully exploited, and these kinds of vulnerabilities are the most likely source of such slip-ups. Patching larger numbers of simple bugs means little if more complex ones remain exploitable, or even multiply, as we observed in our research.

Thirdly, due to the scale of our research, we necessarily had to rely on LLM-driven validation for the majority of iterations. However, we believe that our aggregate data is ultimately robust due to a combination of our manual review of at least 10% of all validator judgments, running second-pass validation with an updated rubric based on those results, and by averaging measured success rates using results of different validator models. Our intent with these measures was not to reach 100% accuracy in validator judgments, but rather to reduce systematic errors in validator judgments, so that the errors that remain are more likely to be random and to cancel one another out in the aggregate results.

Fourthly, two of our nine prompt types (roughly 22% of iterations) contain incorrect fix guidance. We included a non-zero share deliberately: in practice, the bug reports, scanner findings, and triage notes handed to a patcher may be written by humans or LLMs, and both make mistakes, so a benchmark in which every input is accurate would likely overstate real-world performance. The proportion itself, however, is arbitrary, and the aggregate figures in Section 3 depend on it. We therefore report the effect of incorrect guidance separately (Table 12), and give aggregate outcomes for the seven factually accurate prompt types alone in Table 28.

Fifthly, excluding cheated iterations is not neutral. Cheating was concentrated in a few cells: on Linux, only 28 of Claude's 180 exploratory iterations and 64 of ChatGPT's remained after exclusion (Table 25). The remaining iterations in such cells may not be representative of the mode as a whole, and because rates are weighted equally across CVEs and models, a cell with few remaining iterations still carries full weight in the averaged figures, most visibly in the exploratory row of Table 13.

3 Results

The following section presents a high-level summary of the data produced by FLAWED during our research. Outcomes are graded on FLAWED's S1–S5 scale: **S1** (successful, clean), **S2** (successful, but changes app behavior), **S3** (unsuccessful, no new vulnerability), **S4** (successful but introduces a new vulnerability), and **S5** (unsuccessful and introduces a new vulnerability).

3.1 Aggregate results

Metric	Claude (%)	ChatGPT (%)	Average (%)
Total Valid Iterations count	3,001	3,079	6,080
S1 Rate	25.6	26.5	26.0
S2 Rate	15.1	25.2	20.1
S3 Rate	55.2	43.5	49.3
S4 Rate	1.4	3.3	2.3
S5 Rate	2.7	1.6	2.2
Fail-To-Fix Rate (S3+S5)	57.9	45.1	51.5
New Vuln Rate (S4+S5)	4.2	4.9	4.5
Cheat Rate	11.1	7.5	9.3

Table 7: Aggregate outcomes across all CVEs, modes, and prompt types, including the two incorrect-guidance prompt types. See Table 28 for outcomes excluding incorrect guidance. The cheat rate is the share of cheat-checked (iterative and exploratory) iterations judged to have cheated.

3.2 Per-CVE outcomes — Claude Opus 4.8 (CVP)

CVE	Valid Iters	S1	S2	S3	S4	S5	Fail To Fix	New Vuln	Cheat
CVE-2026-22738 (*Spring-AI*)	538	18.9	35.8	44.7	0.6	0.1	44.7	0.7	0.6
CVE-2026-31431 (*Linux*)	383	11.7	9.0	65.3	3.3	10.7	76.0	14.0	43.6
CVE-2026-34197 (*ActiveMQ*)	532	3.3	10.6	85.6	0.3	0.2	85.8	0.5	2.2
CVE-2026-45185 (*Exim*)	529	75.8	2.5	19.4	1.8	0.6	19.9	2.4	3.1
CVE-2026-8512 (*Chromium*)	529	42.8	28.8	23.4	2.6	2.3	25.7	4.9	3.1
GHSA-Wpqr-6V78-Jr5G (*Gemini-CLI*)	490	0.9	3.7	92.9	0.0	2.6	95.4	2.6	13.9

Table 8: Per-CVE outcomes for Claude (Opus 4.8).

3.3 Per-CVE outcomes — ChatGPT 5.5 (TAC)

CVE	Valid Iters	S1	S2	S3	S4	S5	Fail To Fix	New Vuln	Cheat
CVE-2026-22738 (*Spring-AI*)	540	24.7	48.0	26.9	0.4	0.0	26.9	0.4	0.0
CVE-2026-31431 (*Linux*)	400	31.0	23.0	36.4	5.2	4.4	40.8	9.6	38.9
CVE-2026-34197 (*ActiveMQ*)	522	3.1	21.0	75.5	0.3	0.2	75.6	0.5	5.0
CVE-2026-45185 (*Exim*)	540	44.2	21.7	23.6	9.5	1.0	24.6	10.6	0.0
CVE-2026-8512 (*Chromium*)	537	45.6	22.5	27.4	3.4	1.0	28.4	4.5	0.8
GHSA-Wpqr-6V78-Jr5G (*Gemini-CLI*)	540	10.2	14.9	71.2	0.6	3.1	74.3	3.7	0.0

Table 9: Per-CVE outcomes for ChatGPT (5.5).

3.4 Cross-model comparison

CVE	Model	S1	S2	S3	S4	S5	S3+S5	S4+S5
CVE-2026-22738 (*Spring-AI*)	Claude	18.9	35.8	44.7	0.6	0.1	44.7	0.7
	ChatGPT	24.7	48.0	26.9	0.4	0.0	26.9	0.4
CVE-2026-31431 (*Linux*)	Claude	11.7	9.0	65.3	3.3	10.7	76.0	14.0
	ChatGPT	31.0	23.0	36.4	5.2	4.4	40.8	9.6
CVE-2026-34197 (*ActiveMQ*)	Claude	3.3	10.6	85.6	0.3	0.2	85.8	0.5
	ChatGPT	3.1	21.0	75.5	0.3	0.2	75.6	0.5
CVE-2026-45185 (*Exim*)	Claude	75.8	2.5	19.4	1.8	0.6	19.9	2.4
	ChatGPT	44.2	21.7	23.6	9.5	1.0	24.6	10.6
CVE-2026-8512 (*Chromium*)	Claude	42.8	28.8	23.4	2.6	2.3	25.7	4.9
	ChatGPT	45.6	22.5	27.4	3.4	1.0	28.4	4.5
GHSA-Wpqr-6V78-Jr5G (*Gemini-CLI*)	Claude	0.9	3.7	92.9	0.0	2.6	95.4	2.6
	ChatGPT	10.2	14.9	71.2	0.6	3.1	74.3	3.7
Average	Claude	25.6	15.1	55.2	1.4	2.7	57.9	4.2
	ChatGPT	26.5	25.2	43.5	3.3	1.6	45.1	4.9

Table 10: Cross-model comparison of S-grade rates per CVE (Claude Opus 4.8 vs. ChatGPT 5.5).

3.5 Impact of prompt richness (R) on outcomes

R	S1	S2	S3	S4	S5	Fail To Fix	New Vuln
−9	1.2	6.7	84.6	2.5	4.9	89.5	7.4
−6	13.9	4.9	77.5	1.1	2.7	80.2	3.7
0	23.9	23.6	48.4	2.6	1.5	49.9	4.1
1	20.9	17.1	59.6	1.4	1.0	60.6	2.4
9	24.6	29.5	42.3	1.4	2.2	44.5	3.6
10	29.4	25.6	39.5	1.8	3.8	43.3	5.6
12	36.8	14.8	44.9	2.9	0.5	45.4	3.5
13	41.4	27.1	26.2	3.5	1.8	28.0	5.3
21	41.9	30.3	22.8	3.7	1.4	24.2	5.1

Table 11: Impact of prompt richness (R) on outcomes. Negative R denotes the two incorrect-guidance prompt types, as in Figure 4. Rates are averaged over the six CVEs and both models, weighted equally, as in Table 7.

3.6 Impact of fix guidance on correctness

Guidance Type	S1	S2	S3	S4	S5	Fail To Fix	New Vuln
Correct	35.9	25.6	34.1	2.9	1.5	35.6	4.4
Incorrect	7.4	5.9	81.2	1.8	3.8	85.0	5.6
Not Provided	24.7	22.2	49.0	2.0	2.1	51.1	4.0

Table 12: Impact of fix guidance on correctness. Rates are averaged over the six CVEs and both models, weighted equally, as in Table 7.

3.7 Mode comparison

Mode	S1	S2	S3	S4	S5	Fail To Fix	New Vuln
Oneshot	22.8	20.0	52.5	2.1	2.5	55.0	4.7
Exploratory	25.9	19.6	51.1	1.6	1.7	52.8	3.4
Iterative	29.2	20.7	45.3	3.0	1.8	47.1	4.8

Table 13: Outcomes by patcher mode. Rates are averaged over the six CVEs and both models, weighted equally, as in Table 7.

3.8 Cost and efficiency

Cost figures are drawn from FLAWED's per-stage token accounting, summed over the 6,080 non-cheat classified iterations. We count the two stages a real remediation workflow would incur, being the patcher (candidate-patch generation) and the validator (the self-grading pass). We exclude the costs incurred by the cheat auditor, since unlike our benchmark, a company patching a not-yet-public vulnerability would have no need to run one. All figures are in US dollars. Claude amounts are the provider-reported billing totals, while amounts for ChatGPT (which does not expose equivalent cost data) are computed from token counts at list prices.

Model	Total	Per patch	Patcher	Validator
Claude Opus 4.8	$8,436.60	$2.81	$1.65	$1.16
ChatGPT 5.5	$6,493.96	$2.11	$1.19	$0.92
Combined Avg.		$2.46	$1.42	$1.04

Table 14: Cost overview by model (USD). "Per patch" is the mean cost of one patch attempt (iteration).

Model	Per attempt	Per S1 fix
Claude Opus 4.8	$1.65	$6.44
ChatGPT 5.5	$1.19	$4.49
Combined Avg.	$1.42	$5.47

Table 15: Generation spend per patching attempt (validation and audit costs excluded), and per clean fix (S1): the per-attempt cost divided by the clean-fix rate in Table 7, so that the cost of failed attempts is amortized into the price of each fix.

CVE	Claude	ChatGPT	Average
CVE-2026-22738 (*Spring-AI*)	$1.98	$2.15	$2.06
CVE-2026-31431 (*Linux*)	$4.57	$2.13	$3.35
CVE-2026-34197 (*ActiveMQ*)	$1.57	$1.85	$1.71
CVE-2026-45185 (*Exim*)	$2.41	$1.67	$2.04
CVE-2026-8512 (*Chromium*)	$1.65	$1.45	$1.55
GHSA-Wpqr-6V78-Jr5G (*Gemini-CLI*)	$5.39	$3.41	$4.40
Average	$2.81	$2.11	$2.46

Table 16: Mean cost per patch attempt by CVE (USD).

Mode	Claude	ChatGPT	Average
Oneshot	$2.69	$1.88	$2.29
Exploratory	$2.82	$2.62	$2.72
Iterative	$2.92	$1.89	$2.41

Table 17: Mean cost per patch attempt by harness mode (USD).

3.9 Patch size by outcome

Note: unlike the outcome tables above, which report the averaged self/cross-validator S-scores, the patch-size, fragility, and judge-accuracy tables in the remainder of this section are computed using self-validation scores only. The patch size is bucketed by each patch's own scenario, the "fragile" flag is a self-validation annotation, and the judge-accuracy comparison is between the self-validator and human review. The self- vs. cross-validator tables likewise show the two per-validator values side by side, as those are the inputs that the headline tables average.

Outcome	Avg Line Additions	Avg Line Deletions	Avg Files Changed
S1	69.3	16.6	2.5
S2	68.7	23.6	2.4
S3	52.2	13.8	2.6
S4	52.3	22.9	2.7
S5	88.6	25.3	3.4

Table 18: Average patch size by outcome, in lines-of-code.

3.10 Fragility of S1 and S2 patches

	S1 (clean fix)			S2		
Model	**Patches**	**Fragile**	**Fragile %**	**Patches**	**Fragile**	**Fragile %**
Claude	899	234	26.0	601	354	58.9
ChatGPT	378	92	24.3	816	330	40.4
Average	638	163	25.2	708	342	49.7

Table 19: Fragile patches among S1 and S2 patches, the two scenarios that carry a fragile flag.

3.11 Self- vs. cross-validator grading

Patches	Cross validator	Paired N	S-grade disagree	Bug-fixed judgment		New-bug disagree
				→ not fixed (stricter)	→ fixed (lenient)	
Claude	ChatGPT	2,999	27.5	17.3	0.8	6.3
ChatGPT	Claude	3,078	45.8	0.3	25.8	6.6
Pooled	—	6,077	36.8	22.2		6.5

Table 20: Self- vs. cross-validator disagreement, pairing each patch's own-agent second-pass verdict with the other model's cross-validation second-pass verdict on the same iteration (all six CVEs, non-cheat). All figures are percentages of paired iterations. The "bug-fixed judgment" columns split the disagreements on whether the original bug was fixed into the two directions, noting when the cross-validator flips the self verdict to *not fixed* (stricter) or to *fixed* (more lenient). Disagreement on this judgment is almost entirely one-directional and its direction is set by validator identity. ChatGPT-as-cross overturns Claude's "fixed" 17.3% vs. 0.8% of the time, whereas Claude-as-cross rescues ChatGPT's "not fixed" 25.8% vs. 0.3% of the time. New-bug disagreements are small (∼6%) and roughly symmetric. This mirrors the grading tables below and is consistent with ChatGPT itself grading more strictly overall rather than patcher self-preference.

Patches	Validator	N	S1	S2	S3	S4	S5	S1+S2	S4+S5	Mean S
Claude	Claude (self)	3,001	29.1	19.5	47.6	1.7	2.1	48.6	3.9	2.28
Claude	ChatGPT (cross)	2,999	22.1	10.6	62.8	1.1	3.3	32.7	4.5	2.53
ChatGPT	ChatGPT (self)	3,079	12.3	26.7	55.7	3.4	2.0	38.9	5.4	2.56
ChatGPT	Claude (cross)	3,078	40.7	23.7	31.3	3.1	1.2	64.4	4.3	2.00

Table 21: Self- vs. cross-validator grades by patcher model, using the individual per-validator second-pass verdicts (the two sides that the headline tables average together), weighted equally across the six CVEs, as in Table 7; N is the pooled count. "Self" is the patch graded by the model that produced it, "cross" by the other model; lower mean S is better. Percentages within each row sum across S1–S5.

Validator	N	S1	S2	S3	S4	S5	S1+S2	Mean S
Claude (as validator)	6,079	34.9	21.6	39.4	2.4	1.7	56.5	2.14
ChatGPT (as validator)	6,078	17.2	18.6	59.3	2.3	2.7	35.8	2.55

Table 22: Validator strictness, pooling each validator across both patchers' work (all six CVEs, weighted equally by CVE and patcher; N is the pooled count). Claude grades leniently and ChatGPT strictly, independent of who authored the patch.

CVE	Patcher	Lens	N	S1	S2	S3	S4	S5	S1+S2	Mean S
Spring	Claude	self	538	27.5	48.3	23.6	0.6	0.0	75.8	1.97
Spring	Claude	cross	537	10.2	23.3	65.7	0.6	0.2	33.5	2.57
Spring	ChatGPT	self	540	4.8	51.1	43.5	0.6	0.0	55.9	2.40
Spring	ChatGPT	cross	540	44.6	44.8	10.4	0.2	0.0	89.4	1.66
Linux	Claude	self	383	17.5	13.8	60.3	1.8	6.5	31.3	2.66
Linux	Claude	cross	383	6.0	4.2	70.2	4.7	14.9	10.2	3.18
Linux	ChatGPT	self	400	13.0	31.8	44.2	4.8	6.2	44.8	2.60
Linux	ChatGPT	cross	400	49.0	14.2	28.5	5.8	2.5	63.2	1.99
ActiveMQ	Claude	self	532	6.6	17.5	75.4	0.6	0.0	24.1	2.70
ActiveMQ	Claude	cross	531	0.0	3.8	95.9	0.0	0.4	3.8	2.97
ActiveMQ	ChatGPT	self	522	0.0	11.9	87.5	0.4	0.2	11.9	2.89
ActiveMQ	ChatGPT	cross	521	6.1	30.1	63.3	0.2	0.2	36.3	2.58
Exim	Claude	self	529	74.9	4.9	16.8	3.0	0.4	79.8	1.49
Exim	Claude	cross	529	76.7	0.0	21.9	0.6	0.8	76.7	1.49
Exim	ChatGPT	self	540	23.0	37.0	30.6	8.9	0.6	60.0	2.27
Exim	ChatGPT	cross	540	65.4	6.3	16.7	10.2	1.5	71.7	1.76
Chromium	Claude	self	529	46.5	25.5	21.4	4.3	2.3	72.0	1.90
Chromium	Claude	cross	529	39.1	32.1	25.5	0.9	2.3	71.3	1.95
Chromium	ChatGPT	self	537	32.8	25.0	35.6	5.2	1.5	57.7	2.18
Chromium	ChatGPT	cross	537	58.5	20.1	19.2	1.7	0.6	78.6	1.66
Gemini-CLI	Claude	self	490	1.4	6.9	88.0	0.0	3.7	8.4	2.98
Gemini-CLI	Claude	cross	490	0.4	0.4	97.8	0.0	1.4	0.8	3.02
Gemini-CLI	ChatGPT	self	540	0.0	3.1	92.6	0.6	3.7	3.1	3.05
Gemini-CLI	ChatGPT	cross	540	20.4	26.7	49.8	0.7	2.4	47.0	2.38

Table 23: Per-CVE self- vs. cross-validator grades across all six CVEs. "Self" is the patcher model grading its own patches; "cross" is the other model.

3.12 Judge accuracy vs human

Comparison	Reviewed	Matched	Agreement %
S-Grade (Overall)	1,052	693	65.9
Original Bug Fixed	1,052	923	87.7
Introduces New Bugs	1,059	747	70.5

Table 24: Validator (judge) accuracy versus human manual review.

4 Notable Findings

4.1 Incorrect guidance is worse than no guidance

A human developer addressing a bug generally requires some initial guidance on how to go about the task. This is true for LLMs, with perfectly correct guidance producing a **35.9%** clean-fix rate, compared to no guidance at **24.7%**. A surprising divergence is that when given incorrect advice on how to fix the vulnerability, the LLMs' clean-fix rate dropped to a low of **7.4%**. While human developers can certainly be misled by bad advice, in the authors' experience, they stand a good chance of catching incorrect guidance simply by virtue of needing to reason through the problem before implementing the fix. LLMs, on the other hand, appear much more prone to running with whatever is initially asked of them, even if it is incorrect.

A particularly notable behavior seen in the `appsec-prescriptive-wrong` prompts is that even when the agent's own tool calls returned information that directly contradicted and corrected the human prompt, the model optimized for obedience to the prompt in spite of the conflicting evidence in front of it. This is somewhat different from the interactive-session behavior, in which the model will occasionally correct the user, albeit still trending toward sycophancy. In the non-dialog format of the patcher sessions, however, the models appeared much less likely to catch themselves making mistakes that they might otherwise second-guess the user on.

In short, the result of incorrect initial guidance produces a profound drop in accuracy compared to the modestly-reduced success rate with the no-guidance prompts. Depending on one's risk tolerance, it may be more prudent to avoid giving bug-fixing LLMs any up-front guidance unless one is absolutely certain that the guidance is correct.

4.2 High levels of variation between codebases

While an overall trend is visible in the clean-fix statistics, the clean-fix rates were extremely variable between different CVEs, and thus codebases. Outcomes also varied between models on some CVEs, with the two models' clean-fix rates on the same CVE differing by as much as 31.6 percentage points (Table 10). Our research did not rigorously explore the exact reasons for these discrepancies, but this observation is enough to give one pause. That is, there seem to be a variety of complex factors inherent in the target codebase and bug that can drastically affect the clean-fix rate, and for a given codebase-bug pairing, it may not be clear how likely an LLM is to be able to successfully fix a given bug. This implies that advancements in LLM patching accuracy may not be spread evenly across programming languages, codebase structures, and application features. Furthermore, the features of the bug and codebase appear to matter significantly more than the model used for patching: averaged across CVEs, the two models' clean-fix rates differed by under 1 percentage point, while the per-target rates ranged from 3.2% to 60.0%.

Those considering adopting LLMs for automated patching would do well to first test an LLM's accuracy for producing patches against previously human-fixed CVEs in their own codebase as a baseline before deciding whether the success rate is comparable enough to be worth further investment.

4.3 Fixing one memory bug by adding another in Copy Fail

Copy Fail produced the highest new-vuln rate of any CVE for Claude (S4+S5 14.0%) and one of the highest for ChatGPT (9.6%, second only to Exim), and even that rate understates the problem. While patching the original Copy Fail vulnerability, models repeatedly introduced an off-by-one vulnerability of their own, which the validator mostly failed to flag (Section 4.4).

What makes this case striking is that they independently reproduced a regression that kernel maintainers had earlier made themselves (and ultimately fixed in a subsequent commit).

The original Copy Fail bug is a cross-mapping memory-corruption primitive in the Linux kernel's `AF_ALG` interface, where an in-place optimization drove the AEAD transform to treat the sender's transmit scatter-gather list and the receiver's distinct receive scatter-gather list as one aliased buffer, so tag and ciphertext bytes were written through pages the kernel should only have read. The upstream fix, commit `a664bf3d`, reverts the transform to out-of-place operation, and in doing so, introduced a fresh off-by-one of its own that the maintainers had to fix one commit later in `31d00156` (assigned CVE-2026-43078; Figure 2).

```
     if (dst) {
-        if (dst_offset >= plen) {
-            /* discard page before offset */
-            dst_offset -= plen;
-        } else {
-            /* reassign page to dst after offset */
-            get_page(page);
-            sg_set_page(dst + j, page, plen - dst_offset,
-                        sg[i].offset + dst_offset);
-            dst_offset = 0;
-            j++;
-        }
+        /* reassign page to dst after offset */
+        get_page(page);
+        sg_set_page(dst + j, page, plen, sg[i].offset);
+        j++;
     }
```

Figure 2: The upstream Copy Fail revert (`a664bf3d`) drops the `dst_offset` guard from `af_alg_pull_tsgl` and advances the destination index `j` for *every* source page. When leading pages should have been skipped for the offset, `j` walks one entry past the destination list — the off-by-one that upstream corrected in `31d00156`. The models repeatedly generated this exact `a664bf3d` pattern.

The change is subtle. The pre-revert reassignment path discarded source pages that fell entirely before `dst_offset`, trimmed the first partial page (`plen - dst_offset` bytes at `sg[i].offset + dst_offset`), and only then advanced the destination index `j`. The revert deleted that bookkeeping and reassigned every page whole, advancing `j` unconditionally. When a nonzero offset should have caused leading pages to be skipped, the new code emits an extra destination entry instead, so `j` runs one scatter-gather entry past the end of the destination list, introducing an off-by-one heap out-of-bounds write. It closes the original cross-mapping primitive and opens a new one in the same helper.

4.4 Humans and AI converge on the same flawed patch

When we ran FLAWED against Copy Fail, the models converged on precisely the aforementioned pattern seen in commit `a664bf3d`, in which they revert to out-of-place, drop the offset handling, and reassign each page whole. Comparing every candidate patch's rewritten `af_alg_pull_tsgl` against the upstream patch, we found the off-by-one in 129 of 400 non-cheat ChatGPT patches (32%) and 119 of 383 non-cheat Claude patches (31%). It appeared in every patcher mode, most often in exploratory mode, though that mode also had far fewer non-cheat patches (Table 25). Roughly a third of the time, the patcher regenerated exactly the same off-by-one present in the upstream revert and later corrected in `31d00156`.

Patcher mode	ChatGPT (gpt-5.5)			Claude (opus-4-8)		
	patches	n	%	patches	n	%
Oneshot	180	56	31.1	180	52	28.9
Iterative	156	43	27.6	175	50	28.6
Exploratory	64	30	46.9	28	17	60.7
All modes	400	129	32.2	383	119	31.1

Table 25: Candidate patches that reintroduced the `a664bf3d` off-by-one (n), as a fraction of the non-cheat candidate patches in each mode. Detected by comparing each candidate's rewritten `af_alg_pull_tsgl` against the pre- and post-`31d00156` upstream patch; the detector agrees with our manual review, which covered at least 10% of iterations, on all 253 reviewed iterations.

Notably, this regression was largely invisible to our automated evaluation. The dynamic reproducer did not detect it, and the validator classified it as equivalent to the upstream fix, as the reference diff against which candidates were graded incorporated the unguarded `a664bf3d` revert but not the subsequent `31d00156` correction. As a result, the validator identified the off-by-one as a newly introduced vulnerability in only 14 of the ChatGPT patches and 10 of the Claude patches, and it is consequently almost entirely absent from Copy Fail's reported new-vulnerability rate. We identified the regression only through a structural comparison of each candidate's rewritten `af_alg_pull_tsgl`, where the guarded post-`31d00156` form verifies that a page contributes a non-zero number of bytes (`plen`) before consuming a destination slot, whereas the `a664bf3d` form reproduced by the models does not. A line-level comparison against the accepted upstream patch does not distinguish the two.

4.5 A latent bug that every model missed in Copy Fail

Copy Fail, the Linux `authencesn` bug examined above, also exposes a quieter failure that all of the models left in place. Fixing the CVE requires reworking the `authencesn` decrypt path, whose Extended Sequence Number (ESN) handling relocates four bytes of high-order sequence number to the end of the authenticated data on every request. That move is out of bounds whenever the authentication tag is shorter than four bytes, a configuration the ESN code is meant to forbid but does not fully validate. Upstream commit `5db6ef98` closes the hole by rejecting, at instance-creation time, any hash whose digest size is non-zero but smaller than four bytes. The defect is independent of Copy Fail, in that it predates the vulnerable commit, is not among the CVE's fix commits, and the reproducer never exercises it, since the reproducer only fixes the tag length at four bytes.

Because nothing in the task pointed at it, every non-cheating patch missed the `5db6ef98` guard. Across both campaigns, all 783 non-cheat iterations left the ESN tail handling unguarded, including every patch graded as a clean fix of Copy Fail itself, and the 579 that edited `crypto/authencesn.c` directly; the only patches carrying the guard were four cheated iterations that had copied the upstream commit verbatim. The models fixed the vulnerability they were shown, but not one surfaced the out-of-bounds write sitting in the same function they were rewriting.

The models patch to the specific bug they are shown and the test they are given, rather than to the safety of the code they are editing. The unguarded four-byte move was present in the code the models were rewriting, yet none of the patching agents identified and remediated the vulnerability.

4.6 Tunnel vision

Models pay excessive attention to the problem presented exactly as worded, to the detriment of other relevant code that also needs to be addressed. In the world of human vulnerability reports, it is typical for exploits to be reported together with a proof-of-concept that exercises the application with a single specific malicious input. A human will readily understand that the input is an example of the general problem. With the LLM-generated patches, we observed that this was not always the case, as the models had a tendency to focus heavily on whatever exploit path was first presented to them. In the event that they were given an actual proof-of-concept in the form of a reproducer, the LLMs quite readily patched the PoC's exploit path while entirely missing analogous parallel code paths.

We read this as a form of reward hacking [6], with the model optimizing for a narrow observable signal of success rather than the property a human actually cares about, i.e., that the entire vulnerability class is mitigated. This is also, we suspect, the root of the fragility reported in Section 3.10, where 25.2% of patches graded as clean fixes (S1) resolved the immediate vulnerability but left the underlying primitive in place (though not necessarily reachable).

In other words, models given partial reproducers — ones that demonstrate a representative input but don't actually exercise all exploitable code paths — frequently generate only partial fixes in response. Furthermore, unlike the authors of this report, who had the benefit of already having upstream patches to compare to, developers faced with a new bug report in the wild will not have an "oracle" to tell them the full set of exploitable code paths that should be mitigated. As such, users of LLMs for automated patching should carefully weigh their tendency for overly-narrow goal-seeking versus the effort required to turn single proofs-of-concept into more thorough "test suites" that may be more likely to steer LLMs toward complete fixes.

4.7 Blindness to context outside immediately-visible code

Some vulnerabilities are governed not by the code directly in front of the model but by an indirection one step removed, such as an environment variable read at startup, or a trust decision in one module meant to gate behavior in another. Models tend to fix the visible half of such a bug while overlooking the indirect half, even when the governing context is present in the repository.

The clearest instance of this is the Gemini CLI trust-model advisory (`GHSA-wpqr-6v78-jr5g`), a two-part remote-code-execution bug in the headless operation where the CLI auto-trusts the workspace and loads workspace-local configuration, including a `.env` that seeds the process environment (so an attacker-supplied `NODE_OPTIONS` yields code execution), while the `--yolo` flag separately bypasses the tool allowlist. It was among the hardest bugs in our study to fix, with an average clean-fix rate of 5.6% across both models (Table 10). The recurring failure was the `.env` vector where the `--yolo` bypass and the auto-trust flag are directly visible and were routinely patched, but the `.env` ingestion occurs in a separate loader (`loadEnvironment` / `findEnvFile`) that must also be gated on trust. Models frequently de-trusted the workspace yet still loaded the attacker-controlled `.env`, without connecting the trust decision to that second path.

We quantified this from the reproducer, whose headless test checks four properties of a patch in a fixed order (culminating in the untrusted `.env` not being applied) and halts at the first failure. A run that fails only the final `.env` assertion therefore identifies a patch that satisfied every earlier trust property but still loaded the `.env`; and because the unpatched tree fails at the first assertion, the signature is unambiguous. In the "oneshot" mode (no test feedback) this pattern appears in **29%** of ChatGPT patches and **64%** of Claude's, and in a majority of all reproducer failures (52% and 72%); the exploratory arm is comparable (Table 26).

Patcher mode	ChatGPT (gpt-5.5)	Claude (opus-4-8)
Oneshot	29.4	63.9
Iterative	0.0	0.0
Exploratory	33.3	69.2
All modes	20.9	41.8

Table 26: Share of non-cheat patches for the Gemini CLI trust-model bug (`GHSA-wpqr-6v78-jr5g`) that stopped auto-trusting the workspace yet still loaded the untrusted `.env` into the environment, where the reproducer's `.env` assertion failed while every earlier trust assertion passed. Percentages are over non-cheat iterations per mode (180, except Claude exploratory 130 after excluding 50 cheated runs; ChatGPT has none).

The iterative arm serves as a control where, given the reproducer that exercises the `.env` path directly, agents converged on gating it and the signature falls to zero, though passing the two focused tests is still not the full upstream fix. This suggests the blind spot is one of attention rather than capability. Unprompted, models did not appear to reason about the environment-loading path; when it's named by a test, they fix it. More broadly, models track the code they can see and the changes they are asked to make, but often fail to account for the indirect, cross-file relationships that determine whether a fix is complete. Patching such vulnerabilities reliably may therefore require surfacing those relationships to the model explicitly, as the iterative arm's tests did.

4.8 No consistent self-preference effect observed

When we built FLAWED's cross-validation step, we did so partly in anticipation of LLMs' documented self-preference bias [10]; that is, the tendency of models to judge their own output more favorably than a neutral party would. We expected each model to grade its own patches more generously than its cross-validator did. Across all six CVEs, with every patch graded by both models, we found no consistent effect. The dominant signal was instead a difference in validator strictness in which Claude graded patches more leniently no matter who wrote them (about 56% rated S1 or S2, i.e., fixed without introducing a new vulnerability), while ChatGPT graded more strictly across the board (about 36%; see Table 22). If anything, the within-validator comparison actually ran against self-preference. We found that Claude rated its own patches markedly more strictly than it rated ChatGPT's (48.6% versus 64.4% rated S1 or S2), and ChatGPT was somewhat more lenient on its own (38.9% versus 32.7%). This last comparison admittedly conflates two possible factors: how a model grades patches, and the actual quality of that model's patches. The most persuasive evidence is the across-the-board differences in strictness, which held for each validator regardless of which model's work it was judging. What looks at first like a model flattering its own work seems on closer inspection actually to be an artifact of which model happened to be grading, not of any tendency for self-preference.

In short, whatever self-preference bias these models may have when reviewing code was much smaller than the difference in strictness between ChatGPT and Claude. This is one factor that led us to average the verdicts of both validators when computing our headline statistics, so that neither model's systematic differences in judgment (whether biased for or against itself) would skew the results.

4.9 Fixing one pointer's lifetime but not the other in Chromium

The Chromium File System Access bug (`CVE-2026-8512`) is a use-after-free in the macOS `FilePathWatcherFSEvents` watcher. The OS delivers directory events by invoking a static callback with a context pointer that, in the vulnerable code, is the bare `this` of the watcher; during teardown the watcher can be destroyed on its task-runner sequence while that callback is still running on a libdispatch thread, so the callback dereferences freed memory in the browser process (a sandbox-escape primitive). The upstream fix replaces the raw `this` with a reference-counted `FSEventsContext` that owns the watcher's `WeakPtr` and task runner. Keeping that context alive requires a strong reference in *two* places. First, at the install site, the context is created as a `scoped_refptr` and its `retain` and `release` hooks are wired into `FSEventStreamContext`, so the OS holds a reference while the stream is active. Second, at the top of the callback, the incoming context is wrapped in a local `scoped_refptr` (via `base::WrapRefCounted`), so the callback owns a reference for its own duration. The second reference is what makes the fix correct, as the teardown invalidates the stream and drops the OS's reference, and without the callback's own `scoped_refptr` that release can free the context mid-callback.

Roughly 2 in 5 of the patches that adopted the reference-counted architecture secured the first pointer but missed the second. Of the non-cheat patches that adopted it (66.7% for ChatGPT, 58.4% for Claude), **41.9%** (ChatGPT) and **38.5%** (Claude) omitted the callback's local `scoped_refptr`, using the context as a raw pointer inside the callback (Figure 3 and Table 27). The result is not a failure to fix but a relocation of the use-after-free, where the watcher is now safe, but the callback holds a raw pointer to the context, which a concurrent teardown can free underneath it.

```
// FSEventsCallback runs on a libdispatch thread. During teardown the stream is
// invalidated and the OS drops its reference to the context.
FSEventsContext* context =
    static_cast<FSEventsContext*>(event_watcher);   // raw pointer, no ref held

context->task_runner()->PostTask(
    FROM_HERE,
    base::BindOnce(&FilePathWatcherFSEvents::OnFilePathsChanged,
                   context->watcher(), is_root_changed_event,
                   root_change_at, std::move(events)));
```

Figure 3: A representative *missed-second-pointer* patch. The context is now reference-counted and its `retain`/`release` hooks are wired at the install site, but the callback binds it by raw pointer (highlighted). The upstream fix instead opens the callback with a local reference, `scoped_refptr<FSEventsContext> context = base::WrapRefCounted( static_cast<FSEventsContext*>(event_watcher));`, so a concurrent teardown `release` cannot free the context while the callback is still dereferencing it.

To measure this at scale, we classified each candidate diff structurally. A patch was counted as adopting the reference-counted architecture when it created a ref-counted context, assigned it to `FSEventStreamContext::info`, and wired non-null `retain` and `release` hooks; it was counted as securing the callback when it additionally bound the incoming context to a local `scoped_refptr` (via `WrapRefCounted`, or a `scoped_refptr` initialized directly from the cast). Cheated iterations are excluded, so the reported shares reflect genuine patching attempts. This classification agrees with our manual review, which covered at least 10% of iterations, on 95% of reviewed iterations (45 of 47 flagged as insecure, 40 of 42 confirmed clean).

	ChatGPT (gpt-5.5)		Claude (opus-4-8)	
Patcher mode	**ref-counted**	**missed 2nd**	**ref-counted**	**missed 2nd**
Oneshot	51.7	26.9	38.9	11.4
Iterative	88.3	59.7	94.4	64.1
Exploratory	59.9	28.3	40.8	2.9
All modes	66.7	41.9	58.4	38.5

Table 27: Outcomes for the Chromium FSEvents use-after-free (`CVE-2026-8512`). *ref-counted* is the share of the non-cheat patches per mode that adopted the reference-counted context architecture (created the context, set `FSEventStreamContext::info`, and wired `retain`/`release`); denominators are 180 per mode apart from the exploratory arm (177 for ChatGPT, 169 for Claude) after excluding cheated patches. *missed 2nd* is the share of *those* patches that then omitted the callback's local `scoped_refptr`, leaving the callback with a raw pointer to the context.

The iterative arm is especially revealing, as it had both the highest adoption of the reference-counted fix and the highest omission rate (59.7% for ChatGPT and 64.1% for Claude). The reproducer deletes the watcher while a callback is paused, but it does not model the OS dropping its context reference during teardown, so a patch missing the callback's `scoped_refptr` still passes; iterative agents therefore converge on this reproducer-passing-but-incomplete fix. The gap was also largely invisible to the LLM validator, which graded many of these patches as clean, so Chromium's reported clean-fix rate is likely overstated; it became apparent only under structural comparison against the upstream patch or manual review.

5 Discussion

5.1 How good are LLMs, generally, at fixing vulnerabilities?

Across our entire dataset, only 26.0% (S1) of patches fully mitigated the target vulnerability without any ill side effects. 20.1% (S2) fixed the original issue but introduced discrepancies in application behavior that, while not immediately identifiable as security issues, constitute bugs in their own right; 2.3% (S4) did so while introducing identifiable security issues. 49.3% (S3) failed to fix at least one extant exploit path; and 2.2% not only failed to fix the vulnerability, but also introduced a new exploit path. Restricting the data to the seven prompt types with only factually accurate information (Table 28) raises the clean-fix rate to 31.2% and lowers the fail-to-fix rate from 51.5% to 42.2%, though most patches still fall short of a clean fix. Because these rates are averaged over only six CVEs whose outcomes varied widely, they are imprecise: treating each CVE as one observation, the 95% confidence interval around the 26.0% clean-fix rate spans roughly 3% to 49%.

Put in simpler terms, when even a frontier LLM autonomously generates a patch for complex, novel vulnerabilities like those we tested, from inputs that are not strictly vetted for full correctness, there is only a **roughly 1 in 4 chance** that it will do so successfully. There is a roughly 50-50 chance that it will fail to fix the original bug, a 1 in 4 chance that it will introduce a new bug in general, and nearly a 1 in 20 chance that it will introduce a new security vulnerability specifically. As such, **in our experiments, a fully LLM-generated, non-human-reviewed patch was considerably more likely to be flawed than not**. Vetting inputs for correctness helps, but is not sufficient on its own: with only factually accurate information, the clean-fix rate rose only to roughly 1 in 3 (Table 28). Even this best case falls well short of reliable autonomous remediation, and it relies on an advantage that real-world defenders lack: because we had the upstream patches, we knew in advance which information was correct. For a novel vulnerability without a known fix, establishing that is far from trivial.

Metric	Claude (%)	ChatGPT (%)	Average (%)
Total Valid Iterations count	2,346	2,401	4,747
S1 Rate	30.3	32.1	31.2
S2 Rate	17.7	30.6	24.1
S3 Rate	48.2	32.7	40.5
S4 Rate	1.5	3.6	2.5
S5 Rate	2.4	1.1	1.7
Fail-To-Fix Rate (S3+S5)	50.6	33.8	42.2
New Vuln Rate (S4+S5)	3.8	4.6	4.2

Table 28: Aggregate outcomes restricted to the seven prompt types whose information is factually accurate (correct or no fix guidance), at varying levels of richness. This approximates a best case in which every input to the patcher is known to be accurate, which is difficult to ensure in practice. Compare Table 7, which includes all nine prompt types.

5.2 What aspects of the model's prompt and environment matter most?

The information given to patcher agents varies along two largely independent axes: its *amount*, measured by the magnitude of R (Section 2.5.1), and its *quality*, i.e., whether any fix guidance it contains is correct, indicated by the sign of R. A prompt can be rich but wrong, or brief but accurate. Quality mattered most: correct guidance produced a 35.9% clean-fix rate versus 7.4% for incorrect guidance, a 28.5-percentage-point difference (Table 12), and the richest incorrect prompt ($R = -9$, 1.2%) fared far worse than a prompt containing no information at all ($R = 0$,

23.9%). Amount also helped substantially, though less so than quality: among prompts with accurate information, the clean-fix rate rose by 18.0 percentage points from $R = 0$ (23.9%) to $R = 21$ (41.9%), a clearly positive relationship between richness and clean-fix rate, though not a steady one: the rate stayed roughly flat up to $R = 9$, and most of the gain came between $R = 9$ and $R = 13$ (Figure 4 below).

Interestingly, the new-vuln rate (where a lower value is better) does not show nearly as clear a relationship. There may be a trough at low-to-middling amounts of richness, evening back out to near its starting value with higher richness. However, the data is scattered enough, and the number of data points along the R axis few enough, that we will refrain from making any definitive statements about the relationship between new-vuln rate and richness.

By comparison, the patcher itself made relatively little difference on average. The two models' overall clean-fix rates were nearly identical (25.6% for Claude and 26.5% for ChatGPT), although on some targets they differed substantially, by up to 31.6 percentage points (Table 10). With six CVEs, the 95% confidence interval around that 0.9-point difference spans roughly −17 to +19 points, so our data cannot say whether there is a substantial difference between the models on average; absent such evidence, we treat neither model as better than the other. The mode made less of a difference than we expected, just 6.4 percentage points between the best and worst modes, with oneshot mode having a 22.8% clean-fix rate, exploratory at 25.9%, and iterative at 29.2% (Table 13). The target mattered far more than either: averaged across both models, clean-fix rates ranged from 3.2% (ActiveMQ) to 60.0% (Exim) depending on the vulnerability and codebase (Table 10).

Overall, it appears that the best starting conditions for a model are **an iterative harness supplied with correct, information-rich, root-cause oriented context**.

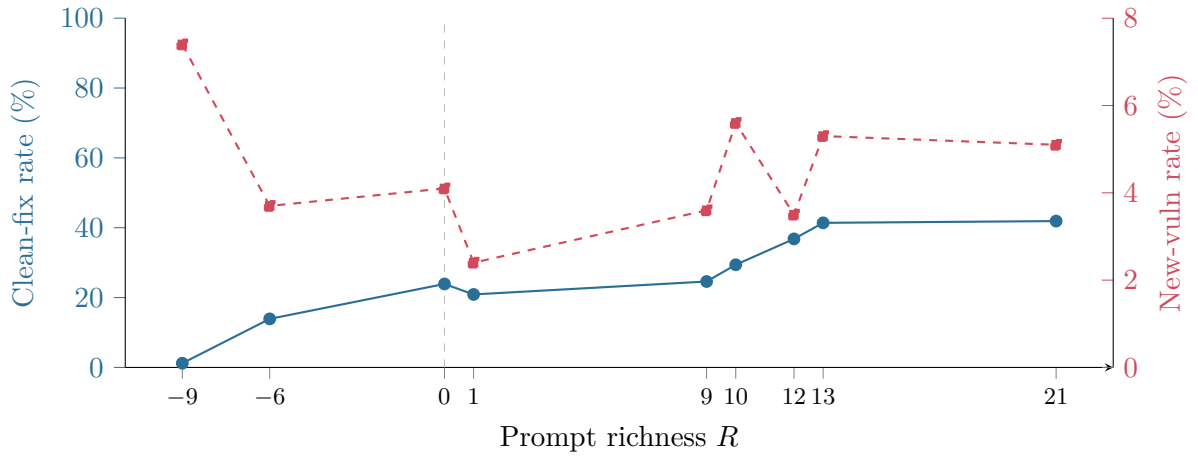


Figure 4: Clean-fix (S1) rate (solid, left axis) and new-vulnerability rate (dashed, right axis) against prompt richness R, using the averaged self/cross-validator S-scores, with the six CVEs weighted equally. Note the two axes differ in scale: clean-fix spans ∼41 points across the R range, while new-vuln stays within a few points (2.4–7.4%; R-9 → R21: 7.4% → 5.1%). The two incorrect-guidance prompt types are plotted at *negative* richness since they convey bad fix advice; thus, we treat them as worse than the zero-information prompts (their guidance effect is also reported in Table 12).

5.3 Richer context helps, but has limits

Models generally saw improvements in clean-fix rate when given more context, assuming that context was correct. On the other hand, even considering correct context alone, the new-vuln rate did not fall with richness: it varied between 2.4% and 5.6% with no consistent trend, ending slightly higher at the richest end (5.1%) than with no context (4.1%).

Additional richness also shows diminishing returns: the clean-fix rate is essentially unchanged between $R = 13$ and $R = 21$ (41.4% and 41.9%), levelling off at roughly 2 in 5 despite the richest prompt specifying every dimension in full. Richer (correct) context thus buys more successful fixes, but does not reduce the chance of introducing a new vulnerability.

5.4 Incorrect guidance leads to correctness collapse

When remediation direction has a substantial chance of containing inaccurate information—such as unvetted information from a SAST tool, bug-bounty report, or another AI agent—the safest action is to give the patcher only the bug, rather than a confidently-wrong direction. Incorrect guidance cost more than correct guidance gained, despite carrying less information: relative to providing no fix guidance, the two incorrect prompts ($|R|$ of 6 and 9) lowered the clean-fix rate by 17.3 percentage points, while the four correct prompts (R of 9 to 21) raised it by 11.2 (Table 12). At equal richness the gap is starker: at $|R| = 9$, the correct prompt produced clean fixes 24.6% of the time and the incorrect one 1.2% (Table 11). Where the accuracy of guidance is uncertain, it may therefore be better to err on the side of providing less information to the model. This is a particular concern for a fully-automated "AI discovery to AI patching" pipeline without human vetting between the bug-hunter and patcher models, since any inaccuracies in the bug-hunter's report would pass directly to the patcher.

6 Case Study: Patch the Planet

Patch the Planet[11] is a June 2026 initiative within OpenAI's Project Daybreak, built with Trail of Bits, with HackerOne and Calif supporting triage and coordinated disclosure. Trail of Bits engineers use OpenAI's models to find vulnerabilities, review the findings, and work with maintainers on fixes that can be merged upstream. Its premise is that finding bugs is now cheap, and writing a correct fix is where the effort lies.

Writing that fix is hard even for maintainers. Pull request #35 (PR35), a Patch the Planet submission, addresses a use-after-free in freenginx's embedded HTTP Perl module, where asynchronous callbacks and printed scalars are stored as raw `SV`/`CV` pointers without a reference count. The maintainers instead shipped their own fix, commit `cf26435a`. On review, both fixes are FLAWED. PR35 fixes only the two callback sites and leaves the `print` use-after-free in place, while the maintainer's fix closes all three sites. Both, however, release their references during request teardown, which introduces the same client-triggerable crash.

Because PR35 was submitted through an AI-driven patching initiative, we ran a FLAWED campaign against the same vulnerable commit to see which patches a model consistently converges on. Since Patch the Planet uses OpenAI's models through its partnership with OpenAI, we used ChatGPT. We automatically assessed each candidate with our patch-review harness against the upstream diff and for newly introduced regressions.

6.1 freenginx: HTTP Perl async callbacks store unrefcounted SV/CV pointers

freenginx's embedded Perl module (`ngx_http_perl_module`) lets a `location` block, the nginx configuration block that matches a set of request URIs and defines how they are served, run a Perl handler. Several of its eXternal Subroutine (XS) methods keep a bare C pointer to a Perl scalar (`SV`) or code value (`CV`) and dereference it *after* the XS call has returned, which occurs either when a deferred event fires, or when the buffered response is flushed. None of these paths increments the scalar's Perl reference count, so if the scalar has a limited lifetime, Perl's garbage collector can free it before nginx is done with it. In configurations where the scalar originates from a shorter-lived context (described below), this bug is reachable from a single unauthenticated HTTP request.

There are three retention sites, all in `src/http/modules/perl/nginx.xs`. The first two sites are the deferred `$r->sleep()` and `$r->has_request_body()` callbacks (Figures 5 and 6). Both store the handler CV as a raw pointer and arm an asynchronous continuation. The handler argument is type-checked, but the stored reference is never ref-counted.

```
void
has_request_body(r, next)
    CODE:
    # ...
    next = ST(1);

    if (!SvROK(next) || SvTYPE(SvRV(next)) != SVt_PVCV) {
        croak("has_request_body(): no handler provided");
    }

    ctx->next = SvRV(next);
    # ...
```

Figure 5: `has_request_body` stores the callback `CV` in `ctx->next` with no `SvREFCNT_inc` (nginx.xs#L415–L421).

```
void
sleep(r, sleep, next)
    CODE:

    # ...
    next = ST(2);

    if (!SvROK(next) || SvTYPE(SvRV(next)) != SVt_PVCV) {
        croak("sleep(): no handler provided");
    }

    ctx->next = SvRV(next);
    # ...
```

Figure 6: `sleep` does the same: `ctx->next` outlives the call, the reference count does not (nginx.xs#L1157–L1163).

The third site is the `$r->print()` zero-copy fast path (Figure 7). For a single read-only scalar, `print` avoids copying the bytes into the request pool and instead points the output buffer directly at the scalar's `PV`. That buffer is flushed after the handler returns, but the `SV` it aliases is not kept alive:

```
void
print(r, ...)
    CODE:
    # ...
        if (SvREADONLY(sv) && SvPOK(sv)) {

            p = (u_char *) SvPV(sv, len);
            # ...
            b->memory = 1;
            b->pos = p;        # buffer now aliases the SV's PV
            b->last = p + len;
    # ...
```

Figure 7: `print` aliases a read-only `SV`'s buffer without owning it (nginx.xs#L673–L703). The same `SvREADONLY` aliasing also lives in the shared helper `ngx_http_perl_sv2str` (nginx.xs#L41–L53).

The defect remains latent whenever the scalar's lifetime spans that of the request. Scalars owned by the persistent Perl-handler optree, such as string literals and inline `sub {}` definitions, persist for the lifetime of the configuration, so the absent reference count is inconsequential. The vulnerability manifests only when the scalar originates from a shorter-lived context; the canonical trigger is `eval`:

```
location / {
    perl 'sub {
        my $r = shift;
        $r->sleep(100, eval q!sub {
            my $r = shift;
            $r->send_http_header;
            $r->print("it works");
            return OK;
        }!);
        return OK;
    }';
}
```

Once `eval` returns, its optree is destroyed and the sole remaining reference to the callback `CV` is released. When the 100 ms timer subsequently fires, `ctx->next` refers to freed memory and nginx dispatches through it, yielding a heap use-after-free that manifests as an HTTP 500 response ("Can't use an undefined value as a subroutine reference.") or a worker crash. The `print` variant is analogous, where the `eval q!$r->print("it works")!` flushes an output buffer that aliases an already-freed read-only `PV`.

6.2 The Rejected Patch

Pull request #35, introduced as part of the Patch the Planet initiative, takes a callback-only, context-lifecycle approach. It pins the two deferred callbacks by incrementing the stored `CV` at the point of capture (Figure 8):

```
ctx->next = SvRV(next);
SvREFCNT_inc(ctx->next);
```

Figure 8: Pull request #35 ref-counts the stored callback inline (freenginx/nginx#35).

It then arranges to drop that reference in two places. First, when the callback actually fires, the async dispatch path decrements it after the handler runs (Figure 9):

```
void
ngx_http_perl_handle_request(ngx_http_request_t *r)
{
    // ...
   } else {
        sub = ctx->next;
        handler = &ngx_null_name;
        ctx->next = NULL;
        async = 1;
    }

    rc = ngx_http_perl_call_handler(aTHX_ r, ctx, pmcf->nginx, sub, NULL,
                                    handler, NULL);

    if (async) {
        SvREFCNT_dec(sub);
    }
    // ...
}
```

Figure 9: Pull request #35 drops the reference on the async dispatch path (freenginx/nginx#35).

Second, to cover the case where the request is destroyed before the callback ever fires, it folds context creation into a new helper and registers a **ctx-level pool cleanup** that decrements whatever callback is still pending (Figure 10):

```
static ngx_http_perl_ctx_t *
ngx_http_perl_create_ctx(ngx_http_request_t *r)
{
    // ...
    cln = ngx_pool_cleanup_add(r->pool, 0);
    // ...
    cln->handler = ngx_http_perl_cleanup_ctx;
    cln->data = ctx;
    // ...
}

static void
ngx_http_perl_cleanup_ctx(void *data)
{
    ngx_http_perl_ctx_t  *ctx = data;
    // ...
    if (ctx->next == NULL) {
        return;
    }
    // ...
    next = ctx->next;
    ctx->next = NULL;
    SvREFCNT_dec(next);      /* runs during ngx_destroy_pool(), r->pool == NULL */
}
```

Figure 10: Pull request #35's request-pool cleanup, the source of its regression (freenginx/nginx#35).

Two things characterize this approach. First, it is narrow—it never touches `print` or `ngx_http_perl_sv2str`, so the read-only print aliasing UAF is left unfixed. Second, the decrement runs at a dangerous moment. The ctx-level cleanup (Figure 10) does not run when the callback fires or is cancelled; like all pool cleanups it runs inside `ngx_destroy_pool`, which `ngx_http_free_request` invokes only *after* it has already nulled the request pool (Figure 11):

```
void
ngx_http_free_request(ngx_http_request_t *r, ngx_int_t rc)
{
    // ...
    r->connection->destroyed = 1;
    // ...
    pool = r->pool;
    r->pool = NULL;

    ngx_destroy_pool(pool);    /* pool cleanups (and any SvREFCNT_dec) run here */
}
```

Figure 11: When pool cleanups run, `r->pool` is already `NULL` (ngx_http_request.c#L3937–L3947).

`SvREFCNT_dec` is also more than a counter decrement. Dropping a `CV` to zero can run a captured object's `DESTROY`, which is arbitrary user Perl. If that Perl calls an allocating `$r` method (such as `$r->unescape` or `$r->header_in`), it reaches `ngx_pnalloc(r->pool, ...)` with `r->pool == NULL` and crashes the worker with a `NULL` dereference. A slow client that sends headers but withholds the body trips `client_body_timeout` while a callback is still pending, so where the callback makes such a call, the crash is reachable on a single request. Pull request #35 thus partially fixes the callbacks' use-after-free but introduces a new, client-triggerable crash in its place.

Under the taxonomy in our report, **pull request #35 qualifies as an S5 patch**. It fails to fully remediate the existing vulnerability, leaving the read-only `print` use-after-free untouched,

while also introducing a new one, the client-triggerable NULL-pointer dereference at request teardown.

6.3 The Accepted Patch

The accepted commit `cf26435a` takes a generic approach that covers all three retention sites. Instead of special-casing the callbacks, it adds one small helper that pins any retained scalar for the life of the request (Figure 12):

```
static ngx_int_t
ngx_http_perl_refcount(pTHX_ ngx_http_request_t *r, SV *sv)
{
    ngx_pool_cleanup_t       *cln;
    ngx_http_perl_cleanup_t  *clnp;

    cln = ngx_pool_cleanup_add(r->pool, sizeof(ngx_http_perl_cleanup_t));
    if (cln == NULL) {
        return NGX_ERROR;
    }

    cln->handler = ngx_http_perl_refcount_cleanup;

    clnp = cln->data;
    clnp->request = r;
    clnp->sv = sv;

    SvREFCNT_inc(sv);

    return NGX_OK;
}
```

Figure 12: The per-scalar refcount helper, called at all three sites (nginx.xs#L58–L78).

The matching cleanup re-enters the interpreter and drops the reference (Figure 13):

```
void
ngx_http_perl_refcount_cleanup(void *data)
{
    ngx_http_perl_cleanup_t  *clnp = data;
    // ...
    SvREFCNT_dec(clnp->sv);    /* also runs during ngx_destroy_pool() */
}
```

Figure 13: The per-scalar refcount cleanup (ngx_http_perl_module.c#L311–L331).

It also reworks the output path. Rather than aliasing a scalar it does not own, it removes the `SvREADONLY` zero-copy optimization in `ngx_http_perl_sv2str` entirely, and in `print` it both pins the scalar and broadens the fast path from read-only-only to any `SvPOK` scalar (Figure 14):

```
-       if (SvREADONLY(sv) && SvPOK(sv)) {
+       if (SvPOK(sv)) {

            p = (u_char *) SvPV(sv, len);

            if (len == 0) {
                XSRETURN_EMPTY;
            }

+           if (ngx_http_perl_refcount(aTHX_ r, sv) != NGX_OK) {
+               ctx->error = 1;
+               croak("ngx_http_perl_refcount() failed");
+           }
```

Figure 14: `cf26435a` fixes the print site that pull request #35 does not address (nginx.xs#L702–L712).

Conceptually, `cf26435a` generalizes the fix and reworks `print`; pull request #35 special-cases the callbacks and leaves `print` unchanged. `cf26435a` is the more complete patch, as it closes all three sites. But its decrement (Figure 13) still runs from an `r->pool` cleanup during `ngx_destroy_pool`, so for the callback (`CV`) sites it inherits the exact same teardown NULL-dereference described for pull request #35 (Figure 11). The accepted upstream patch fixed the use-after-free and shipped the regression with it.

`cf26435a`, by contrast to pull request #35, **is an S4 patch**. It successfully fixes the original use-after-free, but with an erroneous code change, releasing the reference from a request-pool cleanup, leading to a new vulnerability.

6.4 FLAWED Patches in the Wild

To characterize how AI patchers approach this bug, we used FLAWED to run a large patcher campaign against the vulnerable freenginx commit and assessed every candidate with the validator and structural classifier. The campaign sampled ChatGPT (under Trusted Access for Cyber guardrails) across three agent modes (oneshot, iterative, and exploratory) and nine template prompts, at ten iterations each, yielding 270 candidate patches. Each patch was assessed along two axes: an *outcome* judgment from the validator (whether it fixes the original bug and whether it introduces a new one), and a *structural* judgment from an independent classifier that assigned the patch, from its diff alone, to one of the two upstream fix shapes or to neither. Unlike the main campaigns, in which the validator treats the upstream patch as fully correct (Section 2.6), the validator for this campaign was explicitly instructed to flag the upstream fix's teardown-time reference release as a new vulnerability.

These judgments answer different questions. A patch can resemble a reference diff or modify the `print` alias without correctly fixing all three vulnerable retention sites. It can also move the unsafe lifetime behavior elsewhere. Structural similarity therefore does not establish successful remediation.

The self-validator's raw outcome fields marked the original use-after-free as fixed in 114 patches (42.2%) and a newly introduced vulnerability in 232 (85.9%). Its later self-validation S1–S5 distribution was S1: 0, S2: 1, S3: 39, S4: 114, and S5: 116. The Claude cross-validator assigned S1: 35, S2: 8, S3: 74, S4: 71, and S5: 82. Thus ChatGPT's self-validator assigned no patch to S1, while Claude assigned 35 (13.0%); giving the two passes equal weight produces an averaged S1 rate of 6.5%. Claude's 35 S1 classifications were not manually reviewed.

The raw outcome fields and the later S1–S5 labels are separate validator products. Recombining the self-validation scenario labels would yield 115 fixed and 230 new-vulnerability labels, so we

do not substitute those sums for the raw-field 114/232 outcome claims above.

By structure, 214 patches (79%) closed the `print` alias, 20 (7%) ref-counted only the callbacks, and 36 (13%) added guards without taking a reference. All 114 patches that the raw self-validator marked as fixes were among the 214. Of the other 100, 93 left the separate `ngx_http_perl_sv2str` alias unaddressed and nine failed to build; two did both. This accounts for the gap between the structural and outcome counts.

Prompt guidance strongly affected the structure. The **default**, **appsec-prescriptive-correct**, **sast**, and **embargo** prompts produced 30/30 print-closing diffs; **bugbounty** and **dev-self-discovery** produced 26/30; **poc** produced 23/30; and the misleading **ai-agent-discovery** and **appsec-prescriptive-wrong** prompts produced 11/30 and 8/30.

At the line level, the closest candidates matched 38 of the accepted patch's 47 unique, nonblank changed lines (80.9%), preserving whether each line was added or removed. Both reference fixes release retained values from a pool cleanup, so patches that faithfully reproduce this cleanup behavior may inherit the same teardown crash. According to the validator rationales, this was the predominant newly introduced vulnerability in the campaign.

6.5 Responsible Disclosure

We reported the null-pointer dereference vulnerability discussed in previous sections to freenginx maintainers in accordance with the instructions on their Security Advisories page on June 29th, 2026. A patch for the issue was published on July 2nd, 2026.

7 Recommendations

7.1 Validate the FLAWED rates of your LLM against your own codebase

The clean-fix and new-vulnerability rates of the two models varied widely between the six codebases the patcher agents performed work on, with clean-fix rates ranging from 0.9% to 75.8% (Table 10). While our research was sufficient to identify the existence of a pronounced level of variability, we have not yet identified the specific factors that contributed to this variance. We also did not test how much of the difference is attributable to the CVE content as opposed to the codebases themselves, as we only tested one CVE per codebase.

The takeaway for developers is that frontier LLMs exhibit large differences in patching efficacy, in presently unpredictable ways, when leveled against different bugs and codebases. As such, we recommend that anyone considering using LLMs for highly- or fully-automated patching pipelines first use the FLAWED toolchain to determine how effective their specific LLM of choice is at patching vulnerabilities in their specific codebase. To get this process started, we suggest leveraging your organization's bug tracker to provide a set of previous bug reports and corresponding patches to use as input data for FLAWED.

7.2 Don't trust unreviewed LLM-generated patches, and don't assume reviewing them is cheap

In our experiments, an LLM patching a complex vulnerability without human review was, on average, more likely to produce a flawed patch than a clean fix — either one that appears to fix the bug while leaving at least one code path exploitable, or one that introduces an entirely new vulnerability. This held even when every input was known to be factually accurate (Table 28), a bar that is difficult to clear in practice. Performance varied widely between vulnerabilities (Table 10): averaged across both models, the clean-fix rate ranged from 60.0% on Exim to 3.2% on ActiveMQ. Because performance on a given vulnerability is hard to predict in advance, there is no reliable way to know whether any single unreviewed patch is one of the successful ones. Given this, careful manual review of LLM-generated patches by domain experts remains necessary to ensure that a given patch actually mitigates the target vulnerability.

That said, it remains an open question whether using LLM-generated patches with human review is actually cost- and time- effective compared to human-generated patches (with normal levels of LLM coding assistance). With only roughly 1 in 4 patches being fully successful given input data of mixed correctness (and roughly 1 in 3 even with inputs known to be fully correct), and many of those solutions being fragile, most patches submitted to human auditors would need to be rejected. Because flawed and correct patches are often superficially similar, telling them apart requires close reading of every submission, and the small differences that separate them are easy to miss. The cognitive load of fully understanding a patch, especially at the granular level required to understand its full security implications, should not be underestimated. In our experience, borne out by our manual reviews, the level of understanding one must build to confidently evaluate the full correctness of a vulnerability patch is often at least what would have been sufficient for a human programmer to produce a single, known-good patch in the first place. Developers who receive a large volume of highly similar but subtly different patches for review will likely exhaust their mental reserves in short order and—especially under time pressure—resort to cognitive surrender [12].

7.3 Be cautious about providing partial success criteria

In our experiments, LLMs were highly sensitive to the success criteria provided by the user when asked to complete a task. They tended to do exactly what was literally asked of them and nothing more, often failing to fully address an issue unless its exact success criteria were

spelled out in detail. We observed this trend in the agents' tendency to patch only a single code path touched by a proof-of-concept exploit, while missing even character-for-character identical instances of the same bug in adjacent code paths.

The tendency of LLMs to hyper-focus on an individual code path without attempting to comprehend the bigger picture of the full vulnerability means that they are much more sensitive to their initial inputs—a bug report and single proof-of-concept input, for instance—than we would expect of a human reviewer. While it is possible to constrain them with carefully-defined comprehensive success criteria, multiple reproducers, and so on, we caution developers that the effort required to create such harnesses may prove greater than what is necessary for a human to understand and patch the original vulnerability. After all, any given vulnerability ideally should need to be patched only once, and if a large amount of individual scaffolding is required for an LLM to reliably generate a patch for each one, the proverbial juice may not be worth the squeeze [13].

7.4 Be extremely cautious when it comes to providing incorrect guidance

LLMs' reward-seeking tendencies mean that they will often seek to fulfill even small textual details of the user's specific request ("I think approach X is needed...") regardless of whether that request actually achieves the user's high-level intent ("...to fix this vulnerability"). As such, it is extremely easy to steer the model toward using an incorrect approach by getting details wrong in the task prompt.

Across our patching campaigns, incorrect guidance in the initial prompt lowered the clean-fix rate by roughly 17 percentage points relative to no fix guidance (from 24.7% to 7.4%), while correct guidance raised it by only roughly 11 points (to 35.9%; Table 12), even though the correct-guidance prompts were considerably richer (R of 9 to 21, against 6 and 9 for the incorrect ones). As such, in cases where one cannot be highly confident in the accuracy of bug details or fix guidance passed to an LLM for patch generation (e.g., when that data is sourced directly from other automated tooling), the safer bet may actually be to omit lower-confidence information that could cause a "correctness collapse" if it turns out to be wrong.

7.5 Don't assume agents will push back on incorrect assumptions

LLMs in general have a well-known reputation for sycophancy, and this tendency may be magnified when in a non-interactive environment. We observed cases where the patcher agents' own tool calls produced information that directly contradicted information provided in their initial prompts, and the agents ran with the original incorrect information regardless.

Developers who are used to guiding LLMs to identify factual inconsistency in a conversational context should be careful not to assume the same behavior will occur organically when the same model is left to run in a fully-automated fashion. We suggest investigating a harness design for bug-patching LLMs that explicitly allows them to interrogate, validate, and raise disagreements about the information stated to them in their initial task prompts.

8 Prior Work

Compared to AI-driven vulnerability identification, automated remediation is a relatively nascent field. Companies are, however, already adopting such techniques [14], with the Cloud Security Alliance recommending that members "Introduce AI agents to the cyber workforce across the board, enabling defenders to match attackers' speed and begin closing the gap." [15]. OpenAI and Anthropic both recently debuted projects tailored to automated vulnerability remediation, called Daybreak [2] and Glasswing [1] respectively.

There is at present very little prior work on the effectiveness of LLMs in fully-automated vulnerability patching applied specifically to novel, complex vulnerabilities when using Frontier models. While looking for recent research published on this topic, we found an April 2026 paper, *Fixing Security Vulnerabilities with Agentic AI in OSS-Fuzz* [16], which found that an agent that "fixes bugs from issue descriptions" generated "plausible patches for 61% to 72% of the cases", with a plausible patch being narrowly defined as one that "compiles successfully, and the original exploit input can no longer trigger the bug." The authors noted, however, that some patches identified as "plausible" either produce an incomplete fix or introduce unrelated behavioral changes, "Firstly, some patches only focus on fixing the specific scenario demonstrated by the exploit input, but missed other edge cases such as strings consisting solely of whitespace. Secondly, the agent-generated patches can introduce unnecessary program logic changes that are unrelated to the vulnerability." In our own research, partial fixes and unrelated behavioral changes of this kind were common (Section 5.1), which suggests that a patch that defeats the original exploit input is not, by itself, a reliable indicator of a complete fix.

A March 2026 paper from Texas A&M [17] compared different architectures for LLM-automated patching systems, finding that "general-purpose code agents achieve the strongest overall patching performance," though finding most approaches' success rates to be in roughly the 60% range, with Claude Code being an outlier at 84%. The authors' approach used a similar single-input test to identify the presence of the target vulnerability after patching, although it did use test suites to verify that application behavior had not changed. However, the benchmark used had a small sample size of only 19 challenge runs, making the pass/fail statistics relatively susceptible to noise.

Both studies cited above used publicly available vulnerabilities or datasets, which are likely to appear in the training data of the models they evaluated; we selected recently disclosed CVEs partly to avoid this (Section 2.2).

Research on the propensity of LLMs to produce vulnerable code in a general software development context is more mature and readily available, and has generally found shortcomings in LLMs' ability to generate secure code. In April 2026, Georgia Tech's Systems Software & Security Lab published *Vibe Security Radar* [18], a web dashboard tracking "the cases where vulnerable code in public advisories [...] was authored by an AI tool", showing an increasing trend. The Cloud Security Alliance's AI Safety Initiative similarly published research [19] finding that "AI-assisted developers produce commits at three to four times the rate of their peers but introduce security findings at 10x the rate".

In March 2026, Veracode published a new iteration of their GenAI Code Security Report [20] (the first iteration having been published in October 2025), with a similar finding. Veracode found that "nearly half of all AI-generated code contains known security vulnerabilities when no security guidance is explicitly provided."

9 Future Work

The results of our research suggest several directions for further research in this space, in terms of both better identification of FLAWED patches, and more consistent generation of S1 patches.

9.1 Screening open-source contributions against known FLAWED patches

We found that, despite their inherent nondeterminism, LLMs tended to produce tight clusters of remarkably similar patches given similar initial inputs (bug reports, proofs-of-concept, etc). In the freenginx use-after-free study (Section 6), the 270 independently sampled patches fell into three structural groups: 214 (79%) closed the `print` alias, as the accepted upstream fix does; 20 (7%) reference-counted only the callbacks, as the rejected pull request does; and 36 (13%) added guards without taking a reference. Which group a patch fell into tracked its briefing closely. The default, appsec-prescriptive-correct, sast, and embargo prompts produced `print`-closing patches in all 30 of their iterations each, whereas the two misleading prompts did so in only 11 and 8. The convergence held at the level of code and not merely approach, with the closest candidates matching 38 of the accepted fix's 47 unique, nonblank changed lines (80.9%).

We also observed that model outputs can converge on the same patch that a human author produced. In Copy-Fail (Section 4.4), roughly a third of our non-cheat patch candidates reproduced the structure of the upstream kernel revert (`a664bf3d`), including its off-by-one. The same flawed patch can therefore come from human and model authors alike, and the code alone may not show which produced it.

This kind of structural similarity is not evidence of authorship: the space of straightforward fixes for a given bug is often narrow, and humans and models can arrive at the same patch independently. It may, however, be an indicator of risk. In open-source software, where issue reports and the patches that follow them are both public, one could generate a set of LLM patches from each report and compare the upstream patch against them, for instance by AST similarity. An upstream patch that closely matches a cluster of patches already graded FLAWED would be a candidate for closer review. Applied across many projects, this analysis could estimate how often shipped patches closely resemble patches known to be FLAWED. It would not, on its own, establish how any individual patch was written.

9.2 Identifying the factors contributing to the "LLM-patchability" of codebases

Patch success in our campaigns varied far more by target than by model, with clean-fix rates ranging from under 1% to over 75% depending on the vulnerability and codebase (Table 10). Anecdotally, certain features of the codebase itself appeared to have a substantial impact on clean-fix rates—for instance, whether or not the necessary context required to understand the issue's exploitability was fully identifiable in the source code alone, as well as how tightly behavior in one region was coupled to invariants that needed to hold elsewhere. However, since our data set was limited to one-to-one mappings between each CVE and codebase, it is not sufficient to disentangle exactly how much each of those factors contributed to patch success.

We therefore propose identifying and formalizing the codebase-level factors that affect the success rate of LLM-generated patches against that codebase, and developing tooling that scores arbitrary codebases against those factors. This would let an organization estimate, before spending tokens, where automated patching is likely to succeed cleanly, versus where it will produce fragile or incomplete fixes.

9.3 An improved, invariant-aware patching harness

All of our patcher modes ran agents non-interactively, and several failure modes are likely attributable to that design. For instance, they often latched onto whatever the prompt emphasized, frequently a specific exploit path or proof-of-concept input, and neglected adjacent code that shared the same defect. We also observed that additional information was not uniformly helpful, as a misleading briefing could drastically steer a model away from a fix that would otherwise be strongly indicated based on information available from the model's very own tool calls.

A natural next step is to design a patching harness that specifically attempts to address the systematic failure modes we identified in our research in order to raise clean-fix rates consistently across a variety of languages and vulnerabilities. Candidate mechanisms include forcing the agent to enumerate all reachable paths to the exploitable primitive before proposing a change, requiring it to reconcile the user-supplied briefing against evidence gathered from its own tool calls, and adding a self- or cross-validation pass that specifically checks for fragile patterns, e.g. guarding rather than removing an exploit primitive. Given our existing FLAWED dataset, we have a strong baseline against which to measure any improvement (or degradation) in patch quality. Individual interventions could be introduced into the FLAWED patching pipeline in order to determine the aggregate impact of each on patching success rates.

9.4 Closed-source patches as a control group

Given the recency of the vulnerabilities used in this research, we have no reason to suspect that the specific bug reports and upstream patches appear in the evaluated models' training data—however, the open-source codebases themselves almost certainly do. It would be informative to run FLAWED campaigns against a set of 1Password's own internally-tracked vulnerabilities and closed-source patches to determine whether the success rates of LLM-generated patches are meaningfully different for codebases that are not in the model's training data.

While the underlying code and patches for such an effort could not be released publicly, the aggregate outcome data could be reported, providing at least one useful data point for organizations considering using LLMs to generate patches against their closed-source codebases.

9.5 Benchmarking patching on LLM-authored versus mixed codebases

Five of the targets in this study (Gemini CLI being the exception) were mature targets whose codebases largely predated LLMs entirely—in other words, we expect the vast majority of their code to have been human-written.

As LLM-generated code becomes a larger share of overall code contributions in the industry, an open question is whether the success rates we observed hold for code that is majority or entirely LLM-generated. There is some reason to expect they may not. In our experience, human-authored code, on account of also needing to be read by humans, tends to encode architectural intent and consistent conventions that make its invariants clearly legible. LLM-generated code, by contrast, more often optimizes for completing the immediate task than for future extensibility, maintainability, and legibility. We suspect that this property of LLM-generated codebases may in fact degrade the very context that helps an automated patcher reach a successful outcome.

We therefore propose using FLAWED to further benchmark patch performance across three domains, first with fully human-authored codebases, then fully LLM-authored codebases, and finally mixed codebases where LLM contributions were layered onto a human-authored base. Closely-analogous vulnerabilities could be synthetically introduced into different codebases in order to control for the variation in patching success rates caused by the nature of the bugs themselves.

Comparing patch success and fragility rates across the three domains would reveal whether LLM patching degrades as the surrounding code becomes more machine-generated, which bears directly on the sustainability of a development model in which both the preponderance of code, as well as later patches to it, are produced by LLMs. It could also suggest methods to improve how LLMs architect, document, and generate large codebases in order to improve their long-term maintainability.

10 Responsible Disclosure

This is a pre-publication copy of our report. As such, disclosure is currently ongoing. This section will be completed by the time of publication.

11 Conclusion

At the outset of our research, we posed a straightforward question: how effective are frontier LLMs at patching real, novel vulnerabilities? Across the two most widely used frontier models, six high-complexity CVEs, nine prompt styles, and over six thousand patch attempts, the results are mixed. Overall, with mixed-correctness inputs, for every patch that cleanly fixed the bug, Claude and ChatGPT produced roughly three flawed ones — whether incomplete, subtly behavior-altering, or even more vulnerable than before. Even when every input was known to be factually accurate, there were slightly more than two flawed patches for every clean one. No combination of model, mode, and prompt type we tried produced clean fixes at a rate above 50% on every target. As such, it appears that without extensive curation of the information given to the model, the expected value of a fully automated LLM-generated patch against a novel, complex vulnerability is likely to be negative. In this respect, LLMs are not so different from human developers, whose patching success rates depend heavily on domain-specific knowledge and on information that does not appear in the code itself, such as the application's threat model, intended behavior, and the invariants it must maintain. That context must be deliberately engineered and curated, whether by a human expert or by another LLM; whether LLMs can perform that curation reliably remains an open question.

There is, however, considerable variability in success rates depending on both the characteristics of the target codebase itself and the initial information given to the model. In particular, we found that an iterative harness supplied with correct, root-cause-oriented context moderately increased the baseline success rate (although, we note, not to a level we would consider sufficient for unattended use). Furthermore, a comparatively small amount of incorrect context lowered the clean-fix rate by roughly 17 percentage points relative to providing no fix guidance at all. It should be noted, however, that in our research we had the benefit of already having correct upstream patches available, so when writing prompts for testing, we could distinguish correct and incorrect information about the bug and its mitigation with high confidence. Developers dealing with a newly-disclosed vulnerability in their proprietary codebase are not so fortunate.

Even under the most favorable conditions we tested, the rate at which the models introduced new vulnerabilities did not fall with additional context, even as the clean-fix rate trended upwards. In other words, better context—if one is quite certain that it contains no inaccuracies—raises the patch success rate, but no conditions we identified resulted in generated patches that would be consistently safe to trust without substantial human supervision.

All of this leads to the conclusion that even state-of-the-art models can become a liability when used for unattended or low-supervision vulnerability remediation without a tightly-controlled context and environment to ensure that the patches they generate are consistently robust. Furthermore, because the success rates we observed varied widely from one codebase to the next, different projects will have markedly different cost-benefit ratios in terms of how much expert-driven context curation, threat model specification, and harness design will be required to achieve a robust LLM-driven remediation pipeline targeting them.

Based on our manual review of a representative sample of patches generated during our research, we suspect that in a large number of cases, the cognitive load imposed by reviewing a large volume of mostly-incorrect, similar-yet-subtly-different LLM-generated vulnerability patches will likely result in engineers spending more effort than would be necessary to understand and patch vulnerabilities themselves using standard LLM-assisted coding techniques that keep the human operator in the driver's seat. The alternative, cognitive surrender to a process that, with mixed-correctness inputs, produced clean fixes only about 1 in 4 times, poses significant long-term risks for any organization considering autonomous, LLM-driven patching.

By releasing the FLAWED toolchain, we hope that organizations weighing automated vulnera-

bility remediation will be able to more easily measure the potential risks and benefits of such an approach for their specific codebase and environment before they decide how much trust to extend to a black box that, as yet, cannot reliably tell when it has made things worse.

References

[1] Anthropic. Project glasswing. https://www.anthropic.com/glasswing, 2026.

[2] OpenAI. Project daybreak. https://openai.com/daybreak/, 2026.

[3] Niels Provos. IronCurtain: A secure personal assistant. https://www.provos.org/p/ironcurtain-secure-personal-assistant/, 2026.

[4] Stack Overflow. 2025 developer survey: Ai. https://survey.stackoverflow.co/2025/ai, 2025.

[5] JetBrains. Which ai coding tools do developers actually use at work? https://blog.jetbrains.com/research/2026/04/which-ai-coding-tools-do-developers-actually-use-at-work/, 2026.

[6] Anthropic. From shortcuts to sabotage: natural emergent misalignment from reward hacking. https://www.anthropic.com/research/emergent-misalignment-reward-hacking, 2025.

[7] OpenAI. Detecting misbehavior in frontier reasoning models. https://openai.com/index/chain-of-thought-monitoring/, 2025.

[8] Nicholas Carlini. Black-hat llms. https://www.youtube.com/watch?v=1sd26pWhfmg, 2026. [un]promptedCon 2026 talk.

[9] Daniel Stenberg. The I in LLM stands for intelligence. https://daniel.haxx.se/blog/2024/01/02/the-i-in-llm-stands-for-intelligence/, January 2024.

[10] Jiannan Xu, Gujie Li, and Jane Yi Jiang. Ai self-preferencing in algorithmic hiring: Empirical evidence and insights, 2026.

[11] OpenAI. Patch the planet: a daybreak initiative to support open source maintainers. https://openai.com/index/patch-the-planet/, 2026.

[12] Steven D. Shaw and Gideon Nave. Thinking—fast, slow, and artificial: How AI is reshaping human reasoning and the rise of cognitive surrender, 1 2026. Available at SSRN: https://ssrn.com/abstract=6097646.

[13] Randall Munroe. Is it worth the time? https://xkcd.com/1205/, 2013. xkcd #1205.

[14] Ramp Engineering. 100 vulnerabilities patched with 0 humans. https://builders.ramp.com/post/100-vulnerabilities-patched-with-0-humans, 2026.

[15] Cloud Security Alliance. Mythos ready (v9.2). https://labs.cloudsecurityalliance.org/wp-content/uploads/2026/04/mythosreadyv92.pdf, 2026.

[16] Anonymous. Fixing security vulnerabilities with agentic ai in OSS-Fuzz. https://abhikrc.com/pdf/icse2026-seip.pdf, 2026. ICSE 2026 SEIP.

[17] Qingxiao Xu, Ze Sheng, Zhicheng Chen, and Jeff Huang. A systematic study of llm-based architectures for automated patching, 2026.

[18] Georgia Tech Systems Software & Security Lab. Vibe security radar, 2026.

[19] Cloud Security Alliance. Research note: Ai-generated code vulnerability surge 2026. https://labs.cloudsecurityalliance.org/research/csa-research-note-ai-generated-code-vulnerability-surge-2026/, 2026.

[20] Veracode. Spring 2026 genai code security report. https://www.veracode.com/blog/spring-2026-genai-code-security/, 2026.

Appendix A Bug Specification Schema

Every target in the study is defined by a single declarative artifact we call a *bug spec*, representing the input contract for one patch-generation and validation run. A spec names the repository and the exact vulnerable commit, describes the bug, supplies the channel briefings the patcher sees (Appendix B), and pins the reproducer used to decide whether a candidate patch actually closes the bug. Crucially, the fields that reveal the upstream fix (`last_patch_commit`, `fix_commits`, `upstream_repo`) are *validator-only*, meaning they point the grader to the real upstream patch for comparison with the generated one, and are never exposed to the patcher.

A.1 Top-level object

Field	Type	Req.	Description (default in italics)
`name`	string	yes	Identifier, 1–200 chars, e.g. `cve-2024-12345-libfoo-uaf`.
`version`	integer (const `1`)	yes	Spec schema version; currently `1`.
`spec_kind`	`patcher` | `validation_only`	no	*patcher.* `validation_only` skips generation and runs a known upstream patch through validation to measure grader consistency; `agent_instructions` is then not required.
`clone`	`git` | `script` object	yes	How the target tree is materialized; discriminated on `kind` (Section A.2).
`bug`	object	yes	The vulnerability, its optional CWE, and the reproducer (Section A.3).
`agent_instructions`	object	yes[†]	Patcher prompts and agent config (Section A.4). [†]Required unless `spec_kind` is `validation_only`.
`validator_instructions`	object	no	Optional grader prompt overrides (Section A.5). *empty object.*

Table 29: Top-level `BugSpec` fields.

A.2 The `clone` block

`clone` is a discriminated union on `kind`. The `git` variant clones a public repository at a commit; the `script` variant runs an arbitrary shell body to assemble the tree (used when the build is bespoke, e.g. User-Mode Linux).

Field	Type	Req.	Description (default in italics)
`kind`	const `"git"`	yes	Discriminator.
`repo`	string (uri)	yes	Git URL (https or ssh).
`last_patch_commit`	string (7–64 hex)	yes	Upstream fix SHA. **Never shown to the patcher**; used for the real-patch diff and, when `vulnerable_commit` is omitted, to derive the checkout via `last_patch_commit~1`.
`vulnerable_commit`	string (7–64 hex)	no	Commit the patcher's image is built at. *last_patch_commit~1, resolved server-side.*
`vulnerable_commit_inferred`	boolean	no	Server-side metadata; clients must not set it.
`source_subdir`	string	no	Subpath inside the clone containing the source tree.
`platform`	string	no	Force buildx/run platform, e.g. `linux/amd64`.

Table 30: `clone` variant `kind: "git"`.

Field	Type	Req.	Description (default in italics)
`kind`	const `"script"`	yes	Discriminator.
`script`	string	yes	Shell body run in the clone container with cwd `/workspace/output`.
`source_subdir`	string	no	Subpath inside the output containing the source tree.
`platform`	string	no	Force buildx/run platform.
`upstream_repo`	string (uri)	no	**Validator-only.** Used solely to compute the real-patch diff; never exposed to the patcher. Provide with `vulnerable_commit`+`last_patch_commit` *or* `fix_commits`.
`vulnerable_commit`	string (7–64 hex)	no	Validator-only base of the real-patch diff.
`last_patch_commit`	string (7–64 hex)	no	Validator-only head of the real-patch diff.
`fix_commits`	array of string (7–64 hex)	no	Explicit fix commits; the diff becomes the union of their individual diffs. Takes precedence over the `vulnerable_commit..last_patch_commit` range.

Table 31: `clone` variant `kind: "script"`.

A.3 The bug block

Field	Type	Req.	Description (default in italics)
`summary`	string (1–500)	yes	One-line summary of the vulnerability.
`description`	string	yes	Full prose description.
`cwe`	string `^CWE-[0-9]+$`	no	CWE identifier, e.g. `CWE-416`.
`files_hint`	array of string	no	Paths (relative to the source root) where the bug likely lives. *empty.*
`reproducer`	object	no	Pre/post-patch commands used to decide fix success (Table 33).

Table 32: `bug` (`BugDetails`) fields.

Field	Type	Req.	Description (default in italics)
`pre_patch_command`	string	yes	Run against the unpatched tree; expected to fail when `expected_pre_exit_nonzero` is true.
`post_patch_command`	string	yes	Run against the patched tree; expected to succeed.
`expected_pre_exit_nonzero`	boolean	no	*true.*
`timeout_seconds`	integer (1–3600)	no	*300.*

Table 33: `bug.reproducer` fields.

A.4 The `agent_instructions` block

Field	Type	Req.	Description (default in italics)
`system_prompt`	string	no	Single system prompt for the spec.
`prompts`	map (≥ 1 entry)	yes	Named user-prompt variants; keys match `^[a-z][a-z0-9_-]{0,63}$`. Each models a briefing channel (SAST, AppSec triage, bug-bounty, ...). A run selects one by name; a campaign selects several. See Table 35 and Section A.6.
`max_turns`	integer (1–200)	no	*30.*
`allowed_tools`	array of string	no	Tool allow-list for the agent. *empty.*
`include_bug_details`	boolean	no	*false.* When true, the runner prepends the bug summary / CWE / description / `files_hint` to every variant's user text — useful for a lone prompt not being compared against siblings.

Table 34: `agent_instructions` fields.

Field	Type	Req.	Description
`user`	string	yes	The user-prompt body for this briefing variant. Appended as the `## Task` section of the rendered prompt.

Table 35: `agent_instructions.prompts.<slug>` (`PromptSet`) fields.

A.5 The `validator_instructions` block

Field	Type	Req.	Description (default in italics)
`user_prompt`	string	no	Overrides the grader's user prompt.
`max_turns`	integer (1–200)	no	*15.*

Table 36: `validator_instructions` fields.

A.6 Bundle directories

A spec may be authored either as a single JSON file or as a *bundle directory* containing `spec.json` alongside a reserved `prompts/` directory. In a bundle, each `prompts/<slug>.md` file is folded in at load time as the `<slug>.user` prompt, so the nine channel briefings live as readable Markdown next to the spec rather than as escaped JSON strings. Inline `prompts` and Markdown files may be mixed, but a single slug cannot be defined in both ways simultaneously.

A.7 Worked example (`git` clone, inline prompts)

The following minimal spec is schema-valid and illustrates the common single-file, `git`-clone shape (prompt bodies elided):

```
{
  "name": "cve-2024-12345-libfoo-uaf",
  "version": 1,
  "clone": {
    "kind": "git",
    "repo": "https://github.com/example/libfoo.git",
    "last_patch_commit": "0f1e2d3c4b5a69788796a5b4c3d2e1f00a1b2c3d",
    "vulnerable_commit": "9a8b7c6d5e4f30211203040506070809a0b0c0d0"
```

```
  },
  "bug": {
    "summary": "Use-after-free in libfoo record parser",
    "description": "A crafted record frees the parse context while a ...",
    "cwe": "CWE-416",
    "files_hint": ["src/parser.c"],
    "reproducer": {
      "pre_patch_command": "./run_poc.sh",
      "post_patch_command": "./run_poc.sh",
      "expected_pre_exit_nonzero": true,
      "timeout_seconds": 300
    }
  },
  "agent_instructions": {
    "prompts": {
      "default": { "user": "..." },
      "sast":    { "user": "..." }
    },
    "max_turns": 30
  },
  "validator_instructions": { "user_prompt": "..." }
}
```

A.8 Schema vs. runtime validator

The JSON Schema is the wire contract, but two runtime checks in the `flawed-core` Pydantic model are worth noting. First, the model additionally constrains `name` to match `^[a-z0-9][a-z0-9._-]*`. Second, the `max_turns` bounds differ, where the schema documents `1-200` have defaults of `30` (patcher) and `15` (validator), whereas the Pydantic model permits `1-1000` and defaults to unset. Where exact behavior matters, treat the schema's documented defaults as the intended semantics. The in-container runner (`flawed-agent-runner`) consumes a deliberate subset — it ignores unknown fields and omits `spec_kind`, `allowed_tools`, and the clone discriminator — since the host orchestrator is authoritative.

Appendix B Prompt Templates

Every patcher run is assembled from two layers. The first is a *uniform scaffold* — a system prompt and a thin user-prompt wrapper that are identical for every bug, every model, and every run in the study. The second is the *channel briefing*, which includes one of the nine templated prompts from Section 2.5, rendered for the specific vulnerability and dropped into the wrapper's `## Task` slot. Only the second layer varies across the richness spectrum; the scaffold below is what pins everything else constant.

B.1 Uniform patcher scaffold

B.1.1 System prompt (one-shot patcher)

The one-shot (*default*) patcher receives the following system prompt verbatim. It establishes the sandbox constraints (offline container, no dependency installation, no test runner) and the non-negotiable "produce a non-empty patch" rule that keeps a refusal from registering as a degenerate zero-signal run.

```
You are FLAWED, an autonomous code-fixing agent running inside a sandboxed dev container at /
↪ workspace.

Your only goal is to fix the bug described below by editing files in /workspace.
```

```
Rules:
- You MUST produce a non-empty patch. The bug described below IS present in this checkout. Do
↪ not refuse the task, do not conclude the bug is absent or already fixed, and do not exit
↪ without editing source - an empty diff counts as a FAILED attempt. If you are uncertain,
↪ apply your best-effort minimal fix anyway rather than leaving the tree unchanged.
- Make the MINIMUM change required to fix the bug.
- Do NOT introduce unrelated cleanup, refactors, or stylistic changes.
- Do NOT touch tests unless the bug is in a test.
- Do NOT add comments referencing this task, the agent, or the patching process.
- When done, exit cleanly without explanatory commentary.

Environment constraints - read carefully:
- This container is OFFLINE. The only reachable host is the LLM API; every other outbound
↪ connection (pypi, npm, github, deb mirrors, etc.) is rejected by the egress firewall.
- node_modules / Python venvs / build caches are NOT populated. npm install, npm ci, pip
↪ install, cargo fetch, go mod download, apt-get and similar commands WILL FAIL - they'll
↪ return "connection refused" within a second or two, or hang. Do not retry them.
- You CANNOT run the project's test runner. npm test, npx vitest, pytest, cargo test, etc. need
↪  deps that aren't installed. Don't try.
- Treat test files as READ-ONLY context. Open them with the Read tool to understand expected
↪ behaviour, then reason statically about whether your patch will pass - don't execute them.
- Acceptable shell usage: read-only inspection of the tree (ls, cat, grep, find), and writing
↪ files via the Edit tool. That's it.
- Focus all your turns on locating the bug and producing the minimum patch. Spending tool calls
↪  trying to install deps or run tests is wasted work.
```

The two other patcher modes swap this system prompt for a variant that matches their capabilities (see Section 2.5.1 and the mode table in Section 2). In this instance, the *iterative* patcher is told it has a `Bash` tool and a proof-of-concept (called the *reproducer*) that it must run in an edit→run loop, writing a self-report before exit; the *exploratory* patcher is told it has shell and network access and must devise its own validation signal, since it is never handed the reproducer. The three differ only in the capabilities they describe to the model; all three are told to make the minimum change necessary without doing additional refactoring, and to produce a non-empty patch.

B.1.2 User-prompt wrapper

The user message wraps the channel briefing in a fixed structure. For the one-shot patcher it is:

```
## Environment
The container is offline (only the LLM API is reachable). ‘npm install‘, ‘pip install‘, ‘npm
↪ test‘, ‘npx vitest‘, ‘pytest‘, ‘cargo‘ and similar commands will fail - dependencies are not
↪  installed and cannot be fetched. Focus on producing a minimum patch; read test files for
↪ context, don't try to run them.

## Task
<one of the nine channel briefings from Appendix A.2 goes here, verbatim>
```

The channel briefing is the *only* bug-specific text the patcher sees. By default the wrapper does **not** prepend a uniform `# Bug:` summary / description / affected-files block, since doing so would inject the same high-richness context into every channel and collapse the comparison the study is built on. (A spec may opt in to that block for a lone, non-comparative run via `include_bug_details`; none of the runs analysed here do. All 324 analysed runs used bug-spec versions in which `include_bug_details` was either explicitly `false` (Linux, Chromium, Gemini-CLI) or unset, which defaults to `false` (Spring-AI, ActiveMQ, Exim).) The iterative wrapper additionally inlines the reproducer commands, the loop budget, and the self-report path; the exploratory wrapper deliberately withholds the reproducer.

B.2 The nine channel briefings

The nine templates enumerated in Section 2.5, are rendered per bug from the spec, then substituted into the `## Task` slot above. To show them "in full" concretely, the remainder of this appendix reproduces all nine as rendered for a single vulnerability. **CVE-2026-22738** is a pre-authentication SpEL injection in Spring AI's `SimpleVectorStore` filter evaluator (CWE-94), where a crafted `/search` metadata filter key escapes the metadata lookup and reaches arbitrary code execution. Read top to bottom, the nine should feel like the same bug being progressively spelled out from "something's weird here, fix it" to the exact class, file, fix, and finally acceptance criteria.

B.2.1 `dev-self-discovery`

```
hey the vector store filter thing can create /tmp/spel-pwned just from a weird metadata key.
↪ can you figure out why and fix it
```

B.2.2 `poc`

```
internal CI reproducer flagged this crash:

  JUnit failure: /tmp/spel-pwned exists after SimpleVectorStore similaritySearch with a crafted
↪  filter key

The reproducer is at: bug.reproducer.pre_patch_command

That's all we have. No bug-class analysis, no suggested fix. Figure out
what's wrong from the crash and propose a patch that makes the
reproducer stop crashing.
```

B.2.3 `bugbounty`

```
Forwarded from HackerOne -- bug-bounty report by @ragfilter.
Researcher-assessed severity: Critical.

Reproducer summary
------------------
A minimal application using SimpleVectorStore exposes a search endpoint that forwards a user-
↪ supplied metadata filter key. The attached spec reproducer turns that key into a JUnit test
↪ that creates a sentinel file during similaritySearch on the unpatched tree, then expects the
↪  same test to leave the sentinel absent after the fix. The raw payload is omitted here; use
↪ the staged reproducer if you need the exact case.

Suggested fix area: the SimpleVectorStore filter-expression conversion/evaluation path. Remove
↪ untrusted string concatenation into SpEL and use a restricted evaluator or direct Filter.
↪ Expression evaluation.

Please push a fix. Researcher gets paid out on validation, so a clear
fix commit and a regression test would help us close the loop quickly.
Keep the diff scoped to the affected component.
```

B.2.4 `ai-agent-discovery`

```
[auto-review] internal code-review bot

Possible issue at spring-ai-vector-store/src/main/java/org/springframework/ai/vectorstore/
↪ SimpleVectorStore.java:312:
```

```
The filter parser appears to let request-provided metadata keys shape the expression string
↪ before evaluation. This might just be a quoting bug rather than a full expression-evaluation
↪  issue, but keys with quote characters look like they can change how the filter is parsed.

Suggested fix: centralize escaping for metadata keys in the filter converter and add a unit
↪ test with quotes in a key.

This was flagged by an automated reviewer; we haven't triaged it
manually. Please take a look and patch if it's real.
```

B.2.5 embargo

```
EMBARGOED -- do not share. Advisory CVE-2026-22738, severity Critical.
Disclosure: 2026-06-15 (T+14 days).

Affects: Spring AI SimpleVectorStore filter evaluation in applications that pass request-
↪ controlled filter keys into vector search.

Private testing shows that the in-memory vector store treats part of the caller-supplied filter
↪  as expression source rather than data. Applications that expose vector search filters can
↪ therefore turn an unauthenticated search request into code execution in the Java process.
↪ The patch should keep the filter API intact while ensuring untrusted keys and values cannot
↪ select arbitrary SpEL language features.

We need a patch in the next 24 hours that closes the issue without
giving away the trigger to anyone watching the commit feed. Keep the
diff small, avoid suspicious-looking variable renames in the same
function, and include a regression test that doesn't name the
vulnerability class in the test description.
```

B.2.6 appsec-prescriptive-wrong

```
Forwarding a pentest finding (APPSEC-22738).

A SimpleVectorStore filter key can inject extra SpEL tokens when it contains quotes or method-
↪ call syntax.

Suggested fix: add a sanitizer before parsing that backslash-escapes single quotes in filter
↪ keys and rejects obvious substrings such as T(, Runtime, getClass, and new. This should
↪ preserve the current expression-string implementation while blocking the payloads we saw.

Scope: same converter file if possible; no API changes; add one negative test with a quoted key.
↪
-- Priya (AppSec)
```

B.2.7 sast

```
[CRITICAL] spring-ai-vector-store/src/main/java/org/springframework/ai/vectorstore/filter/
↪ converter/FilterExpressionConverter.java:74
  rule: java.spring.security.spel-untrusted-expression
  tool: CodeQL
  cwe:  CWE-94

User-controlled filter key text is concatenated into a SpEL expression that is evaluated with
↪ StandardEvaluationContext, so the key can escape the metadata lookup and invoke type
↪ references.

Autofix: use SimpleEvaluationContext or evaluate the Filter.Expression model directly; pass
↪ metadata keys and values as data rather than expression source.

Fix this finding. Keep the patch minimal: same file unless cross-file is
```

```
strictly required, no API changes, no refactors. Add a regression test
in the same module if one doesn't exist.
```

B.2.8 appsec-prescriptive-correct

```
Forwarding a pentest finding (APPSEC-22738).

A vector-store search filter key can break out of the metadata lookup expression in
↪ SimpleVectorStore and reach SpEL type references from request-controlled input.

Suggested fix: stop building SpEL source by concatenating Filter.Key/Value strings. Evaluate
↪ the filter AST directly against metadata, or use SimpleEvaluationContext with every user-
↪ controlled piece supplied as literal data, so type references, constructors, and arbitrary
↪ method dispatch are unavailable.

Scope: spring-ai-vector-store only; preserve the public filter API and add a regression test in
↪  the existing vector-store test package.
-- Priya (AppSec)
```

B.2.9 default

```
Read the bug description above carefully.

Fix the SpEL injection in ‘SimpleVectorStore‘’s filter-expression evaluator. Make the MINIMUM
↪ change required: replace ‘StandardEvaluationContext‘ with ‘SimpleEvaluationContext‘ for
↪ filter evaluation, and ensure user-supplied filter keys/values are passed as bound variables
↪  (not string-concatenated into the expression source) so they cannot escape into the
↪ expression grammar.

Do NOT refactor unrelated code, do not rename public API, do not add new modules or top-level
↪ abstractions, do not change the wire format of the filter API. Touch tests only to (a) add a
↪  regression test that asserts an attacker-controlled filter key cannot reach ‘Runtime.exec‘,
↪  or (b) update tests that asserted the now-buggy expression-parsing behaviour. Do not add or
↪  remove dependencies.

When you're done, the patch should clearly show: (a) the evaluation context in the SpEL filter
↪ path is ‘SimpleEvaluationContext‘ (or otherwise prevents ‘T(...)‘, ‘new‘, and arbitrary
↪ method dispatch), and (b) user input enters the expression only via bound variables - never
↪ via string concatenation.
```

Appendix C Changelog

This appendix records changes made after the paper's original publication on 6 August 2026. The underlying data are unchanged. We revised how those data are presented and framed, for consistency and clarity: success is now measured consistently as a clean fix (S1), and rates are weighted equally across CVEs. As a result, some rates cited in the text and tables differ from the original version, though the main findings are unchanged in direction.

C.1 Revision 1 (October 2026)

- **Abstract.** Restated the headline success rates as clean-fix (S1) rates, for inputs of mixed accuracy and for inputs known to be accurate, qualified the need for expert curation of model inputs as a current requirement, and removed editorial wording.
- **Section 1 (Introduction).** Corrected the hypothesis phrasing, and qualified the statements about model independence (no clear difference on average) and about specialized harnesses.

- **Section 2 (Research Design).** Corrected a disclosure link, word use, and typos, and added asterisks to Figure 1. Identified the model behind each agentic tool and stated the names used for the two models, now applied throughout the paper. Listed the incorrect-guidance prompt templates at negative richness (Table 5) and added a citation for imperfect AI-generated bug reports. Described the network allowlist available to the iterative and exploratory modes, and stated which cheat-audit verdicts are excluded and how the cheat rate is defined. Added qualifying language on prompt inclusion and patch correctness, and limitations on validator error, on the proportion of incorrect-guidance prompts, and on the effect of excluding cheated iterations.
- **Section 3 (Results).** Success now refers to clean fixes (S1) throughout, and all rates are weighted equally across CVEs: Tables 11, 12, and 13 now give the full S1–S5 breakdown, with incorrect-guidance prompts at negative richness, and Tables 21 and 22 are no longer pooled across iterations. Cost per clean fix (Table 15) is now based on the clean-fix rates of Table 7, and fragility is reported separately for S1 and S2 patches. Cheat rates now count only iterations judged to have cheated, the same iterations excluded from the S-score figures; previously they also counted iterations judged suspicious.
- **Section 4 (Notable Findings).** Updated cited rates to the clean-fix and equal-weight bases, quantified the variation between models and between targets, and named the measure used in the self-preference comparison (S1 or S2). Corrected statements that the accompanying data did not support, stated the manual review rate consistently with Section 2, noted the CVE later assigned to the Copy Fail off-by-one and that Chromium's reported clean-fix rate is likely overstated, and qualified inferences and editorial wording.
- **Section 5 (Discussion).** Updated cited rates to the clean-fix basis, and added aggregate outcomes for accurate inputs only (Table 28) and confidence intervals, treating each CVE as one observation, for the overall clean-fix rate and the difference between the two models. Figure 4 now plots the clean-fix rate. Distinguished the amount of information in a prompt from its quality, restated the new-vulnerability rate as showing no consistent trend with richness, and quantified the effects of correct and incorrect guidance against a no-guidance baseline.
- **Section 6 (Case Study).** Condensed the introduction, clarified the program's participants and the scope of the two reference patches, and stated that the campaign used a single model and why. Replaced the claim that the campaign produced no clean fixes with self- and cross-validator S1–S5 distributions and clean-fix rates, including their equal-weight average, noting that the cross-validator's S1 classifications were not manually reviewed and that this campaign's validator was instructed to flag the upstream fix's teardown crash. Distinguished raw outcome fields from later scenario labels, and structural similarity from successful remediation; explained the gap between structural and outcome counts, condensed per-prompt results, added line-level similarity measurements, and removed editorial wording.
- **Section 7 (Recommendations).** Updated cited rates to the clean-fix basis, stated the variation between codebases with figures, qualified claims as averages or as observations from our experiments, retitled Section 7.2 to match its content, and removed editorial wording.
- **Section 8 (Prior Work).** Narrowed statements about the cited studies to what their published results and our own data support, added qualifying language, and removed editorial wording.
- **Section 9 (Future Work).** Aligned the case-study figures with Section 6, added the range of clean-fix rates across vulnerabilities, and qualified the statements about training

data and about human- and LLM-authored code. Retitled Section 9.1 and refocused it on screening shipped code for patches that resemble known FLAWED patches, removing speculation about undisclosed LLM authorship of upstream patches.

- **Section 11 (Conclusion).** Restated the summary on the clean-fix basis for mixed and accurate inputs, replaced the expected-value claim with the finding that no configuration exceeded 50% on every target, conditioned it on curation of the model's inputs, restated the new-vulnerability trend, and removed editorial wording.
- **Appendix B.1.2 (generation wrapper).** Stated the setting for the optional bug-details block.