

How Safe Are AI-Generated Patches? A Large-scale Study on Security Risks in LLM and Agentic Automated Program Repair on SWE-bench

AMIRALI SAJADI, Drexel University, USA

KOSTADIN DAMEVSKI, Virginia Commonwealth University, USA

PREETHA CHATTERJEE, Drexel University, USA

Large language models (LLMs) and their agentic frameworks are increasingly adopted to perform development tasks such as automated program repair (APR). While prior work has identified security risks in LLM-generated code, most have focused on synthetic, simplified, or isolated tasks that lack the complexity of real-world program repair. In this study, we present the first large-scale security analysis of LLM-generated patches using 20,000+ GitHub issues. We evaluate patches proposed by developers, a standalone LLM (Llama 3.3 Instruct-70B), and three top-performing agentic frameworks (OpenHands, AutoCodeRover, HoneyComb). Finally, we analyze a wide range of code, issue, and project-level factors to understand the conditions under which generating insecure patches is more likely.

Our findings reveal that Llama introduces many new vulnerabilities, exhibiting unique patterns not found in developers' code. Agentic workflows also generate a number of vulnerabilities, particularly when given more autonomy. We find that vulnerabilities in LLM-generated patches are associated with distinctive code characteristics and are commonly observed in issues missing specific types of information. These results suggest that contextual factors play a critical role in the security of the generated patches and point toward the need for proactive risk assessment methods that account for both issue and code-level information.

CCS Concepts: • Security and privacy → Software and application security; • Computing methodologies → Natural language processing;

Additional Key Words and Phrases: LLMs, Agentic Frameworks, Security, Automated Program Repair

ACM Reference Format:

Amirali Sajadi, Kostadin Damevski, and Preetha Chatterjee. 2018. How Safe Are AI-Generated Patches? A Large-scale Study on Security Risks in LLM and Agentic Automated Program Repair on SWE-bench. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 25 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

LLMs are rapidly transforming software development, assisting with tasks like code completion, test generation, and bug fixing [6, 41, 71, 83, 89]. They are also increasingly integrated into agentic frameworks that enable autonomous planning and execution of complex tasks, such as automated program repair (APR), where LLM-based agents identify, diagnose, and patch software bugs directly in real-world projects [81, 86, 88]. These systems have significantly outperformed traditional APR approaches and are now used to solve repairs across numerous real-world software projects, spanning thousands of issues from production-grade libraries and frameworks. This widespread adoption of LLMs in APR

Authors' Contact Information: **Amirali Sajadi**, amirali.sajadi@drexel.edu, Drexel University, Philadelphia, Pennsylvania, USA; **Kostadin Damevski**, kdamevski@vcu.edu, Virginia Commonwealth University, Richmond, Virginia, USA; **Preetha Chatterjee**, preetha.chatterjee@drexel.edu, Drexel University, Philadelphia, Pennsylvania, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

systems raises critical concerns about their potential security implications [58, 65, 73, 87]. Studies show that 30-50% of code generated with LLM-based code assistants contains vulnerabilities [20, 57, 80]. When embedded within complex automated program repair frameworks, LLM-generated patches are often pushed directly into CI/CD pipelines, where a single insecure repair can rapidly propagate across dependent services and compromise entire software stacks.

While several studies have shown that LLMs are prone to generating insecure or low-quality code, most focus on simplified tasks such as coding exercises or identifying vulnerabilities in isolated code snippets [35, 51, 57, 72]. These scenarios lack the complexity and context of program repair in real projects, where understanding environment constraints or project-specific dependencies is critical. Moreover, such scenarios use LLMs as generic code generators rather than APR systems that produce targeted edits for specific bugs. APR agents also operate in a different setting than ordinary code agents. Code AI agents generate code from a short user prompt, while APR agents take a bug report, inspect the current repository state, and then propose candidate patches to fix a specific bug [81, 88]. Hence, assessing their security requires evaluation frameworks that capture the full end-to-end repair workflow rather than isolated snippet-based analyses.

Although recent benchmarks like SWE-bench [32] have enabled the assessment of LLMs on real-world software engineering tasks, security-specific analyses in these contexts remain limited, and we still lack a systematic understanding of when APR-style LLM and agentic frameworks introduce new vulnerabilities, or how those vulnerabilities relate to properties of the patches and the surrounding development context. Recent studies have demonstrated that adversarial bug reports can mislead APR agents into producing insecure yet functionally correct patches [12, 59]. These efforts evaluate security risks only assuming adversarial intent and on relatively small datasets. No prior work has performed a large-scale and security-focused empirical analysis of patches generated by LLM and agentic APR systems, especially assessing if vulnerabilities in patches arise even without adversarial intent.

Using SWE-bench, we analyze patches generated by Llama 3.3 and three state-of-the-art agentic APR frameworks, as well as the developer-written fixes for the same issues. We first quantify how often these systems introduce new vulnerabilities and which CWE types they tend to produce. We then examine how code-level, issue-level, and project-level factors influence the security of the proposed solutions, identifying conditions that result in vulnerability-inducing AI-generated patches. Our overarching goal with this work is to identify when and where APR systems are most likely to generate vulnerable patches, informing guidelines for when to trust their output and when to require additional checks. Overall, these analyses provide, to our knowledge, the first comprehensive evidence of the conditions and the frequency with which APR systems introduce security vulnerabilities when resolving real GitHub issues. Specifically, we ask the following research questions:

- **RQ1:** How secure are the patches produced by Llama 3.3 on real-world issue resolution tasks? How does their security compare to developer-written patches?
- **RQ2:** To what extent do agentic APR workflows generate secure patches in real-world settings?
- **RQ3:** What code, issue, or project characteristics are associated with the generation of vulnerable code by LLMs or agentic APR frameworks when resolving issues?

Our findings highlight the security risks of LLM-generated code at scale. When evaluated on 20,000+ GitHub issues, the standalone Llama introduced 135 new vulnerabilities i.e., nearly 11x higher than the 12 new vulnerabilities introduced by developers. On a smaller test set of 2,200+ issues, the agentic frameworks also produced vulnerabilities, with the highest counts observed in the framework that gave the LLM greatest autonomy. Notably, some of the most recurring vulnerabilities introduced by LLMs (CWE-95, CWE-502) were distinct from those written by developers

(e.g., CWE-732, CWE-377), suggesting that LLMs are not only replicating developers’ errors but also exhibiting new behavioral patterns. We found that vulnerable outputs are more likely when LLM patches involve a higher number of files and unique file type modifications. Further, most standalone LLM-generated vulnerabilities were observed in instances involving bug-related issues that do not contain code examples, information about the expected behavior of the code, or steps to reproduce the issue. These findings suggest that the risk of generating vulnerabilities is not only a function of the code produced but also of the context in which the LLM operates. As such, our insights can help inform proactive risk assessments of LLM and agent contributions, complementing the existing vulnerability detection tools, by *highlighting high-risk repair tasks even before the code is generated or reviewed*.

2 Methodology

Figure 1 shows an overview of our methodology. Using GitHub issues from SWE-bench, we collect developer patches and candidate patches generated by a standalone LLM (Llama 3.3 [50]) and three agentic frameworks (OpenHands [81], AutoCodeRover [88], HoneyComb [30]). We analyze the prevalence and patterns of vulnerabilities for each of these code generation sources (RQ1, RQ2). In addition, we identify code, issue, and project-level factors that are associated with the presence of these vulnerabilities (RQ3). Our data and research artifacts publicly accessible through our replication package to support further research and validation of our findings [3].

2.1 Dataset and Task Setup

We leverage the SWE-bench dataset [32], which consists of real-world GitHub issues paired with pull requests that resolve them. SWE-bench includes a *train set* of approximately 19,000 issue–PR pairs collected from 37 popular Python repositories and a *test set* of 2,294 issue–PR pairs from 12 repositories.

For the standalone model, we use Llama 3.3 Instruct (70B) to generate patches for both the train and test sets of SWE-bench, comprising approximately 21,000 instances in total. Since Llama is open-source and possible to run at scale, this generation step, although computationally heavy, is feasible. In contrast, for the agentic frameworks, we rely on publicly released patches that are available only for the test set. Due to the substantial computational and financial cost required to run these iterative frameworks across the train set, we restrict our framework-based analysis to the test split.

Including both standalone LLM and agentic framework settings ensures the correct scope for addressing our research questions (RQ1–RQ2). These two settings also reflect distinct real-world usage scenarios. Standalone LLMs can better represent how developers interact with models, prompting them to get help with issue resolution and manually integrating the generated code. This makes the RQ1 analysis relevant for understanding the security implications of everyday developer usage. Agentic frameworks, in contrast, embody more autonomous approaches to issue resolution and reflect a growing trend toward automated, end-to-end program repair. Evaluating both settings enables a broader understanding of the security risks posed by current and emerging LLM-based workflows. Note that although any LLM-generated code can be further revised by developers, in line with SWE-bench and related work, our evaluation isolates the LLM’s own contributions. This allows us to assess the baseline security performance of the models and agents “out of the box,” without confounding effects from human review or refinement.

2.2 Patch Generation with LLM and Agentic Frameworks

2.2.1 RQ1. Llama 3.3 Instruct (70B). To examine the security risks of patches produced by a standalone LLM, we generate candidate patches across the entire dataset (train+test) using the Llama 3.3 Instruct 70B model [50]. Recent

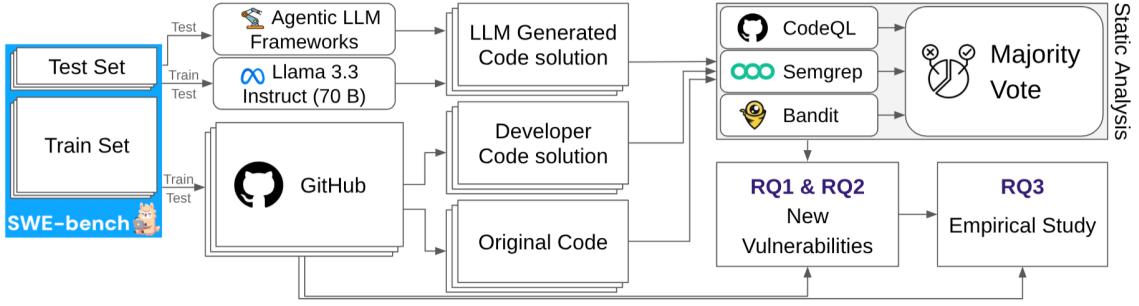


Fig. 1. Overview of Our Methodology for LLM-Based Code Generation and Security Evaluation.

benchmarks indicate that this model’s code generation capabilities are comparable to those of GPT-4o [56] and exceed the performance of Claude 3 Opus [8] and GPT-4 Turbo [55], making it a strong choice for our study [13, 78]. We select this specific model over the older Code Llama family because published benchmark results show that Llama 3.3 outperforms the Code Llama variants. Further, due to the computational cost of generating patches for over 20,000 issues, evaluating additional 70B-scale models is not feasible.

Bug Localization: We use the oracle retrieval method, i.e., supplying the model with the corresponding issue description and the exact project files modified in the developer’s ground-truth fix. This ensures that the model receives the most relevant parts of the codebase that are needed to address the issue. We intentionally adopt the oracle retrieval method since it allows us to isolate and evaluate the security of the code generated by the LLM itself, without conflating generation quality with the efficacy of retrieval methods. Additionally, the default SWE-bench retrieval baseline, BM25, fails to retrieve any relevant files for approximately 40% of test instances [32]. The focus of our study is on assessing the security of generated code when an LLM is provided with the correct context.

LLM Prompts: We adopt the prompt structure used in SWE-bench, with slight modifications. Rather than generating diffs like SWE-bench, we explicitly instruct the model to produce complete files. While diff generation is popular for its lower token overhead and ease of use with patching tools [82], in standalone LLM settings, it often yields syntactically invalid or partially applicable patches, even after post-processing [32]. Our preliminary experiments confirmed this limitation. In contrast, producing full files ensures self-contained outputs without relying on diff syntax. It also reflects realistic usage, where developers expect standalone models to produce runnable code, not diffs. For this study, we prioritize reliability over minimal token usage.

2.2.2 RQ2. Agentic LLM Frameworks. To investigate whether *agentic workflows introduce security risks in LLM-generated patches*, we analyze solutions from three of the top-performing frameworks on the SWE-bench leaderboard: *OpenHands+CodeAct v2.1 (Claude-3.5-Sonnet-20241022)* [81], *AutoCodeRover-v2.0 (Claude-3.5-Sonnet-20241022)* [88], and *HoneyComb* [30]. These framework-model combinations resolved 29.38%, 24.89%, and 22.06% of the SWE-bench test set issues, respectively. As the top publicly available entries on the leaderboard at the time of our study, they represent the strongest functional performance that agentic workflows could achieve under this benchmark [2]. Hereafter, we refer to these models as OH, ACR, and HC.

To enable a uniform comparison across models, we modified and extended the official SWE-bench evaluation harness. This harness typically applies a candidate patch to a Dockerized snapshot of the original repository state and runs corresponding test cases to assess functional correctness. For our purposes, we modified this process to capture the full impact of each patch on the project’s files. Specifically, for every file modified by a patch, we extract and save two

versions: one pre-patch snapshot and one post-patch snapshot. We also track any newly added or renamed files, along with their full file paths, to ensure consistent mapping in future steps. This process yields two datasets per framework: one consisting of pre-patch files and another consisting of post-patch files. These datasets capture the complete file-level changes made by the framework while applying the patches.

It is important to emphasize that our RQ2 evaluation is not a direct comparison between agentic frameworks and Llama. OH, ACR, and HC are evaluated using the test split, whereas the Llama and developer patches are analyzed on the combined train+test set. Our research questions are intentionally scoped to characterize behavior within each setting. For Llama, we measure how often its patches introduce vulnerabilities compared to developers on the same issues; for the agentic frameworks, we ask whether they also introduce vulnerabilities on the subset of issues they attempt to resolve, and identify the kinds of vulnerabilities they introduce through their patches. We therefore restrict all quantitative conclusions to the datasets on which each method is actually evaluated and avoid any claims of statistical superiority or inferiority between Llama and the frameworks.

2.3 Developer-written Patch Collection

To create a security baseline for comparison with LLM-generated code, we collect the original developer-written solutions for each issue from the corresponding PR using the GitHub API [22]. For each instance, we retrieve the final version of the files modified or created by the developer as part of the PR and merged into the codebase.

2.4 Vulnerability Detection

Manually analyzing the security of 20,000+ patches across multiple sources is infeasible. Therefore, in line with prior studies [15, 28, 70, 72], we utilize established OWASP-recommended [84] static analysis tools to automate the vulnerability detection process. The following tools were selected based on their widespread use [36, 72, 73], support for both rule-based and taint analysis, and their capacity to detect a wide range of real-world vulnerabilities [40]:

CodeQL [23] is a static analysis tool developed by GitHub that is often used to identify security issues and code quality problems through heuristic and taint analysis techniques.

Semgrep [66] is a static analysis tool that uses pattern-based rules and taint analysis to detect a wide range of vulnerabilities, e.g., injection risks and insecure API usage.

Bandit[60] is a heuristic-based and security-focused static analysis tool designed for Python projects. Bandit inspects abstract syntax trees to detect common security issues.

When analyzing patches, we run each static analysis tool on both the original (pre-patch) and modified (post-patch) files. By comparing the two, we classify findings as either *persisting vulnerabilities* (present in both the original and modified code) or *new vulnerabilities* (introduced only after the patch). Only the latter are counted in our results, as they directly capture the security risks added as a result of the modifications.

A common concern with static analysis tools is their accuracy, particularly their tendency to produce false positives [16, 27, 48, 52]. To overcome any shortcomings in the recall and precision of static analysis tools, we create a multi-step process as follows:

Step 1: To maximize coverage (recall), we combined the results of three static analysis tools, in line with prior studies [67], since it has been shown that different tools have complementary strengths and that combining them provides a more comprehensive assessment of code quality [11, 53, 54, 77]. The findings of the static analysis tools are aggregated using their reported vulnerability types (CWEs). Using multiple tools in this way improves coverage and recall in two ways: (A) it broadens the set of CWE types we can observe, since any single tool is limited to the

classes of vulnerabilities it supports, and (B) even when all three tools nominally cover the same CWE, any one of them can miss a concrete instance that another flags due to differences in internal analyses. All three tools perform pattern-based static analysis, but CodeQL and Semgrep additionally support taint analysis (relying on out-of-the-box taint specifications for popular APIs), and CodeQL can perform partial constraint reasoning. These complementary analyses mean that aggregating results across tools yields the highest number of true positive findings overall, thereby maximizing recall. Here, the file-level focus of the analyses not only distinguishes between new and pre-existing issues but also helps reconcile the fact that static analysis tools can report inconsistent information (e.g., severity levels, line ranges, or explanations) for the same underlying vulnerabilities [38].

Step 2: To increase confidence (precision), we adopt a conservative *majority voting strategy*: a generated or developer-written patch is treated as a security risk only if at least two of the three tools independently identify new vulnerabilities in the same file(s) it modifies or creates. Moving beyond aggregation, the majority vote dismisses findings reported by only a single tool as likely false positives. Prior work has shown that a higher concentration of static analysis warnings correlates with an increased likelihood of risky code, and that considering such concentrations can improve precision in practice [11, 77]. Additionally, merging results from multiple static analysis tools has been shown to significantly reduce the likelihood of incorrect detections [5, 17, 49]. Accordingly, independent agreement across tools both strengthens the case that a patch should be treated with caution and reduces false positives, thereby improving precision while retaining the broader coverage benefits of combining multiple tools. This conservative approach can provide a balanced and trustworthy basis for our large-scale comparative analysis across different patch sources, including developer patches, standalone LLMs, and agentic frameworks.

Step 3: To ensure validity of all instances and to guarantee that only valid security-relevant issues enter our analysis, we performed a manual inspection in which the first author inspected the patched file, and judged whether the tool-reported issue constituted a plausible security threat rather than a benign code smell. Throughout the manual inspection, whenever a finding occurred inside a function, we conservatively assumed attacker-controlled input could reach the function. The instance was retained only if no adequate input validation or sanitization was evident within the function, along the path to the relevant vulnerable lines (e.g., whitelists, length/range checks, special-character checks, or calls to dedicated sanitizers). We excluded cases where inputs were hard-coded to safe values in one-off scripts or otherwise not reusable as library functionality. We also evaluated the security of instances in light of intent. For example, instances labeled for use of SHA-1 were removed when hashing served non-cryptographic purposes (e.g., content addressing). Conversely, we retained findings when untrusted parameters could influence critical operations, such as command execution (`shell=True` with function arguments), deserialization, path construction, or SQL string creation, and no adequate sanitization was evident.

This multi-step process addresses known limitations of fully automated static analysis and ensures the quality of our findings prior to our quantitative analyses.

2.5 Metrics for Empirical Analysis

We analyze the prevalence and properties of vulnerabilities (RQ1 & RQ2) using CWE categories. We collect various code, issue, and project-level metrics to investigate the factors that are associated with the generation of vulnerabilities (RQ3).

2.5.1 RQ1 & RQ2: Vulnerability Prevalence and Properties. We analyze the prevalence and types of vulnerabilities (identified via majority voting) in code generated by Llama 3.3 Instruct (70B) and agentic frameworks (OH, ACR, HC), and compare them to vulnerabilities in developer-written patches from GitHub. Specifically, we examine:

Vulnerability prevalence: We measure the number of GitHub issues containing at least one new vulnerability.

Vulnerability types and overlaps: We measure the distribution of vulnerability types, based on CWE categories.

Further, we investigate the similarities and differences between the vulnerability patterns in LLM and developer-written code.

2.5.2 RQ3: Code, Issue, Project Characteristics Associated with LLM or agent-generated Vulnerabilities. We investigate whether the context in which the model is tasked with resolving an issue influences the security of the generated code. We analyze a range of metrics that have been shown in prior work to correlate with the riskiness of code contributions [4, 25, 26, 63], as well as introduce new metrics. Overall, these factors reflect different dimensions of task complexity and clarity. For instance, we hypothesize that issues that clearly and extensively describe the problem can be easier for LLMs to address, while tasks involving large or complex code changes may introduce more room for errors. In addition, characteristics of the project, such as size, complexity, and developer activity, may affect the likelihood of producing vulnerable code [24]. We perform this analysis for all vulnerabilities introduced both by Llama and the agentic workflows, allowing us to identify shared and distinct risk factors across different generation techniques.

We group these factors into three categories: *code-related*, *issue-related*, and *project-level* characteristics:

Code-related Characteristics: We analyze various properties of the LLM-generated code, including:

Files Added or Modified: A larger number of files can indicate broader changes and a higher overall risk [4, 25, 26, 63].

Unique File Types Modified: Modifying diverse file types may introduce different complexities to the issue resolution process [25, 26].

Sensitive File Modifications: Inspired by prior work on anomaly detecting malicious commits [25], we define a list of sensitive file types (e.g., .conf, .yml, .json) whose modification can increase the risk of vulnerabilities.

LOC Generated by LLM: This measures how many lines were added or rewritten by the LLM. Generating more lines of code may suggest that the model is undertaking more complex tasks, handling broader context, or replacing larger code chunks, which can raise the likelihood of mistakes or security issues.

Cyclomatic Complexity of LLM-Generated Code: This metric reflects the complexity of the functions introduced or rewritten by the LLM. Prior studies also analyzed the cyclomatic complexity of LLM-generated code [20, 76]. Higher complexity in generated code suggests that the model is constructing more control flows, which can be more error-prone and harder to verify for correctness and security.

Issue-related Characteristics: We extract several features from the GitHub issue associated with each instance:

Issue Type and Bug Type Classification: We manually label whether the issue is a bug, a feature request, or a question, following established comprehensive taxonomies [34, 74]. Further, for issues categorized as bugs, we classify the bug type using a comprehensive, empirically grounded, and widely-adopted taxonomy[9]. The bug categories are: *Configuration Issues*, *Network issues*, *Database-related issues*, *GUI-related issues*, *Performance issues*, *Permission/deprecation issues*, *Security issues*, *Program anomaly issues*, *Test code-related issues*, and *Miscellaneous/other issues*.

Information Completeness: Based on prior work emphasizing the importance of issue clarity [74], we manually annotate whether each issue includes: (1) *Expected Behavior*: how the software is supposed to behave, (2) *Observed Behavior*: how it actually behaves, and (3) *Steps to Reproduce*: the process needed to trigger the issue. The presence of this information can aid in understanding the bug's context and identifying the root cause of the problem.

Words: We measure the total word count of the issue description and all the comments made on the issue prior to the date of first commit in the corresponding PR.

Comments Prior to First PR: We measure the number of comments on the issue thread made before the first pull request was submitted to address the problem. A higher comment count may indicate increased back-and-forth discussion, potentially reflecting communication challenges or ambiguities that required clarification through developer interactions.

Presence of Code Snippets: Using regular expressions, we identify code snippets in issues and use them as indicator of technical detail and potential guidance for bug resolution.

Manual annotation of *issue type*, *bug type*, and *information completeness* was performed independently by two annotators, each with at least three years of programming experience. For each data point, they examine both textual and code-related data, including the GitHub issue title, description, discussion thread, etc. The annotators followed a shared set of annotation instructions (included in our replication package). They followed an iterative analysis comprising multiple sessions. Inter-rater agreement was initially measured using Cohen’s kappa for each category (all values > 0.61). The annotation process proceeded through several rounds of labeling and discussion. This resulted in a final strong agreement level (Cohen’s kappa values > 0.9). **Project-related Characteristics**: Finally, we assess project characteristics associated with the vulnerable LLM patches:

Contributors: Using the GitHub API, we measure the number of developers contributing to a project. Specifically, we count all users listed as GitHub contributors, i.e., users who have made at least one commit to the repository’s default branch. This metric reflects the breadth of developer involvement in a project, which can influence code quality and review process. Projects with more contributors often have greater oversight, which can lower the risk of security vulnerabilities.

Files: The number of files in a given project can serve as a proxy for its potential complexity and size. Prior work on open-source software effort estimation has used file count as an indicator of project scale and structural complexity [61].

Further, size of a project has been shown to have direct associations with the number of potential vulnerabilities [24].

Cyclomatic Complexity and Maintainability Index of Project: We use Radon [1], a popular code analysis tool [29, 42], to compute each project’s average maintainability index and cyclomatic complexity. Cyclomatic complexity is computed as the average of complexity scores across all functions in the codebase. Higher values indicate more intricate control logic throughout the project, potentially making it harder for LLMs to reason about and modify the code securely.

Maintainability index is derived from several factors, including cyclomatic complexity, lines of code, and Halstead volume, and is designed to estimate how easy it is to maintain the codebase. A higher maintainability index suggests code that is easier to understand and modify, which may reduce the risk of generating vulnerabilities.

We measure the code, issue, and project-level metrics on the entire SWE-bench dataset. For Llama-generated vulnerabilities, we report the metrics calculated on the data associated with all 21,294 issues in the dataset. For agent-generated vulnerabilities, these metrics are calculated only on the corresponding data associated with the *test* set of 2,294 issues.

3 Results

3.1 RQ1: Security of patches generated by Developers vs. Llama 3.3

Vulnerability Prevalence: Table 1 presents the number of files with new vulnerabilities introduced by *Developers* and *Llama 3.3* patches. The raw counts vary significantly across tools and code sources. For the full SWE-bench dataset

(train+test), Llama 3.3 generated 1440 vulnerabilities, of which 185 (12.8%) passed the majority vote filter. Upon manual inspection, 135/185 vulnerabilities remained. In contrast, developer patches for the full dataset triggered 1398 alerts, but only 17 vulnerabilities (1%) were retained after majority vote filtering. We manually verified 12/17 of these vulnerabilities to be valid security concerns.

Patch Source	Split	Majority Vote	Manual Check
Llama 3.3 Developers	Train + Test	185	135
	Train + Test	17	12
Llama 3.3 Developers	Test	14	9
	Test	3	2
OH	Test	195	44
	Test	3	3
ACR	Test	2	2
HC	Test		

Table 1. Number of vulnerabilities detected across evaluation splits.

When comparing individual tools, Bandit reported the highest number of vulnerabilities overall: 1383 in Llama-generated code and 1055 in developer code. Semgrep detected 323 vulnerabilities in LLM outputs but just 9 in developer code. CodeQL, the most conservative tool, identified 62 vulnerabilities in LLM-generated code and 41 in developer code. Overall, our results suggest that, consistent with prior work [91], Bandit produces more false positives which had to be removed in the next two steps.

It is worth noting that all reported numbers are the new vulnerabilities that were injected only after the LLM/Developers modified the files or added the code to the repository. These vulnerabilities did not exist in the code before the patches were applied. For instance, in an issue from the *Qiskit* project, Llama introduced a CWE-95 vulnerability by using `eval()` to interpret user-provided input strings as Python code. These strings are strictly sanitized in developers' code. The LLM-generated code, however, converted the string into a Python lambda function without proper sanitization or sandboxing. This enabled the possibility of a code injection attack, since malicious input could be executed directly.

Vulnerability Types and Overlaps: Table 2 contains the most common vulnerabilities in Llama-generated and developer-written code¹. The most common vulnerabilities in Llama-generated patches include command injection (CWE-78, 79 instances), eval injection (CWE-95, 39), insecure deserialization (CWE-502, 17), and path traversal (CWE-22, 14). While less frequent, other vulnerability types like incorrect permission assignment or improper authorization also appeared in the LLM patches. The full set of vulnerabilities are included in our replication package.

Llama 3.3-Generated Code	
CWE-78 (OS Command Injection)	79
CWE-95 (Eval Injection)	39
CWE-502 (Insecure Deserialization)	17
CWE-22 (Path Traversal)	14

Developer-Written Code	
CWE-78 (OS Command Injection)	4
CWE-377 (Insecure Temporary File)	3
CWE-732 (Permission Misconfiguration)	3
CWE-22 (Path Traversal)	3

Table 2. Most recurring CWE types found in Llama-generated and developer-written patches across the train & test sets.

¹Each tool can assign multiple CWEs to the same finding and each vulnerability can be assigned different CWEs by different tools. Therefore the total number of CWE types is higher than the total number of vulnerabilities.

Developer-written patches showed far fewer vulnerabilities overall, but some of the top categories i.e., command injection (CWE-78, 4) and path traversal (CWE-22, 3), are similar to the LLM results. These two common CWEs (mostly related to improper input handling and insufficient sanitization when interacting with system commands or file paths) and their presence in both sets may stem from the fact that LLMs are trained on human code, inheriting common patterns, including insecure ones.

Despite the similarities in vulnerability patterns, certain categories were far more prominent in the Llama-generated code and extremely rare within developer-written patches. Notably, eval injection (CWE-95) was identified in 39 LLM patches and only one developer patch. Similarly insecure deserialization (CWE-502) was repeated in 17 LLM patches while being detected in only two developer patches. These issues likely stem from the model's tendency to translate task instructions into overly literal code, for example, using `eval()` to dynamically execute strings, or defaulting to insecure algorithms like MD5. Unlike developers, who often follow project standards or rely on secure external frameworks, Llama preferred simpler but less secure implementations. This behavior reflects the model's narrow focus on completing the task at hand with simpler code, often at the expense of secure best practices. For instance, in an issue requesting improved password hashing, Llama generated a new custom hasher that used SHA-256 instead of upgrading the existing Argon2-based implementation. While SHA-256 is straightforward to apply, it lacks protective features against brute-force attacks (CWE-327). Additionally, the LLM produced the following implementation of the `check_password()` function, which includes a vulnerable use of `hashlib.md5()`:

```
def check_password(raw_password, encoded_password, setter=None):
    ...
    data = raw_password.encode('utf-8')
    return hashlib.md5(data).hexdigest() == encoded_2['hash']
```

If such patches were merged without security checks e.g., CI-integrated static analysis runs or manual reviews, the consequences for a real system could be severe. In the password-hashing example, the model introduces a secondary MD5-based check that directly compares attacker-controllable input against stored hashes. In a production environment, this would expose users to significantly higher risk of credential compromise; brute-force and dictionary attacks become dramatically more feasible, and any leakage of the hash database (through a separate vulnerability) would have far more damaging consequences than under the original design. More broadly, many of the CWE categories we observe in LLM patches e.g., eval injection, insecure deserialization, and command injection, are associated with high-impact outcomes such as arbitrary code execution, privilege escalation, or unauthorized access to sensitive data. Because these vulnerabilities are introduced in the course of fixing issues, they are unlikely to be caught by functional tests alone, and can remain latent until exploited. As a result, if LLM-generated fixes are integrated without additional security checks, they may fix the immediate bug while degrading the system's security.

RQ1 Summary: Across the SWE-bench train+test set, the standalone Llama 3.3 model introduces 11x more new vulnerabilities than developers: after majority voting and manual validation, we confirm 135 vulnerabilities in LLM-generated patches versus 12 in developer patches (Table 1). While both developers and the LLM introduce issues such as command injection (CWE-78) and path traversal (CWE-22), categories like eval injection (CWE-95) and insecure deserialization (CWE-502) are far more prevalent in LLM-generated code. Moreover, none of the confirmed LLM vulnerabilities overlap with developer vulnerabilities in the same files or issues, indicating that LLM-generated fixes can introduce distinct security weaknesses rather than merely replicating existing developer errors.

3.2 RQ2: Security of patches generated by the agentic APR workflows

Vulnerability Prevalence: Table 1 also reports the number of vulnerabilities for agentic frameworks i.e., OH, ACR, and HC, when evaluated on the SWE-bench test set. Similar to RQ1, we manually inspect all instances to validate our data.

The static analyses and the majority voting process resulted in 195, 3, and 2 instances marked as potential vulnerabilities by OH, ACR, and HC, respectively. As with Llama and developer patches, Bandit consistently reported the highest number of vulnerabilities, followed by Semgrep and then CodeQL. This pattern holds across all three frameworks. However, despite the high number of raw detections, the majority vote strategy proved highly selective: for example, OH generated over 1,383 alerts across all tools cumulatively, yet only 195 vulnerabilities (approximately 15%) were retained in the final filtered set. This reinforces the importance of using a stricter intersection criterion when analyzing code security. Next, led by the manual inspection process, 44 out of 195 OH vulnerabilities, and all 5 vulnerabilities from ACR and HC were confirmed as valid security threats.

While all frameworks introduced new vulnerabilities, OH produced a disproportionately larger number. To better understand this disparity, we closely examined the results from OH. Among other factors, OH is given full autonomy, not only to inspect and modify project files, but also to execute commands, run tests, and generate new scripts as part of its process. This broad operational scope means that it often touches more files than necessary to resolve an issue. Our analysis indicates that out of the 195 vulnerabilities initially attributed to OH, 12 were located in test files and 35 were found in files intended to reproduce the issue (e.g., `reproduce.py`, `reproduce_error.py`). Additionally, 75 of the marked instances were parts of various packages downloaded into the repository and included as part of the patch by OH. Since such files are not typically intended for integration into the final production system or regular execution, we exclude them from the remainder of our analysis. Furthermore, we observe that, all the remaining 44 vulnerabilities in OH originated from just 11 issue instances. In fact, three issues alone, *django ticket #14056*, *pytest issue #5787*, and *django ticket#14681*, accounted for 35 of the 44 final vulnerabilities. In the most extreme case, *django ticket #14056*, OH generated a diff file that exceeded 500,000 lines in length. This pattern reflects a potential problem in the framework’s control mechanisms. Rather than solving the issue with minimal, targeted changes, OH can enter error-prone iterative loops that result in massive file rewrites or reimplementations of the existing core software components. These behaviors likely stem from the limitations of the agentic process in comprehending the full context of the issue and the project. As a result, OH may generate excessive or redundant code that is potentially vulnerable.

In one instance, OH attempted to resolve an issue in the *django* project, related to invalid SQL generation from constraint expressions involving ordering (e.g., `Lower("name").desc()`). The original bug caused form validation to fail with SQL syntax errors because DESC was incorrectly placed inside a WHERE clause. The developer patch resolved this cleanly by removing any ordering operations (like `desc()`) before constructing validation expressions:

```
if isinstance(expr, OrderBy):
    expr = expr.expression
expressions.append(Exact(expr, expr.replace_expressions(...)))
```

In contrast, the OpenHands’ patch bypassed Django’s ORM entirely and rewrote the validation logic using raw SQL:

```
where_clause = " AND ".join(conditions)
sql = f"SELECT 1 FROM {table_name} WHERE {where_clause} LIMIT 1"
cursor.execute(sql, params)
```

While this approach can address the original issue, it introduced a new vulnerability, CWE-89 (SQL injection), flagged by both Bandit and Semgrep. Specifically, although query parameters were bound securely, the `field_name` values, used to construct column identifiers, were directly inserted into the query string:

```
conditions.append(f"qn({field_name}) = %s")
```

If these expressions contain malicious input (e.g., `F("name"); DROP TABLE users; -")`), they could directly manipulate query. Django's ORM normally safeguards against such risks, but manual SQL construction ignores these protections. This example illustrates how a model, may introduce security risks by deviating from design principles.

Vulnerability Types and Overlaps: Among the agentic frameworks, OH, responsible for 44 vulnerabilities, exhibited a similar vulnerability pattern to that of Llama 3.3: command injection (CWE-78, n=10), eval injection (CWE-95, n=8), and insecure deserialization (CWE-502, n=6) were all among the common CWEs in OH patches. This alignment suggests that even when LLMs are embedded within frameworks, some core security risks and patterns persist.

Interestingly, even the relatively smaller number of vulnerabilities generated by ACR and HC included categories like command injection and eval injection. Given that the dataset used in RQ2 is smaller than in RQ1, the presence of these CWE types across all three agentic frameworks highlights the persistent security risks posed by autonomous repair agents.

Moreover, we find that vulnerabilities introduced by the agentic frameworks are entirely different from those written by the developers. These results echo the same observations in Section 3.1. While developers, Llama, and agentic frameworks all can add vulnerabilities to the codebase, the vulnerabilities produced by the LLMs or agentic frameworks are not simply repeating the developers' mistakes.

RQ2 Summary: While the agentic frameworks were evaluated on a smaller subset of data (test-set), **all three still introduced a number of vulnerabilities**. OH, in particular, produced an order of magnitude more vulnerabilities than ACR or HC. These results indicate that, despite improvements in autonomy and reasoning, current agentic workflows remain vulnerable to security issues, especially when given full control over the codebase.

3.3 RQ3: What code, issue, or project characteristics are associated with the generation of vulnerable code by LLMs when resolving issues?

3.3.1 Code-Level Factors. We analyze the code-level characteristics of the files associated with vulnerability-inducing LLM patches and compare them to those from all LLM-modified or generated files. Table 3 summarizes these comparisons for Llama-generated patches (RQ1). We note that the dataset for RQ1 is significantly larger than RQ2. As a result, due to the limited number of vulnerable examples in ACR and HC, we report statistical comparisons only for OH. However, we include the raw values for all three frameworks in Table 4.

Metric (Llama)	Vuln Mean	All Mean	p-value	Cliff's δ
Files Added or Modified (All)	1.72	1.25	<0.001	0.200
Files Added or Modified (Python)	1.60	1.16	<0.001	0.199
Unique File Types Modified	1.10	0.99	0.007	0.081
Sensitive Files Modified	0.00	0.007	0.369	-0.006
LOC Generated by LLM (Python)	1740.50	2218.12	<0.001	0.197
Avg. Cyclomatic Complexity of LLM-Generated Code	2.88	3.02	0.8796	-0.009

Table 3. Summary statistics comparing vulnerable vs. all Llama/agency-generated code for code-level metrics

Metric	OH		ACR		HC	
	V	A	V	A	V	A
Files Added or Modified (All)	344.27	11.38	1.33	1.07	1.50	1.46
Files Added or Modified (Python)	68.83	4.46	1.33	1.07	1.50	1.40
Unique File Types Modified	3.73	1.41	1.00	0.98	1.00	0.90
Sensitive Files Modified	0.0909	0.0136	0.00	0.00	0.00	0.004
LOC Generated by LLM (Python)	1218.6	1591.0	387.33	1294.6	1591	1764
Avg. Cyclomatic Complexity of LLM-Generated Code	4.015	4.513	7.352	4.303	6.470	4.363

Table 4. Mean values for vulnerable (V) vs. all (A) agentic-generated patches across code-level metrics.

Files Added or Modified: As shown in Table 3, Llama-generated vulnerable patches (*Vuln Mean*) involve modifications to a larger number of files compared to the entire LLM-generated set (*All Mean*), both in overall (1.72 vs. 1.25) and within Python files (1.60 vs. 1.16), with statistically significant differences ($p < 0.001$, Cliff's $\delta = 0.2 \& 0.19$).

Among the agentic frameworks, OH showed an even more dramatic pattern: vulnerable instances modified 344.27 files on average (vs. 11.38), including Python files (68.83 vs. 4.46), with large effect size of $\delta=0.77$ and a p-value<0.001. This reflects the framework's tendency to perform excessive rewrites. Additionally, the differences in ACR (1.33 vs. 1.07) and HC (1.50 vs. 1.46) showed higher means for vulnerable instances.

Figure 2 compares the distribution of changed/added Python files per instance between vulnerable and all patches, across different generative sources. We observe that all systems show a general increase in the number of modified files in vulnerable instances relative to the rest. OH stands out with a substantial shift, indicating significantly broader edits. Llama-generated patches, similar to ACR and HC, also display strong differences with lower overall file counts. These trends highlight that vulnerability-prone patches tend to involve more scattered edits across multiple files.

Unique File Types Modified: Vulnerable code modifications by Llama affected a slightly broader range of file types (mean = 1.10 vs. 0.99). This difference is statistically significant ($p = 0.007$), although the effect size is negligible (Cliff's $\delta = 0.096$). This suggests that file type diversity alone is a weak indicator of vulnerability risk but may still inform broader risk assessments when combined with other features.

Across the frameworks, we observe similar patterns: ACR (1.00 vs. 0.98) and HC (1.00 vs. 0.90) both show higher mean values for the vulnerable instances. Further, OH exhibited a more notable increase (3.73 vs. 1.41), with statistical significance and meaningful effect size ($\delta = 0.345$, $p = 0.0011$). This reinforces earlier findings about OH's broader file editing behavior, which may contribute to higher vulnerability risk in its generated patches.

Sensitive file modifications: using our predefined a set of sensitive file types (e.g., .conf, .yml, .json), we compare their modification rates in vulnerable and all instances. In Llama-generated code vulnerable instances did not show significant changes in modifying such files ($p = 0.369$), indicating that Llama 3.3 does not disproportionately touch sensitive configuration files when generating insecure code.

This pattern holds for ACR (0.0 vs. 0.0) and HC (0.0 vs. 0.004). OH, on the other hand, exhibited a statistically significant increase in sensitive file edits (0.09 vs. 0.01, $p < 0.008$), suggesting that its more expansive editing behavior may lead it to touch more sensitive files. However, the effect size remained negligible ($\delta = 0.081$), and results across frameworks were mixed. This indicates that while sensitive file edits are a contributing factor in certain frameworks, they are not a consistent predictor of vulnerability on their own.

LOC Generated by LLM: This factor represents the scale of code generation by the LLM. In the Llama-generated patches, vulnerable instances involve less extensive edits (1740 vs. 2218 lines on average), a statistically significant difference ($p < 0.001$) with a positive effect size ($\delta = 0.197$). This suggests that larger code generations are not more prone to

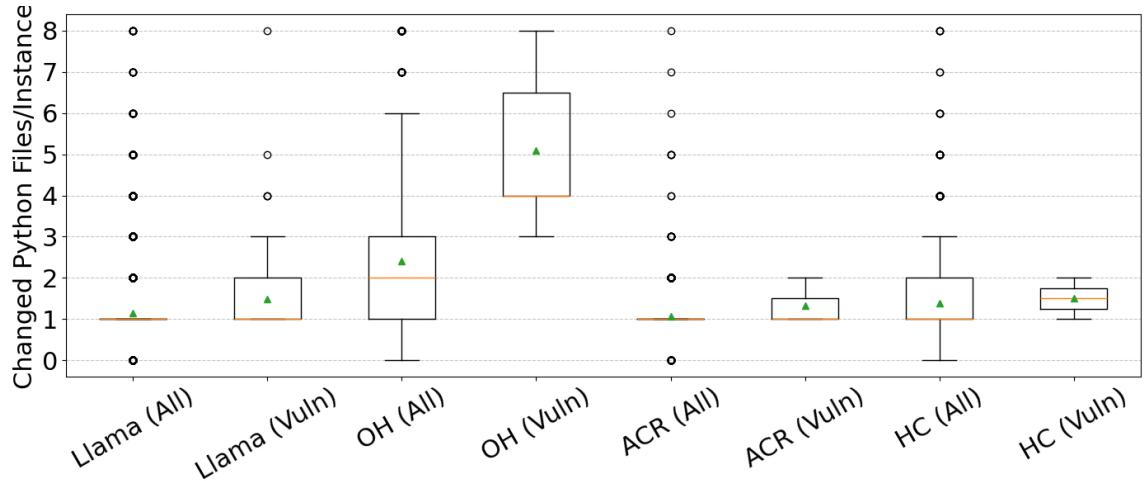


Fig. 2. Number of changed files per instance (capped at 10). Vulnerable cases show greater variability and significantly higher averages than the general set.

vulnerabilities. As demonstrated in prior work, even highly concise edits can introduce substantial security risks. For instance, severe vulnerabilities such as Heartbleed and ShellShock stemmed from only a handful of modified lines, showing how small changes to core logic can have extreme consequences [46]. This finding also aligns with the empirical studies of developer activity, revealing that vulnerability-inducing commits are shrinking in size over time, with smaller commits associated with more severe vulnerabilities [7, 31]. Viewed in this context, our finding suggests that the brevity of LLM-generated patches may reflect the models' difficulty in fully reasoning about program context. When models alter a few lines of critical logic based on incomplete assumptions, these compact changes can still meaningfully compromise the security of software.

Among the agentic frameworks, ACR shows the strongest contrast, with vulnerable outputs averaging fewer lines (387 vs. 1294). OH also shows a reduction in generated LOC for vulnerable instances (1218 vs. 1591), with a small effect size ($\delta = -0.10$, $p = 0.51$). In contrast, HC exhibits the opposite trend, with vulnerable code averaging more lines (1764 vs. 1591). These contrasting patterns suggest that while the scale of generation can be associated with vulnerability likelihood, the relationship can be highly dependent on the specific framework's behavior and other contextual factors. *Cyclomatic Complexity of LLM-Generated Code:* We assess whether the vulnerable outputs contain more complex control flows by measuring the average cyclomatic complexity of functions in files generated or modified by the LLM. In the Llama patches, vulnerable instances show slightly lower complexity (2.88 vs. 3.02), but this difference is not statistically significant ($p = 0.8796$, $\delta = -0.009$), suggesting that structurally simple code may still contain security flaws.

Among agentic frameworks, OH shows a similar trend, with vulnerable files having lower complexity (4.015 vs. 4.513), though the difference is not statistically significant ($p = 0.688$, $\delta = 0.035$). Further, HC shows a slight increase in the complexity of vulnerable files (6.47 vs. 4.363). Notably, ACR exhibits a significant increase in complexity of vulnerable files (7.352 vs. 4.303). Our observations highlight that vulnerability is not limited to structurally complex code as simple logic can still be insecure. Yet, abnormally high complexity, as with the patches of ACR, can serve as an indicator for security risk, offering a practical signal for prioritizing code review.

Code-Level Factors

Across all systems, vulnerable patches typically modify more files and file types, but involve fewer lines of generated code. Higher-risk patches in ACR exhibit abnormally high code complexity.

3.3.2 Issue-Level Factors. We analyze the characteristics of GitHub issues to determine whether specific issue features correlate with the introduction of vulnerabilities.

Issue Type and Bug Type Classification: Among all the vulnerable instances generated by Llama and the agentic frameworks, the vast majority of issues were labeled as *Bug*. For Llama-generated vulnerable patches, 105 out of 135 (77.7%) were classified as *Bug*, 29 (21.4%) as *Feature*, and 1 (0.7%) as *Question*. In contrast, OH had 9 *Bugs* and 35 *Feature*; ACR had 2 *Bugs* and 1 *Feature*; and HC had 2 *Bugs*. Here, OH stands out as it often generated vulnerable code when attempting to resolve issues that seek to implement new feature within projects.

Table 5 summarizes the distribution of bug types across all vulnerable instances. The most frequent categories are *Program Anomaly* and *Configuration*. Other bug types, such as *Performance*, *GUI-related*, appear less frequently. Overall, vulnerabilities of Llama 3.3 and the agentic frameworks mostly involve addressing core logic or environment-related issues, where deeper contextual understanding is critical.

Bug Type	Llama	OH	ACR	HC
Program Anomaly Issue	63	7	1	2
Configuration Issue	18	2	–	–
Miscellaneous/Other Issue	8	–	–	–
Performance Issue	5	–	–	–
GUI-Related Issue	3	–	–	–
Test Code-Related Issue	3	–	–	–
Network Issue	3	–	–	–
Permission/Deprecation Issue	2	–	–	–
Security Issue	–	–	1	–

Table 5. Bug type distribution across vulnerable instances.

Information Completeness: Among vulnerable LLM cases, nearly all (101/105) bug-related issues included *observed behavior*, but just about 64.7% (68/105) included *steps to reproduce*, and only 39% (41/105) provided *expected behavior*. These rates indicate that while problems are usually described, some important information, i.e., what the code is supposed to do and how to reproduce the issue, is often missing. This lack of information may increase ambiguity, forcing the model to infer expected behavior without explicit instructions, thereby raising the risk of inadequate implementations.

Examining the 11 unique GitHub issues for agentic workflows (6 for OH, 3 for ACR, 2 for HC), we observe a high rate of information completeness for OH instances: all vulnerable issue reports included observed and expected behavior and provided steps to reproduce. In ACR, both vulnerable instances (2/2) had all three types of information. For HC, 2 out of 3 issues include observed behavior, 1 provides expected behavior, and 2 offer reproduction steps. While sample sizes are limited, we observe that framework-generated vulnerabilities can occur even when issue descriptions are reasonably specified.

Words: We observed minor differences in the length of issue descriptions and associated comments in vulnerable vs. all Llama-generated instances. On average, problem descriptions contained 111.71 words for all issues and 105.78 words for vulnerable instances. Neither the word count of problem descriptions alone ($p = 0.5346$, Cliff's $\delta = -0.031$) nor the combined word count including comments ($p = 0.4726$, $\delta = 0.036$) reached statistical significance. These results

suggest that the overall length of issue content alone is not a strong differentiator in predicting whether the model will introduce vulnerabilities, and possibly other factors, such as clarity or specificity, can play a more important role than the amount of available textual data.

Across the frameworks, we observed mixed patterns. OH and ACR had higher average word counts when combining issue and comment text (547.16 and 435.67 words, respectively), while HC's was much lower (96.0). This illustrates the unique way each framework is impacted by the available information.

Comments Prior to First PR: Llama-generated vulnerable instances had slightly more comments on average (4.1) compared to all issues (3.6), but the difference was not significant ($p = 0.68$, $\delta = 0.025$), indicating that comment count does not clearly distinguish vulnerable from non-vulnerable LLM-generated patches.

Across all three frameworks, none of the vulnerable issues contained comments prior to the first pull request. This absence suggests that, unlike in the Llama-generated set, prior discussion was not associated with an increase in the vulnerabilities observed within the frameworks.

Presence of Code Snippets: We observe that only 47.4% of issues associated with vulnerable LLM outputs include code snippets in the issue body, compared to 58.6% of all issues. This difference is statistically significant ($\chi^2 = 6.44$, $p = 0.011$), suggesting that the presence or absence of code snippets is not evenly distributed between the two groups. The chi-squared test evaluates whether there is a meaningful association between two categorical variables, in this case, code snippet presence and vulnerability occurrence. To assess the strength of this association, we calculate the phi coefficient ($\phi = 0.017$), a standardized effect size for 2×2 contingency tables. While statistically significant, the effect size is negligible, indicating a weak association. Nonetheless, the direction of the effect suggests that Llama 3.3 is more prone to generating vulnerabilities when issue descriptions lack code snippets, possibly due to the absence of concrete examples, stack traces, or code context, which increases ambiguity. Overall, these trends indicate that the potential ambiguities in issues that lack code snippets can increase the risk of vulnerabilities.

Issue-Level Factors

Insecure patches from [Llama 3.3](#) are associated with issues where code snippets are absent, and when expected behavior or reproduction steps are missing. Most of these issues are labeled as *Bug* and involve core logic or configuration problems. OH is particularly sensitive to issues related to feature development and those without code snippets.

3.3.3 Project-Level Factors. We also examined whether broader project characteristics, such as codebase size, complexity, and contributor count, are associated with the number of vulnerabilities introduced by LLM or agent-generated code. We used Spearman's rank correlation coefficient (ρ) to assess monotonic relationships between each project-level factor and the number of detected vulnerabilities across instances.

Files in Repository: We examined whether the size of the codebase, as measured by the total number of Python files in the repository, correlates with vulnerability outcomes. For Llama-generated code, the correlation was weak and not statistically significant ($\rho = 0.136$, $p = 0.362$). Similarly, we found no meaningful correlations in any of the agentic frameworks: OH ($\rho = 0.70$, $p = 0.642$), ACR ($\rho = 0.205$, $p = 0.168$), and HC ($\rho = 0.171$, $p = 0.251$). These findings suggest that codebase size is not a reliable indicator of LLM-generated vulnerabilities.

Maintainability and Complexity: Similarly, we found no statistically significant correlation between vulnerability counts and average maintainability index or the cyclomatic complexity per function in the projects. Within the LLM-generated code, maintainability ($\rho = -0.006$, $p = 0.968$) and complexity ($\rho = -0.062$, $p = 0.679$) were not correlated with vulnerability

introduction. The same holds for the frameworks. For example, in OH, maintainability ($\rho = -0.021$, $p = 0.888$) and complexity ($\rho = 0.049$, $p = 0.744$) were not predictive of vulnerability counts. ACR and HC showed similar patterns. These findings indicate that the internal complexity or maintainability of a project is not a strong predictor of insecure code generation by LLM-based systems.

Contributor Count: We also examined whether the number of contributors in a project correlates with the introduction of vulnerabilities. For Llama-generated code, the association was weak and statistically insignificant ($\rho = 0.184$, $p = 0.217$). The same held true across all agentic frameworks: OH ($\rho = 0.210$, $p = 0.156$), ACR ($\rho = 0.158$, $p = 0.288$), and HC ($\rho = 0.109$, $p = 0.467$). These findings suggest that community size, used here as a rough proxy for project activity, does not significantly influence LLM vulnerability outcomes, although the consistently positive (albeit weak) correlations hint at a very slight directional trend.

Project-Level Factors

Project size, complexity, and contributor count showed weak, non-significant correlations with vulnerability rates. As a result, these factors alone were not observed to reliably predict insecure LLM-generated code.

RQ3 Summary: Across all systems, vulnerable patches are most strongly associated with code and issue-level characteristics rather than project-level properties. At the code level, insecure patches consistently modify more files (and, for OH, more file types), while often involving relatively small, localized edits in each file. In ACR, vulnerable patches also exhibit substantially higher cyclomatic complexity than non-vulnerable ones. At the issue level, vulnerabilities predominantly present in bug reports and configuration-related issues, especially when expected behavior and reproduction steps are under-specified and when issue descriptions lack concrete code snippets. In contrast, project-level factors such as repository size, cyclomatic complexity, and contributor count show non-significant correlations with vulnerability counts.

4 Threats to Validity

Our security analysis uses static analysis tools, which can produce false positives and vary in coverage. To mitigate this threat, we used three tools (Semgrep, Bandit, and CodeQL), maximizing coverage, and applied a majority voting technique to reduce false positives and increase confidence in the identified vulnerabilities. We further, performed a manual analysis on all the instances to remove all remaining false positives.

Moreover, our evaluation focused on one high-performing LLM and the top three publicly available agentic frameworks at the time. While there are many other models and frameworks in the space, our selection captures a broad and representative sample of state-of-the-art systems. Additionally, while our LLM-based approach generated responses for both the train and test sets (a process that took well over a week), we acknowledge the limitation in terms of data scale and have adjusted our conclusions accordingly. For instance, AutoCodeRover reports a cost of \$0.70 per instance to achieve their test set results, meaning it would cost over \$13,000 to generate patches for the full 19,000 instances of training set, which is beyond our available resources. To keep our conclusions accurate, we therefore refrain from making strong statistical claims about the agentic systems, report their raw counts transparently, focus on qualitative analysis of the vulnerable instances they produce, and avoid direct quantitative comparisons with the standalone LLM.

We also acknowledge that our RQ3 analysis of issue types is limited by scale as manually annotating all 20,000 SWE-bench issues is infeasible, and many issues lack reliable labels. As a result, we do not claim that bug-fixing issues are inherently more prone to vulnerable patches; we only report that, within our manually annotated vulnerable instances, most affected issues are labeled as bugs.

Lastly, manual issue annotations may be prone to human error. To improve reliability, all annotations were independently performed by two reviewers, whom ensured a strong inter-rater agreement (Cohen’s Kappa > 0.9).

5 Discussion

5.1 Llama vs. Developers (RQ1)

The RQ1 results show that Llama introduces substantially more new vulnerabilities than developers on the same SWE-bench issues, even after we apply a conservative detection pipeline with majority voting and manual validation. Qualitatively, the vulnerabilities in LLM-generated code differ from developer-written patches. Both introduce issues such as command injection and path traversal, but LLM patches disproportionately exhibit categories like eval injection and insecure deserialization, and in some cases downgrade existing cryptographic mechanisms or include multiple vulnerabilities within a single patch. These patterns indicate that, in an APR setting, LLMs do more than repeat common developer mistakes; rather, they demonstrate distinct vulnerability patterns when asked to repair real-world bugs.

These findings indicate that Llama’s patches should be treated as an *additional source of security risk*. From an adoption standpoint, this indicates a clear trade-off: the LLM can accelerate development, but it also introduces a non-trivial probability of latent vulnerabilities, even in mature projects. This implies that (i) generated patches should not bypass existing code review and security processes, (ii) organizations deploying LLM-based APR tools should consider integrating security scanning and some form of manual inspection into workflows that accept LLM patches, and (iii) using these LLM fixes as a replacement for developer-written patches, without extra safeguards, is likely to increase both the frequency and diversity of vulnerabilities in codebases.

5.2 Agentic APR Frameworks (RQ2)

The RQ2 results show that agentic APR workflows built around LLMs do introduce security vulnerabilities, and that these vulnerabilities are not evenly distributed across frameworks or issues. In particular, OH’s patches display significantly different behavior than those of ACR and HC. A small subset of issues result in very large, multi-file diffs and multiple confirmed vulnerabilities produced by OH, whereas the other frameworks tend to modify fewer files and produce smaller patches.

This pattern points to a problem of highly autonomous APR frameworks. For an agent with broad permissions to edit the repository, run arbitrary commands, and generate new scripts for hundreds of iterations, any initial incorrect assumption regarding the codebase or the task can lead to long ineffective attempts to fix a problem. Over iterations, the patch may grow in size and scope, drifting away from the project’s original design and security expectations by modifying components outside the fault-localization region that the agent does not fully comprehend.

The comparatively small number of vulnerabilities observed in ACR and HC should not be interpreted as evidence that these systems are inherently secure, but it can suggest that tighter control over search space and modification scope can reduce the likelihood of vulnerabilities. ACR, particularly, operates with more constrained edit patterns and less extreme diff sizes than OH, which plausibly limits opportunities for large incorrect rewrites. At the same time, the vulnerabilities these frameworks introduce often fall into the same CWE categories e.g., command injection, eval injection, indicating that the core security issues of the underlying LLMs persist even when used in more restricted frameworks.

The results of RQ2 suggest that (i) giving agents unconstrained autonomy over large repositories can substantially increase the risk of vulnerabilities, especially via large, cross-file edits; (ii) frameworks that keep their edits more

localized appear to have smaller vulnerability counts; and (iii) in all cases, the vulnerabilities introduced by agentic frameworks are distinct from those found in developer patches (similar to RQ1), emphasizing that these tools add a new layer of risk rather than merely repeating developers' mistakes.

For practitioners considering agentic APR tools, these findings point to potentials for important design and deployment guidelines. Frameworks should incorporate explicit limits on edit scope e.g., caps on diff size, limits on the number of files that can be modified in a single attempt, etc. Additionally, given that a small number of issues can account for a disproportionate share of vulnerabilities, it is beneficial to detect high-risk repair tasks such as those that repeatedly fail tests or result in unreasonably large diff files, and route them to human review rather than allowing the agent to continue autonomously.

5.3 Code, Issue, and Project Characteristics of Vulnerable Instances (RQ3)

Our analysis of over 20,000 LLM-generated patches suggests that vulnerability-prone outputs are not random. Instead, they appear around a relatively consistent set of characteristics, especially at the *code* and *issue* level.

At the *code level*, we see a strong association between insecure patches and broader, more scattered edits. Vulnerable patches tend to touch more files and spread their changes across multiple locations, rather than making a self-contained change inside a single function or module. These are the situations where an LLM or APR agent must reason about cross-file dataflows and the interaction between different components, rather than a small local context. Our results indicate that such multi-file, dispersed edits are where current systems are most likely to introduce security weaknesses. In practice, if a repair task can be localized to a single file or a specific component, allowing an LLM to propose a patch in that narrow context appears substantially safer than giving it free rein over a large part of the repository. When tasks require multi-file edits, it may be preferable to decompose the work into smaller changes, each operating over a part of the code, rather than requesting one large cross-file patch. Additionally, we observe that in ACR, vulnerable patches are associated with significantly higher cyclomatic complexity. This is not a persistent signal across all systems, indicating that risk factors can depend on the internal workflow of the APR system. This suggests that for some APR frameworks, complexity-sensitive safeguards, e.g., requiring human review or additional checks when edits occur in highly complex functions, may be justified, even if the same signal is less informative elsewhere.

At the *issue level*, vulnerabilities are commonly found in bugs where the issue descriptions lack information such as clear statements about the observed and expected behavior or steps to reproduce the bug. This shows the importance of providing APR systems with complete and precise problem descriptions in order to minimize the risk of insecure patches. Interestingly, across both Llama and the frameworks, higher word counts in issue descriptions do not strongly distinguish vulnerable from non-vulnerable instances, indicating that the quality and relevance of the information can be a more significant factor than sheer verbosity. Long stack traces, or unrelated debugging conversations do not necessarily help the LLM, and may even divert its attention away from the important details.

At the *project level*, we do not observe strong correlations between vulnerability introduction and repository size, code complexity, maintainability, or contributor count. This may seem counterintuitive as one might expect that larger or more complex projects are inherently harder to repair securely. However, modern issue-tracking and modular design practices often constrain a given fix to a specific feature, module, or service, so many repairs remain local even in large systems. Conversely, a small codebase with tangled responsibilities and high cross-file coupling can force the model to reason across many interacting components even for a relatively simple problem.

Overall, the fact that vulnerability-prone patches correlate with a small set of observable code and task-related characteristics suggests a path toward lightweight, automated risk assessment for APR patches, analogous to prior work

in abnormal commit detection [25, 26, 37, 63]. Simple signals such as how widely a patch is spread across the codebase and the type of available information in an issue description can be combined into a prediction model that scores risk of APR usage *before* and *after* patch generation. In the input, issue-level signals can be used to flag high-risk tasks for APR systems. In the output, an additional set of code-level features can be used to evaluate generated patches. A risk model leveraging these factors would not replace static analysis or code review, but can be integrated into CI/CD pipelines as an early warning and triage mechanism where high-risk APR tasks or patches get routed to stricter security checks or human review. The signals needed for such risk assessment are already present in typical development workflows, making the integration of risk assessment into CI/CD a promising direction for future work.

6 Related Work

Security Analyses of LLM-based APR Systems. Recent work has begun to analyze the security of LLM-based APR agents, rather than only their repair capabilities. Przymus et al. show that carefully crafted **adversarial bug reports** can mislead APR pipelines, inducing CVE reversions, vulnerability injection, CI/CD abuse, and denial-of-service even when some defenses such as AI-assisted reviews are deployed [59]. Their results highlight the bug-report interface as a new attack surface and suggest that current defenses block only a subset of malicious reports.

Chen et al. conduct a complementary red-teaming study of APR agents in realistic CI/CD settings, asking whether seemingly valid issue statements can cause agents to generate vulnerable patches [12]. They challenge the common assumption that tests passing implies safety for APR-generated patches, and demonstrate that agents can be compelled into introducing exploitable behavior while still satisfying existing test suites.

Both works build on a rapidly growing ecosystem of LLM-based APR agents which primarily focus on improving repair success and benchmark performance in repository-scale settings, often without any security checks [19, 30, 64, 81, 85, 88, 90]. In contrast to these adversarial perspectives, our study analyzes LLM and agentic APR outputs under **non-adversarial** SWE-bench issues, quantifying how often vulnerabilities arise by accident and linking them to code and task-level factors such as edit scope and issue information completeness, as well as comparing them directly to developer-written patches.

Security Analyses of LLM-generated Code. Given the growing adoption of LLMs by developers [14, 33, 68], recent studies have begun to examine various aspects of LLM-generated code [18, 32, 43, 82]. While many of these studies have focused on functionality, others have examined the security implications of LLM-generated code [20, 35, 51, 57, 72, 76]. Pearce et al. [57] conducted one of the first assessments on the security of LLM-generated code, showing that even state-of-the-art models can produce vulnerable code in security-sensitive scenarios. Since then, many studies have confirmed these findings by analyzing LLM-generated code across different tasks, models, and user interaction settings [45, 58, 62, 73].

Most prior work has focused on small code snippets or controlled experimental settings [35, 51, 57, 72]. These settings often lack the full project context and complexity inherent in real-world software maintenance tasks, limiting their ability to assess how vulnerabilities emerge in practice. However, a recent study by Fu et al. [20] analyzed Copilot-generated code snippets in GitHub projects and identified several security weaknesses. It is worth mentioning that because these completions are reviewed and integrated by developers, the model is not fully in control of the final output. Our work complements and extends this work by analyzing LLM-generated code in real-world GitHub issues, where models are given autonomy to modify any relevant files. To our knowledge, this is the first large-scale empirical study to link vulnerabilities introduced by LLMs and agentic frameworks to a broad set of contextual factors, including

issue descriptions, code-level attributes, and project structure, in realistic software development settings where models are tasked with resolving issues.

Vulnerability Detection. Automated vulnerability detection can be performed using static or dynamic analysis, with static analysis being widely adopted for large-scale, code-centric studies. static analysis detects weaknesses by reasoning about source code without execution through techniques such as rule/pattern matching, taint and dataflow analysis, and abstract interpretation [84]. Tools like CodeQL, Semgrep, and SpotBugs leverage these techniques with CWE-based queries and rules that support automated evaluations of code [21, 66, 75]. More recent work has explored leveraging LLMs for detection, either directly, by identifying vulnerabilities in code snippets or projects [10, 79], or indirectly by enhancing static analysis pipelines [47, 69]. In the latter case, LLMs have been used to tasks such as inferring vulnerability-specific taint specifications, identifying function call chains, and reasoning over dataflow paths to improve static taint analysis [39, 40, 44]. While LLM-based detection introduces new opportunities, established static analysis tools remain widely used in practice, particularly in CI/CD pipelines, where their deterministic query semantics and reproducibility are often essential.

Abnormal Commit Detection. Past studies have explored various techniques to identify and characterize the risk of potential changes in software systems, particularly focusing on GitHub commits [25, 26, 37, 63]. This line of research will generally aim to identify outlier commits, also known as risky or abnormal commits, through assessing various commit attributes. For instance, Gonzalez et al. introduced Anomalicious [25], an approach to identify malicious commits. By analyzing the commits and identifying the characteristics of a “normal” developer commit, they made an attempt at detecting malicious code contributions.

While these studies provide valuable foundations for identifying risky, malicious, or insecure commits, they have been developed to evaluate human-written code. Our work is the first to assess the characteristics of insecure code produced by LLMs and agentic frameworks. By comparing vulnerable and non-vulnerable generated patches across a range of factors (e.g., scope of modifications, issue type, information completeness) we identify the unique characteristics of high-risk agentic or LLM-generated commits.

7 Conclusions

This study presents the first large-scale security evaluation of an LLM and three agent-generated patches for real-world issue resolution tasks. We find that Llama can introduce up to 11x more new vulnerabilities than developers when addressing issues. Notably, some of the most frequent vulnerabilities introduced by Llama (e.g., CWE-95, CWE-502) differ significantly from those found in developer-written code (e.g., CWE-732, CWE-377), indicating that the model is not merely mimicking human mistakes but also introducing distinct patterns of failure. Further, our analysis of agentic frameworks shows that increasing LLM autonomy can further amplify vulnerability risks, especially in tasks that involve development of new features or tasks that require cross-file reasoning.

Importantly, our findings show that vulnerable outputs are not random. Rather, they are associated with particular code and issue-level characteristics, such as a higher number of modified files, number of unique file types, and the absence of code snippets or certain information in the issue description. These patterns suggest that the riskiness of LLM or agent-generated patches can be assessed not only by detecting specific vulnerable lines of code but also by developing complementary risk assessment techniques that evaluate both the generated code and the broader context in which a patch is proposed e.g., issue type or information completeness. Such systems can complement existing vulnerability detection techniques, often reliant on costly manual reviews, by narrowing the focus to potentially high-risk code contributions.

References

- [1] [n. d.]. Radon: Python tool to compute code metrics. <https://radon.readthedocs.io/en/latest/> Accessed: 2025-05-30.
- [2] 2024. SWE-bench Experiments Repository. <https://github.com/SWE-bench/experiments>. Accessed: 2025-05-30.
- [3] 2025. Replication Package. <https://figshare.com/s/174a976de48f28ae1482?file=57891562>
- [4] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I Maletic. 2008. What's a typical commit? A characterization of open source software repositories. In *2008 16th IEEE international conference on program comprehension*. IEEE, 182–191.
- [5] Areej Algaith, Paulo Nunes, Fonseca Jose, Ilir Gashi, and Marco Vieira. 2018. Finding SQL injection and cross site scripting vulnerabilities with diverse static analysis tools. In *2018 14th European dependable computing conference (EDCC)*. IEEE, 57–64.
- [6] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated unit test improvement using large language models at meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. 185–196.
- [7] Rodrigo Andrade and Vinicius Santos. 2021. Investigating vulnerability datasets. In *Workshop de Visualização, Evolução e Manutenção de Software (VEM)*. SBC, 26–30.
- [8] Anthropic. 2024. The Claude 3 Model Family. <https://www.anthropic.com/news/clause-3-family>. Accessed: 2025-05-05.
- [9] Gemma Catolino, Fabio Palomba, Andy Zaidman, and Filomena Ferrucci. 2019. Not all bugs are the same: Understanding, characterizing, and classifying bug types. *Journal of Systems and Software* 152 (2019), 165–181.
- [10] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering* 48, 9 (2021), 3280–3296.
- [11] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, and Christoph Treude. 2024. An empirical study of static analysis tools for secure code review. In *Proceedings of the 33rd ACM SIGSOFT international symposium on software testing and analysis*. 691–703.
- [12] Simin Chen, Yixin He, Suman Jana, and Baishakhi Ray. 2025. Red Teaming Program Repair Agents: When Correct Patches can Hide Vulnerabilities. arXiv:2509.25894 [cs.SE] <https://arxiv.org/abs/2509.25894>
- [13] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolaos Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. 2024. Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference. arXiv:2403.04132 [cs.AI]
- [14] Kyle Daigle and GitHub Staff. 2025. Survey: The AI wave continues to grow on software development teams. GitHub. https://github.blog/news-insights/research/survey-ai-wave-grows/?utm_source=chatgpt.com Accessed: 2025-05-29.
- [15] Antonio Della Porta, Stefano Lambiase, and Fabio Palomba. 2025. Do Prompt Patterns Affect Code Quality? A First Empirical Assessment of ChatGPT-Generated Code. *arXiv preprint arXiv:2504.13656* (2025).
- [16] Carlo Dimastrogiovanni and Nuno Laranjeiro. 2016. Towards understanding the value of false positives in static code analysis. In *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 119–122.
- [17] Edvin Dyremark and Tobias Elfstrand. 2025. False Positives, Real Problems: Evaluating Static Analysis Tools.
- [18] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53.
- [19] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1469–1481.
- [20] Yujia Fu, Peng Liang, Amjad Tahir, Zengyang Li, Mojtaba Shahin, Jiaxin Yu, and Jinfu Chen. 2023. Security weaknesses of copilot generated code in github. *arXiv preprint arXiv:2310.02059* (2023).
- [21] GitHub. 2022. CodeQL. <https://github.com/github/codeql>.
- [22] GitHub. 2022. GitHub REST API Documentation. <https://docs.github.com/en/rest?apiVersion=2022-11-28> Accessed: 2025-05-05.
- [23] GitHub. n.d.. CodeQL. <https://github.com/github/codeql>. Accessed: 2025-05-05.
- [24] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2021. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software* 172 (2021), 110653.
- [25] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schäfer. 2021. Anomalicious: Automated detection of anomalous and potentially malicious commits on github. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 258–267.
- [26] Raman Goyal, Gabriel Ferreira, Christian Kästner, and James Herbsleb. 2018. Identifying unusual commits on GitHub. *Journal of Software: Evolution and Process* 30, 1 (2018), e1893.
- [27] Zhaoqiang Guo, Tingting Tan, Shiran Liu, Xutong Liu, Wei Lai, Yibiao Yang, Yanhui Li, Lin Chen, Wei Dong, and Yuming Zhou. 2023. Mitigating false positive static analysis warnings: Progress, challenges, and opportunities. *IEEE Transactions on Software Engineering* 49, 12 (2023), 5154–5188.
- [28] Sivana Hamer, Marcelo d'Amorim, and Laurie Williams. 2024. Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers. In *2024 IEEE Security and Privacy Workshops (SPW)*. IEEE, 87–94.
- [29] Hassan Bapeer Hassan, Qusay Idrees Sarhan, and Árpád Beszédes. 2024. Evaluating Python Static Code Analysis Tools Using FAIR Principles. *IEEE Access* 12 (2024), 173647–173659.
- [30] Honeycomb. 2025. Honeycomb. <https://www.honeycomb.sh/>

- [31] Muhui Jiang, Jinan Jiang, Tao Wu, Zuchao Ma, Xiapu Luo, and Yajin Zhou. 2024. Understanding vulnerability inducing commits of the linux kernel. *ACM Transactions on Software Engineering and Methodology* 33, 7 (2024), 1–28.
- [32] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. <https://openreview.net/forum?id=VTF8yNQM6>
- [33] Samia Kabir, David N Udo-Imeh, Bonan Kou, and Tianyi Zhang. 2024. Is stack overflow obsolete? an empirical study of the characteristics of chatgpt answers to stack overflow questions. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. 1–17.
- [34] Rafael Kallis, Andrea Di Sorbo, Gerardo Canfora, and Sebastiano Panichella. 2019. Ticket tagger: Machine learning driven issue classification. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 406–409.
- [35] Raphaël Khouri, Anderson R Avila, Jacob Brunelle, and Baba Mamadou Camara. 2023. How secure is code generated by chatgpt?. In *2023 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE, 2445–2451.
- [36] Lukas Kree, René Helmke, and Eugen Winter. 2024. Using Semgrep OSS to Find OWASP Top 10 Weaknesses in PHP Applications: A Case Study. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 64–83.
- [37] Larissa Leite, Christoph Treude, and Fernando Figueira Filho. 2015. UEDashboard: Awareness of unusual events in commit histories. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 978–981.
- [38] Valentina Lenarduzzi, Fabiano Pecorelli, Nytti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software* 198 (2023), 111575.
- [39] Haonan Li, Hang Zhang, Kexin Pei, and Zhiyun Qian. 2025. The Hitchhiker’s Guide to Program Analysis, Part II: Deep Thoughts by LLMs. *arXiv preprint arXiv:2504.11711* (2025).
- [40] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations*.
- [41] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250.
- [42] Suryadiputra Liawatimena, Harco Leslie Hendric Spits Warnars, Agung Trisetyarso, Edi Abdurahman, Benfano Soewito, Antoni Wibowo, Ford Lumban Gaol, and Bahtiar Saleh Abbas. 2018. Django web framework software metrics measurement using radon and pylint. In *2018 Indonesian Association for Pattern Recognition International Conference (INAPR)*. IEEE, 218–222.
- [43] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971* (2024).
- [44] Puzhuo Liu, Chengnian Sun, Yaowen Zheng, Xuan Feng, Chuan Qin, Yuncheng Wang, Zhenyang Xu, Zhi Li, Peng Di, Yu Jiang, et al. 2025. Llm-powered static binary taint analysis. *ACM Transactions on Software Engineering and Methodology* 34, 3 (2025), 1–36.
- [45] Yue Liu, Thanh Le-Cong, Ratnadira Widayarsi, Chakkrit Tantithamthavorn, Li Li, Xuan-Bach D Le, and David Lo. 2024. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–26.
- [46] Johan Lövgren. 2023. Can Developer Data Predict Vulnerabilities?: Examining Developer and Vulnerability Correlation in the Kibana Project.
- [47] Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. GRACE: Empowering LLM-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software* 212 (2024), 112031.
- [48] Ibéria Medeiros, Nuno F Neves, and Miguel Correia. 2014. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd international conference on World wide web*. 63–74.
- [49] Na Meng, Qianxiang Wang, Qian Wu, and Hong Mei. 2008. An approach to merge results of multiple static analysis tools (short paper). In *2008 The eighth international conference on quality software*. IEEE, 169–174.
- [50] Meta AI. 2025. Llama 3: Open Foundation and Instruction-Tuned Language Models. <https://ai.meta.com/llama/>. Accessed: 2025-05-05.
- [51] Ahmad Mohsin, Helge Janicke, Adrian Wood, Iqbal H Sarker, Leandros Maglaras, and Naeem Janjua. 2024. Can we trust large language models generated code? a framework for in-context learning, security patterns, and code evaluations across diverse llms. *arXiv preprint arXiv:2406.12513* (2024).
- [52] Muhammad Nadeem, Byron J Williams, and Edward B Allen. 2012. High false positive detection of security vulnerabilities: a case study. In *Proceedings of the 50th annual ACM Southeast Conference*. 359–360.
- [53] Paulo Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. 2017. On combining diverse static analysis tools for web security: An empirical study. In *2017 13th European dependable computing conference (EDCC)*. IEEE, 121–128.
- [54] Paulo Jorge Costa Nunes, José Fonseca, and Marco Vieira. 2015. phpSAFE: A security analysis tool for OOP web application plugins. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 299–306.
- [55] OpenAI. 2023. GPT-4 Turbo Model Card. <https://platform.openai.com/docs/models/gpt-4-turbo>. Accessed: 2025-05-05.
- [56] OpenAI. 2025. GPT-4o Model Card. <https://platform.openai.com/docs/models/gpt-4o>. Accessed: 2025-05-05.
- [57] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? assessing the security of github copilot’s code contributions. *Commun. ACM* 68, 2 (2025), 96–105.
- [58] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. 2023. Do users write more insecure code with AI assistants?. In *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*. 2785–2799.

- [59] Piotr Przymus, Andreas Happe, and Jürgen Cito. 2025. Adversarial Bug Reports as a Security Risk in Language Model-Based Automated Program Repair. arXiv:2509.05372 [cs.SE] <https://arxiv.org/abs/2509.05372>
- [60] PyCQA. n.d. Bandit. <https://github.com/PyCQA/bandit>. Accessed: 2025-05-05.
- [61] Fumin Qi, Xiao-Yuan Jing, Xiaoke Zhu, Xiaoyuan Xie, Baowen Xu, and Shi Ying. 2017. Software effort estimation based on open source projects: Case study of Github. *Information and Software Technology* 92 (2017), 145–157.
- [62] Leonardo Criollo Ramírez, Xavier Limón, Ángel J Sánchez-García, and Juan Carlos Pérez-Arriaga. 2024. State of the Art of the Security of Code Generated by LLMs: A Systematic Literature Review. In *2024 12th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, 331–339.
- [63] Christoffer Rosen, Ben Grawi, and Emad Shihab. 2015. Commit guru: analytics and risk prediction of software commits. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 966–969.
- [64] Fernando Vallecillos Ruiz, Max Hort, and Leon Moonen. 2025. The Art of Repair: Optimizing Iterative Program Repair with Instruction-Tuned Models. *arXiv preprint arXiv:2505.02931* (2025).
- [65] Amirali Sajadi, Binh Le, Anh Nguyen, Kostadin Damevski, and Preetha Chatterjee. 2025. Do LLMs consider security? an empirical study on responses to programming questions. *Empirical Software Engineering* 30, 3 (2025), 101.
- [66] Semgrep. n.d. Semgrep. <https://github.com/semgrep/semgrep>. Accessed: 2025-05-05.
- [67] Muhammad Usman Shahid, Chaudhry Mujeeb Ahmed, and Rajiv Ranjan. 2025. LLM-CSEC: Empirical Evaluation of Security in C/C++ Code Generated by Large Language Models. *arXiv preprint arXiv:2511.18966* (2025).
- [68] Inbal Shani and GitHub Staff. 2023. *Survey reveals AI's impact on the developer experience*. GitHub. <https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/> Accessed: 2025-05-29.
- [69] Ze Sheng, Zhicheng Chen, Shuning Gu, Heqing Huang, Guofei Gu, and Jeff Huang. 2025. LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. *arXiv preprint arXiv:2502.07049* (2025).
- [70] Mohammed Latif Siddiq, Joanna Cecilia da Silva Santos, Sajith Devareddy, and Anna Muller. 2024. Sallm: Security assessment of generated code. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. 54–65.
- [71] Mohammed Latif Siddiq, Joanna Cecilia Da Silva Santos, Ridwanul Hasan Tanvir, Noshin Ulfat, Fahmid Al Rifat, and Vinícius Carvalho Lopes. 2024. Using large language models to generate junit tests: An empirical study. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 313–322.
- [72] Mohammed Latif Siddiq, Lindsay Roney, Jiahao Zhang, and Joanna Cecilia Da Silva Santos. 2024. Quality assessment of chatgpt generated code and their use by developers. In *Proceedings of the 21st International Conference on Mining Software Repositories*. 152–156.
- [73] Mohammed Latif Siddiq and Joanna CS Santos. 2023. Generate and pray: Using sallms to evaluate the security of llm generated code. *arXiv preprint arXiv:2311.00889* (2023).
- [74] Yang Song and Oscar Chaparro. 2020. Bee: A tool for structuring and analyzing bug reports. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1551–1555.
- [75] SpotBugs. n.d. SpotBugs. <https://spotbugs.github.io/>.
- [76] Rebeka Tóth, Tamas Bisztray, and László Erdődi. 2024. LLMs in Web Development: Evaluating LLM-Generated PHP Code Unveiling Vulnerabilities and Limitations. In *International Conference on Computer Safety, Reliability, and Security*. Springer, 425–437.
- [77] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2023. Are automated static analysis tools worth it? An investigation into relative warning density and external software quality on the example of Apache open source projects. *Empirical Software Engineering* 28, 3 (2023), 66.
- [78] UC Berkeley SkyLab and LM Arena. 2024. Chatbot Arena Leaderboard. <https://lmarena.ai/>. Accessed: 2025-05-05.
- [79] Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. 2024. Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *2024 IEEE symposium on security and privacy (SP)*. IEEE, 862–880.
- [80] Jixin Wang, Xitong Luo, Liuwen Cao, Hongkui He, Hailin Huang, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2024. Is Your AI-Generated Code Really Safe? Evaluating Large Language Models on Secure Code Generation with CodeSecEval. arXiv:2407.02395 [cs.SE] <https://arxiv.org/abs/2407.02395>
- [81] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. arXiv:2407.16741 [cs.SE] <https://arxiv.org/abs/2407.16741>
- [82] You Wang, Michael Pradel, and Zhongxin Liu. 2025. Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. *arXiv preprint arXiv:2503.15223* (2025).
- [83] Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. 2023. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 172–184.
- [84] Dave Wichers, itamarlavender, will obrien, Eitan Worcel, Prabhu Subramanian, kingthorin, coadaflorin, hblanksanship, GovorovViva64, pfhorman, GouveaHeitor, Clint Gibler, DSotnikov, Ajin Abraham, Noam Rathaus, and Mike Jang. n.d. Source Code Analysis Tools. https://owasp.org/www-community/Source_Code_Analysis_Tools. Accessed: 2025-05-05.

- [85] Boyang Yang, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, Bach Le, and Haoye Tian. 2025. Enhancing repository-level software repair via repository-aware knowledge graphs. *arXiv preprint arXiv:2503.21710* (2025).
- [86] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. <https://arxiv.org/abs/2405.15793>
- [87] Yifan Yao, Jinhao Duan, Kaidi Xu, Yuanfang Cai, Zhibo Sun, and Yue Zhang. 2024. A survey on large language model (llm) security and privacy: The good, the bad, and the ugly. *High-Confidence Computing* (2024), 100211.
- [88] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) (ISSTA 2024). Association for Computing Machinery, New York, NY, USA, 1592–1604. [doi:10.1145/3650212.3680384](https://doi.org/10.1145/3650212.3680384)
- [89] Ying Zhang, Wenjia Song, Zhengjie Ji, Na Meng, et al. 2023. How well does LLM generate security tests? *arXiv preprint arXiv:2310.00710* (2023).
- [90] Juuang Zhao, Donghao Yang, Li Zhang, Xiaoli Lian, Zitian Yang, and Fang Liu. 2024. Enhancing automated program repair with solution design. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 1706–1718.
- [91] Xin Zhou, Duc-Manh Tran, Thanh Le-Cong, Ting Zhang, Ivana Clairine Irsan, Joshua Sumarlin, Bach Le, and David Lo. 2024. Comparison of static application security testing tools and large language models for repo-level vulnerability detection. *arXiv preprint arXiv:2407.16235* (2024).