Are AI-Generated Fixes Secure? Analyzing LLM and Agent Patches on SWE-bench

Amirali Sajadi
Drexel University
Philadelphia, PA, USA
amirali.sajadi@drexel.edu

Kostadin Damevski
University of Richmond
Richmond, VA, USA
kdamevski@vcu.edu

Preetha Chatterjee
Drexel University
Philadelphia, PA, USA
preetha.chatterjee@drexel.edu

***Abstract*—Large Language Models (LLMs) and their agentic frameworks are increasingly adopted to automate software development tasks such as issue resolution and program repair. While prior work has identified security risks in LLM-generated code, most evaluations have focused on synthetic or isolated settings, leaving open questions about the security of these systems in real-world development contexts. In this study, we present the first large-scale security analysis of LLM-generated patches using 20,000+ issues from the SWE-bench dataset. We evaluate patches produced by a standalone LLM (Llama 3.3) and compare them to developer-written patches. We also assess the security of patches generated by three top-performing agentic frameworks (OpenHands, AutoCodeRover, HoneyComb) on a subset of our data. Finally, we analyze a wide range of code, issue, and project-level factors to understand the conditions under which LLMs and agents are most likely to generate insecure code. Our findings reveal that the standalone LLM introduces nearly 9x more new vulnerabilities than developers, with many of these exhibiting unique patterns not found in developers' code. Agentic workflows also generate a significant number of vulnerabilities, particularly when granting LLMs more autonomy, potentially increasing the likelihood of misinterpreting project context or task requirements. We find that vulnerabilities are more likely to occur in LLM patches associated with a higher number of files, more lines of generated code, and GitHub issues that lack specific code snippets or information about the expected code behavior and steps to reproduce. These results suggest that contextual factors play a critical role in the security of the generated code and point toward the need for proactive risk assessment methods that account for both code and issue-level information to complement existing vulnerability detection tools.**

***Index Terms*—LLMs, Agentic Frameworks, Security, Issue Resolution**

I. Introduction

LLMs are rapidly transforming software development, assisting with tasks like code completion, test generation, and bug fixing [1]–[5]. They are also increasingly integrated into agentic frameworks that enable autonomous planning and execution of complex tasks, such as issue resolution, where LLM-based agents identify, diagnose, and patch software bugs directly in real-world projects [6]–[8]. This widespread adoption of LLMs raises critical concerns about their potential security implications [9]–[12]. Studies show that 30-50% of code generated with AI code assistants contains vulnerabilities [13]–[15]. As LLMs are increasingly entrusted with making direct commits to production repositories, the risk of introducing vulnerabilities grows. Even a single vulnerable commit, as seen in the Log4Shell exploit, can have far-reaching and costly consequences [16]. Thus, there is an urgent need to understand and mitigate the security risks posed by LLM-based systems.

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 [13], [17]–[19]. These scenarios lack the complexity and context of issue resolution in real projects, where understanding environment constraints or project-specific dependencies is critical. Additionally, when LLMs are embedded in agentic frameworks that automate repair pipelines, these risks may be amplified [20]–[22]. As a result, it remains unclear how secure LLM-generated patches are when fixing real-world issues. Although recent benchmarks like SWE-bench [23] have enabled the assessment of LLMs on real-world software engineering tasks, security-specific analyses in these contexts remain limited. Such evaluations are essential for understanding the circumstances under which vulnerabilities emerge in LLM-generated bug fixes.

In this work, we conduct an empirical study of LLM and agent-generated code in real-world issue resolution tasks, with a focus on security. Using the SWE-bench dataset, we analyze patches generated both by a standalone LLM and three agentic frameworks, comparing them to developer-written patches. Our goal is not only to assess whether LLMs and agentic frameworks introduce security vulnerabilities in realistic settings, but also to understand the conditions under which this occurs. We aim to identify the characteristics of risky LLM and agent-generated contributions and contexts that lead to insecure outputs. We ask the following research questions:

- **RQ1**: How secure are the patches produced by standalone LLMs 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?

To answer these questions, we evaluate the vulnerability prevalence, type, and overlaps in: 1) *Developer-written* patches, 2) *Standalone LLM* patches (Llama 3.3 70B [24]), and 3) *Agentic APR Framework* patches (OpenHands [6],

AutoCodeRover [8], and Honeycomb [25]). Using a majority voting technique to suppress false positives across three static analysis tools (Semgrep [26], Bandit [27], and CodeQL [28]), we identify the new vulnerabilities introduced through the patches. Lastly, we examine a comprehensive set of of code, issue, and project-level metrics to assess the conditions under which LLMs are most likely to introduce vulnerabilities.

Our findings highlight the security risks of LLM-generated code at scale. When evaluated on 20,000+ GitHub issues, the standalone LLM introduced 185 new vulnerabilities i.e., nearly **9x higher** than the 20 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-327) were almost entirely 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 generated lines of code. Further, most 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 contexts even before the code is reviewed. We make our replication package publicly available to support further research and validation of our findings [29].

II. 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) and three agentic frameworks (OpenHands, AutoCodeRover, HoneyComb). 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 associated with the presence of these vulnerabilities (RQ3).

A. Dataset and Task Setup

We leverage the SWE-bench dataset [23], 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 this study, we analyze two types of generated patches: those produced by a standalone LLM (Llama) and those generated by three top-performing agentic LLM frameworks (OpenHands, AutoCodeRover, HoneyComb).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.

B. Patch Generation with LLM and Agentic Frameworks

1) ***RQ1.*** *Llama 3.3 Instruct (70B):* To examine the *security risks of patches produced by standalone LLMs*, we generate candidate patches across the entire dataset (train+test) using the Llama 3.3 Instruct 70B model [24]. Recent benchmarks indicate that this model's code generation capabilities are comparable to those of GPT-4o [30] and exceed the performance of Claude 3 Opus [31] and GPT-4 Turbo [32], making it a strong choice for our study [33], [34].

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 [23]. 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 [35], in standalone LLM settings, it often yields syntactically invalid or partially applicable patches, even after post-processing [23]. 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

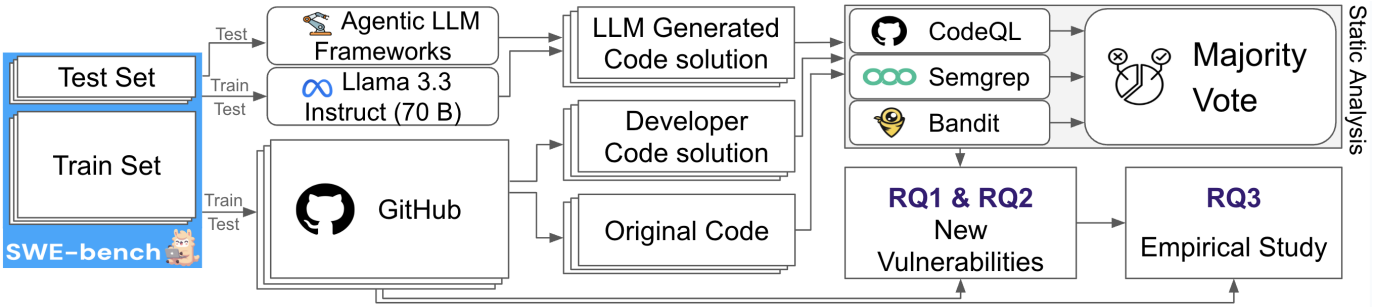

Fig. 1. Overview of Our Methodology for LLM-Based Code Generation and Security Evaluation.

runnable code, not diffs. For this study, we prioritize reliability over minimal token usage.

*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)* [6], *AutoCodeRover-v2.0 (Claude-3.5-Sonnet-20241022)* [8], and *Honeycomb* [25]. These models resolved 29.38%, 24.89%, and 22.06% of the SWE-bench test set issues, respectively. We selected them based on their resolution rates at the time of our study and the availability of their generated patches on the SWE-bench official repositories [36]. 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.

C. 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 [37]. 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.

D. Vulnerability Detection Using Majority Voting

Manually analyzing the security of 20,000+ patches is infeasible. Therefore, we rely on three established OWASP-recommended [38] static analysis tools to automate the vulnerability detection process. The following tools were selected based on their widespread use [12], [18], [39], support for both rule-based and taint analysis, and their capacity to detect a wide range of real-world vulnerabilities [40]:

CodeQL [28] 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 [26] 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 [27] is a heuristic-based and security-focused static analysis tool designed for Python projects. Bandit inspects abstract syntax trees to detect common security issues.

A common concern with static analysis tools is their limited accuracy, particularly their tendency to produce false positives [41]–[43]. Therefore, to improve reliability and mitigate tool-specific false positives, we adopt a *majority voting strategy*: a vulnerability is accepted if at least two of the three tools detect it in the same file. Prior work has shown that static analysis tools often report distinct sets of vulnerabilities with limited overlap, and some studies have explored combining results via union or hybrid strategies to improve coverage [44], [45]. In contrast, our approach prioritizes precision over recall; by requiring agreement among tools, we aim to reduce false positives and improve the confidence in our findings. While this majority-vote strategy may miss some true positives, it offers a more conservative and reliable performance.

We focus on identifying only the *new vulnerabilities* introduced by LLM-generated code. By comparing original and modified files, we distinguish vulnerabilities introduced by the LLM from those already present. This allows us to isolate the security risks attributable to LLM-generated patches.

We apply the same static analysis tools and majority vote aggregation process to detect vulnerabilities across all sources, LLMs, agentic frameworks, and developer-written patches.

E. 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).

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) 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 [46]–[49], 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 [50]. 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 numbers of files can indicate broader changes and a higher overall risk [46]–[49].

 # Unique file types modified: Modifying diverse file types may introduce different complexities to the issue resolution process [46], [49].

 # Sensitive file modifications: Inspired by prior work on anomaly detecting malicious commits [49], 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 [14], [51]. 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 [52], [53]. Further, for issues categorized as bugs, we classify the bug type using a comprehensive, empirically grounded, and widely-adopted taxonomy [54]. 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 [53], 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 it as an 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, during which disagreements were resolved. 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 [55]. Further, size of a project has been shown to have direct associations with the number of potential vulnerabilities [50].

Cyclomatic Complexity and Maintainability Index of Project: We use Radon [56], a popular code analysis tool [57], [58], 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.

III. Results

A. RQ1: Security of patches generated by developers vs. standalone LLM

Vulnerability Prevalence: Table I presents the number of files with new vulnerabilities introduced by *Developer* 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 (20.1%) passed the majority vote filter. In contrast, developer patches for the full dataset, triggered 1398 alerts (primarily driven by Bandit's high sensitivity) but only 20 vulnerabilities (2%), identified in 17 files, were retained after majority vote filtering.

Source	Semgrep	CodeQL	Bandit	Majority Vote
Llama 3.3 (Train+Test)	323	62	1055	185
Developers (Train+Test)	9	41	1348	17
OH (Test)	447	64	1383	195
ACR (Test)	5	3	12	3
HC (Test)	19	3	37	2

TABLE I
NUMBER OF VULNERABILITIES DETECTED PER TOOL. MAJORITY VOTE PRESENTS THE NUMBER OF FILES WITH AT LEAST ONE VULNERABILITY.

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 Bandit produces most false positives, consistent with prior work [59].

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 patch generation. 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: As shown in Table II, the most common vulnerabilities in Llama-generated patches include command injection (CWE-78, 97 instances), eval injection (CWE-95, 54), insecure deserialization (CWE-502, 21), path traversal (CWE-22, 19), and weak cryptography (CWE-327, 12). While less frequent, other vulnerability types like incorrect permission assignment or error message exposure also appeared in the LLM patches. The full set of vulnerabilities are included in our replication package.

Llama-Generated Code	
CWE-78 (OS Command Injection)	97
CWE-95 (Eval Injection)	54
CWE-502 (Insecure Deserialization)	21
CWE-22 (Path Traversal)	19
CWE-327 (Weak Crypto)	12
Developer-Written Code	
CWE-78 (OS Command Injection)	4
CWE-732 (Permission Misconfiguration)	4
CWE-502 (Insecure Deserialization)	3
CWE-22 (Path Traversal)	3
CWE-377 (Insecure Temporary File)	3
CWE-703 (Exception Handling)	3

TABLE II
MOST RECURRING CWE TYPES FOUND IN LLAMA-GENERATED AND DEVELOPER-WRITTEN PATCHES ACROSS THE TRAIN & TEST SETS.

Developer-written patches showed far fewer vulnerabilities overall, but some of the top categories i.e., command injection (CWE-78, 4), deserialization (CWE-502, 3), and path traversal (CWE-22, 3), are similar to the LLM results. These three 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 entirely absent from developer-written patches. Notably, eval injection (CWE-95, n=54 instances) and the use of weak cryptographic algorithms (CWE-327, n=12) ranked among the most frequent LLM vulnerabilities. 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, standalone LLMs may prefer 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']
```

RQ1 Summary: Overall, we observed that the standalone LLM introduced up to **9× more** vulnerabilities into the codebase than developers when resolving real-world issues. This highlights a significant risk associated with deploying LLMs in software development pipelines without careful security auditing. Moreover, none of the specific vulnerabilities found in Llama-generated code overlap with those in developer code (either in terms of affected files or associated issues,) indicating that LLMs can introduce distinct vulnerabilities.

B. RQ2: Security of patches generated by the agentic APR workflows

Vulnerability Prevalence: Table I 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 apply the majority vote strategy to retain only high-confidence vulnerabilities.

Among the agentic frameworks evaluated on the SWE-bench test set, OH produced 195 vulnerabilities (after majority vote filtering), while ACR and HC each produced 3 and 2 vulnerabilities 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, 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.

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`). 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 148 vulnerabilities in OH originated from just 15 issue instances. In fact, three issues alone, *django ticket #27854*, *sphinx issue #8870*, and *pylint issue #4421*, accounted for 124 of the 148 remaining vulnerabilities. In the most extreme case, *django ticket #27854*, 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 reimplementation 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 OpenHand's 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 148 vulnerabilities, exhibited a significantly similar vulnerability pattern to that of the standalone LLM: command injection (CWE-78, n=36), insecure deserialization (CWE-502, n=29), and eval injection (CWE-95, n=21) were all among the most common CWEs in OH patches. This alignment suggests that even when LLMs are embedded within frameworks, the 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 almost entirely different from those written by the developers. These results echo the same observations in III-A: While developers, standalone LLMs, and agentic frameworks all can add vulnerable 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 only 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.

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

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 III 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 IV.

Metric (Llama)	Vuln Mean	All Mean	p-value	Cliff's δ
Files Added or Modified (All)	1.72	1.25	<0.001	0.225
Files Added or Modified (Python)	1.60	1.14	<0.001	0.234
Unique File Types Modified	1.11	1.00	0.0002	0.096
Sensitive Files Modified	0.005	0.007	0.919	-0.001
LOC Generated by LLM (Python)	106.56	81.84	<0.001	0.265
Avg. Cyclomatic Complexity of LLM-Generated Code	2.88	3.03	0.7051	-0.019

TABLE III
SUMMARY STATISTICS COMPARING VULNERABLE VS. ALL LLAMA/AGENTIC-GENERATED CODE FOR CODE-LEVEL METRICS.

Metric	OH		ACR		HC	
	V	A	V	A	V	A
Files Added or Modified (All)	158.40	11.38	1.33	1.07	1.67	1.46
Files Added or Modified (Python)	68.83	4.46	1.33	1.07	1.67	1.40
Unique File Types Modified	2.79	1.41	1.00	0.98	1.00	0.90
Sensitive Files Modified	0.191	0.013	0.00	0.00	0.00	0.004
LOC Generated by LLM (Python)	997.98	1614.53	387.33	1294.69	2406.00	1764.98
Avg. Cyclomatic Complexity of LLM-Generated Code	2.79	1.41	1.00	0.98	1.00	0.90

TABLE IV
MEAN VALUES FOR VULNERABLE (V) VS. ALL (A) AGENTIC-GENERATED PATCHES ACROSS CODE-LEVEL METRICS.

 # Files Added or Modified: As shown in Table III, 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. 1.14), with statistically significant differences ($p < 0.001$, Cliff's $\delta = 0.23$).

Among the agentic frameworks, OH showed an even more dramatic pattern: vulnerable instances modified 158.40 files on average (vs. 11.38), including Python files (68.83 vs. 4.46), with effect size of δ=0.42 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 (1.67 vs. 1.46) showed higher means for vulnerable instances.

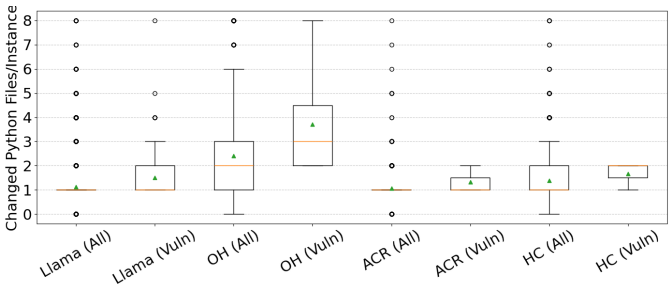

Fig. 2. Number of changed files per instance (capped at 10). Vulnerable cases show greater variability and significantly higher averages than the general set.

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 = 1.11 vs. 1.00). This difference is statistically significant (p = 0.0002), although the effect size is negligible (Cliff's δ = 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 (2.79 vs. 1.41), with statistical significance and meaningful effect size (δ = 0.153, p = 0.0031). 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 = 0.919), indicating that LLMs do 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 (0.19 vs. 0.01, $p < 0.001$), suggesting that its more expansive editing behavior may lead it to touch more sensitive files. However, the effect size remained negligible (δ = 0.076), 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 more extensive edits (106.6 vs. 81.8 lines on average), a statistically significant difference ($p < 0.001$) with a positive effect size ($\delta = 0.265$). This suggests that larger code generations are more prone to vulnerabilities.

Among the agentic frameworks, ACR shows the strongest contrast, with vulnerable outputs averaging fewer lines (387 vs. 1295). OH also shows a reduction in generated LOC for vulnerable instances (998 vs. 1615), with a small effect size (δ = -0.148, p = 0.082). In contrast, HC exhibits the opposite trend, with vulnerable code averaging more lines (2406 vs. 1765). These contrasting patterns suggest that while the scale of generation is 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. 3.03), but this difference is not statistically significant (p = 0.7051, δ = -0.019), suggesting that structurally simple code may still contain security flaws.

Among agentic frameworks, OH shows a similar trend, with vulnerable files having lower complexity (3.63 vs. 4.51), though the difference is not statistically significant (p = 0.074, δ = -0.085). Further, HC shows a slight increase in the complexity of vulnerable files (5.28 vs. 4.36). Notably, ACR exhibits a significant increase in complexity of vulnerable files (7.35 vs. 4.30). 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 (Llama, OH) and file types (OH), and involve more generated code (Llama, HC). Higher-risk patches in ACR exhibit abnormally high code complexity.

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, 138 out of 185 (73.4%) were classified as *Bug*, 40 (21.27%) as *Feature*, and 3 (1.59%) as *Question*. In contrast, OH had 10 *Bugs* and 5 *Feature*; ACR had 2 *Bugs* and 1 *Feature*; and HC had 2 *Bugs*.

Table V 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 the standalone LLM and 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	82	7	2	1
Configuration Issue	27	2	–	–
Performance Issue	6	–	–	–
GUI-Related Issue	4	–	–	–
Test Code-Related Issue	3	1	–	–
Permission/Deprecation Issue	2	–	–	–
Security Issue	2	–	–	1
Network Issue	1	–	–	–
Miscellaneous/Other Issue	11	–	–	–

TABLE V
BUG TYPE DISTRIBUTION ACROSS VULNERABLE INSTANCES.

Information Completeness: Among vulnerable LLM cases, nearly all (136/138) bug-related issues included *observed behavior*, but just about 65.9% (91/138) included *steps to reproduce*, and only 49.2% (68/138) 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 18 unique GitHub issues for agentic workflows (15 for OH, 3 for ACR, 2 for HC), we observe a high rate of information completeness for OH instances: all 10 vulnerable issue reports included observed behavior, 8 included expected behavior, and 9 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 112.55 words for vulnerable instances. Neither the standalone word count of problem descriptions (p = 0.753, Cliff's δ = 0.013) nor the combined word count including comments (p = 0.191, δ = 0.056) 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 sheer verbosity.

Across the frameworks, we observed mixed patterns. OH and ACR had higher average word counts when combining issue and comment text (355.1 and 435.7 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 (4.14) compared to all issues (3.57), but the difference was not significant (p = 0.22, δ = 0.053), indicating that comment count

does not possibly distinguish vulnerable from non-vulnerable LLM-generated patches.

Among the frameworks, OH had significantly fewer comments than the general set ($p < 0.001$, δ = -0.315), while ACR and HC showed no statistically significant differences (p = 0.6256 and p = 0.2324, respectively). While the number of comments is not a strong predictor overall, certain frameworks may be more sensitive to the availability, or absence, of prior discussion when generating code.

Presence of Code Snippets: We observe that only 48.1% 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 (χ^2 = 7.85, p = 0.0051), 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 (ϕ = 0.019), 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 LLMs are 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.

OH shows an even stronger pattern (χ^2 = 32.29, p $<$ 0.0001, ϕ = 0.039), suggesting that missing code context may be particularly harmful for more autonomous systems. While ACR had no vulnerable issues with code snippets (0 out of 3), both HC issues contained code snippets (2 out of 2). Overall, these trends indicate that the potential ambiguities in issues lacking code snippets can increase the risk of vulnerabilities.

Issue-Level Factors

Insecure patches 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 without code snippets.

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 ($\rho = 0.188$, p = 0.206). Similarly, we found no meaningful correlations in any of the agentic frameworks: OH ($\rho = 0.171$, p = 0.251), 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 ($\rho = -0.082$, p = 0.583) and complexity ($\rho = -0.064$, p = 0.670) were not correlated with vulnerability introduction. The same holds for the frameworks. For example, in OH, maintainability ($\rho = 0.098$, p = 0.512) and complexity ($\rho = -0.007$, p = 0.962) 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 LLM-generated code, the association was weak and statistically insignificant ($\rho = 0.166$, p = 0.2635). The same held true across all agentic frameworks: OH ($\rho = 0.161$, p = 0.280), 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 do not reliably predict insecure LLM-generated code.

RQ3: Summary of Findings. Our analysis of over 20,000 LLM-generated patches reveals that vulnerability-prone outputs are associated with a consistent set of characteristics, particularly at the *code* and *issue level*. At the code level, vulnerable patches tend to involve broader and more scattered edits, modifying more files and generating more lines of code, especially in more autonomous frameworks like OH. While unique file types and sensitive file modifications showed some variation, their predictive value remains limited across all systems. Cyclomatic complexity was not a strong differentiator of insecure code, except in ACR, where vulnerable patches had significantly higher complexity.

At the issue level, vulnerabilities were frequently found in tasks labeled as bugs, particularly those lacking full contextual descriptions (e.g., missing expected behavior or reproduction steps). However, for both the standalone LLM and frameworks, higher word counts did not strongly distinguish vulnerable from non-vulnerable cases. A key trend was that issues lacking concrete code snippets were more frequently linked to insecure patches, highlighting the importance of prompts with specific code snippets for secure patch generation.

At the project level, no strong correlation was found between vulnerability introduction and repository size, complexity, maintainability, or contributor count. These results suggest

that security risks are impacted more by how LLMs interact with individual tasks and contextual cues than by the broader structural properties and attributes of the repository.

Overall, vulnerabilities in LLM-generated patches are most strongly associated with *code-level* factors like broader edits and file changes, and *issue-level* factors such as missing contextual information, especially the absence of code snippets. In contrast, *project-level* attributes show no clear relationship with vulnerability outcomes.

IV. Threats to Validity

Our security analysis relies on static analysis tools, which are known to produce false positives and vary in coverage. To mitigate this threat, we used three tools (Semgrep, Bandit, and CodeQL) and applied a majority voting technique to reduce false positives and increase confidence in the identified vulnerabilities. This approach deliberately trades off recall for precision; while we detect fewer vulnerabilities than we might using a single tool or a union-based strategy, we gain higher confidence that the findings represent true security issues. Nevertheless, we acknowledge that some false positives or missed vulnerabilities may remain.

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. Further, 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 of our agentic study in terms of data scale and have adjusted our conclusions accordingly. For example, 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.

Lastly, manual 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) through discussions.

V. Related Work

Security of LLM-generated Code. Given the growing adoption of LLMs by developers [60]–[62], recent studies have begun to examine various aspects of LLM-generated code [23], [35], [63], [64]. While many of these studies have focused on functionality, others have examined the security implications of LLM-generated code [13], [14], [17]–[19], [51]. Pearce et al. [13] 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 [9], [12], [65], [66].

Most prior work has focused on small code snippets or controlled experimental settings [13], [17]–[19]. 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. [14] 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.

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 [46], [48], [49], [67]. 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 [49], 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.

VI. Conclusions

This study presents the first large-scale security evaluation of LLM and agent-generated patches for real-world issue resolution tasks. We find that standalone LLMs can introduce up to 9× more new vulnerabilities than developers when addressing issues. Notably, some of the most frequent vulnerabilities introduced by LLMs (e.g., CWE-95, CWE-327) differ significantly from those found in developer-written code (e.g., CWE-732, CWE-377), indicating that LLMs are 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 are underspecified or require an understanding of the existing project context.

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, longer generated code, 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] Z. Li, C. Wang, Z. Liu, H. Wang, D. Chen, S. Wang, and C. Gao, "Cctest: Testing and repairing code completion systems," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1238–1250.

[2] M. L. Siddiq, J. C. Da Silva Santos, R. H. Tanvir, N. Ulfat, F. Al Rifat, and V. Carvalho Lopes, "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*, 2024, pp. 313–322.

[3] Y. Zhang, W. Song, Z. Ji, N. Meng *et al.*, "How well does llm generate security tests?" *arXiv preprint arXiv:2310.00710*, 2023.

[4] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, 2024, pp. 185–196.

[5] Y. Wei, C. S. Xia, and L. Zhang, "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*, 2023, pp. 172–184.

[6] X. Wang, B. Li, Y. Song, F. F. Xu, X. Tang, M. Zhuge, J. Pan, Y. Song, B. Li, J. Singh, H. H. Tran, F. Li, R. Ma, M. Zheng, B. Qian, Y. Shao, N. Muennighoff, Y. Zhang, B. Hui, J. Lin, R. Brennan, H. Peng, H. Ji, and G. Neubig, "OpenHands: An Open Platform for AI Software Developers as Generalist Agents," 2024. [Online]. Available: https://arxiv.org/abs/2407.16741

[7] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. [Online]. Available: https://arxiv.org/abs/2405.15793

[8] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1592–1604. [Online]. Available: https://doi.org/10.1145/3650212.3680384

[9] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" in *Proceedings of the 2023 ACM SIGSAC conference on computer and communications security*, 2023, pp. 2785–2799.

[10] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (llm) security and privacy: The good, the bad, and the ugly," *High-Confidence Computing*, p. 100211, 2024.

[11] A. Sajadi, B. Le, A. Nguyen, K. Damevski, and P. Chatterjee, "Do llms consider security? an empirical study on responses to programming questions," *Empirical Software Engineering*, vol. 30, no. 3, p. 101, 2025.

[12] M. L. Siddiq and J. C. Santos, "Generate and pray: Using sallms to evaluate the security of llm generated code," *arXiv preprint arXiv:2311.00889*, 2023.

[13] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[14] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security weaknesses of copilot generated code in github," *arXiv preprint arXiv:2310.02059*, 2023.

[15] J. Wang, X. Luo, L. Cao, H. He, H. Huang, J. Xie, A. Jatowt, and Y. Cai, "Is your ai-generated code really safe? evaluating large language models on secure code generation with codeseceval," 2024. [Online]. Available: https://arxiv.org/abs/2407.02395

[16] Cybersecurity and Infrastructure Security Agency (CISA), "Mitigating Log4Shell and Other Log4j-Related Vulnerabilities," https://www.cisa.gov/news-events/cybersecurity-advisories/aa21-356a, 2021, accessed: 2025-05-28.

[17] A. Mohsin, H. Janicke, A. Wood, I. H. Sarker, L. Maglaras, and N. Janjua, "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.

[18] M. L. Siddiq, L. Roney, J. Zhang, and J. C. D. S. Santos, "Quality assessment of chatgpt generated code and their use by developers," in *Proceedings of the 21st International Conference on Mining Software Repositories*, 2024, pp. 152–156.

[19] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by chatgpt?" in *2023 IEEE international conference on systems, man, and cybernetics (SMC)*. IEEE, 2023, pp. 2445–2451.

[20] R. Khan, S. Sarkar, S. K. Mahata, and E. Jose, "Security threats in agentic ai system," 2024. [Online]. Available: https://arxiv.org/abs/2410.14728

[21] K. Huang, "Agentic ai threat modeling framework: Maestro," https://cloudsecurityalliance.org/blog/2025/02/06/agentic-ai-threat-modeling-framework-maestro, Feb. 2025, accessed: 2025-05-01.

[22] J. Chen and R. Lu, "Ai agents are here. so are the threats," *Unit 42, Palo Alto Networks*, May 2025. [Online]. Available: https://unit42.paloaltonetworks.com/agentic-ai-threats/

[23] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can language models resolve real-world github issues?" in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[24] Meta AI, "Llama 3: Open foundation and instruction-tuned language models," https://ai.meta.com/llama/, 2025, accessed: 2025-05-05.

[25] Honeycomb, "Honeycomb," 2025. [Online]. Available: https://www.honeycomb.sh/

[26] Semgrep, "Semgrep," https://github.com/semgrep/semgrep, n.d., accessed: 2025-05-05.

[27] PyCQA, "Bandit," https://github.com/PyCQA/bandit, n.d., accessed: 2025-05-05.

[28] GitHub, "Codeql," https://github.com/github/codeql, n.d., accessed: 2025-05-05.

[29] "Replication package," 2025. [Online]. Available: https://figshare.com/s/84a2649f26c971b8c581

[30] OpenAI, "Gpt-4o model card," https://platform.openai.com/docs/models/gpt-4o, 2025, accessed: 2025-05-05.

[31] Anthropic, "The claude 3 model family," https://www.anthropic.com/news/claude-3-family, 2024, accessed: 2025-05-05.

[32] OpenAI, "Gpt-4 turbo model card," https://platform.openai.com/docs/models/gpt-4-turbo, 2023, accessed: 2025-05-05.

[33] UC Berkeley SkyLab and LMArena, "Chatbot arena leaderboard," https://lmarena.ai/, 2024, accessed: 2025-05-05.

[34] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez, and I. Stoica, "Chatbot arena: An open platform for evaluating llms by human preference," 2024.

[35] Y. Wang, M. Pradel, and Z. Liu, "Are" solved issues" in swe-bench really solved correctly? an empirical study," *arXiv preprint arXiv:2503.15223*, 2025.

[36] "Swe-bench experiments repository," https://github.com/SWE-bench/experiments, 2024, accessed: 2025-05-30.

[37] GitHub, "Github rest api documentation," 2022, accessed: 2025-05-05. [Online]. Available: https://docs.github.com/en/rest?apiVersion=2022-11-28

[38] D. Wichers, itamarlavender, will obrien, E. Worcel, P. Subramanian, kingthorin, coadaflorin, hblankenship, GovorovViva64, pfhorman, GouveaHeitor, C. Gibler, DSotnikov, A. Abraham, N. Rathaus, and M. Jang, "Source code analysis tools," https://owasp.org/www-community/Source_Code_Analysis_Tools, n.d., accessed: 2025-05-05.

[39] L. Kree, R. Helmke, and E. Winter, "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, 2024, pp. 64–83.

[40] Z. Li, S. Dutta, and M. Naik, "Iris: Llm-assisted static analysis for detecting security vulnerabilities," in *The Thirteenth International Conference on Learning Representations*, 2025.

[41] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou, "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities," *IEEE Transactions on Software Engineering*, vol. 49, no. 12, pp. 5154–5188, 2023.

[42] C. Dimastrogiovanni and N. Laranjeiro, "Towards understanding the value of false positives in static code analysis," in *2016 Seventh Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 2016, pp. 119–122.

[43] M. Nadeem, B. J. Williams, and E. B. Allen, "High false positive detection of security vulnerabilities: a case study," in *Proceedings of the 50th annual ACM Southeast Conference*, 2012, pp. 359–360.

[44] I. Medeiros, N. F. Neves, and M. Correia, "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*, 2014, pp. 63–74.

[45] P. J. C. Nunes, J. Fonseca, and M. Vieira, "phpsafe: A security analysis tool for oop web application plugins," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2015, pp. 299–306.

[46] R. Goyal, G. Ferreira, C. Kästner, and J. Herbsleb, "Identifying unusual commits on github," *Journal of Software: Evolution and Process*, vol. 30, no. 1, p. e1893, 2018.

[47] A. Alali, H. Kagdi, and J. I. Maletic, "What's a typical commit? a characterization of open source software repositories," in *2008 16th IEEE international conference on program comprehension*. IEEE, 2008, pp. 182–191.

[48] C. Rosen, B. Grawi, and E. Shihab, "Commit guru: analytics and risk prediction of software commits," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 966–969.

[49] D. Gonzalez, T. Zimmermann, P. Godefroid, and M. Schäfer, "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, 2021, pp. 258–267.

[50] A. Gkortzis, D. Feitosa, and D. Spinellis, "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities," *Journal of Systems and Software*, vol. 172, p. 110653, 2021.

[51] R. Tóth, T. Bisztray, and L. Erdődi, "Llms in web development: Evaluating llm-generated php code unveiling vulnerabilities and limitations," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2024, pp. 425–437.

[52] R. Kallis, A. Di Sorbo, G. Canfora, and S. Panichella, "Ticket tagger: Machine learning driven issue classification," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2019, pp. 406–409.

[53] Y. Song and O. Chaparro, "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*, 2020, pp. 1551–1555.

[54] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019.

[55] F. Qi, X.-Y. Jing, X. Zhu, X. Xie, B. Xu, and S. Ying, "Software effort estimation based on open source projects: Case study of github," *Information and Software Technology*, vol. 92, pp. 145–157, 2017.

[56] "Radon: Python tool to compute code metrics," accessed: 2025-05-30. [Online]. Available: \url{https://radon.readthedocs.io/en/latest/}

[57] S. Liawatimena, H. L. H. S. Warnars, A. Trisetyarso, E. Abdurahman, B. Soewito, A. Wibowo, F. L. Gaol, and B. S. Abbas, "Django web framework software metrics measurement using radon and pylint," in *2018 Indonesian Association for Pattern Recognition International Conference (INAPR)*. IEEE, 2018, pp. 218–222.

[58] H. B. Hassan, Q. I. Sarhan, and Á. Beszédes, "Evaluating python static code analysis tools using fair principles," *IEEE Access*, vol. 12, pp. 173 647–173 659, 2024.

[59] X. Zhou, D.-M. Tran, T. Le-Cong, T. Zhang, I. C. Irsan, J. Sumarlin, B. Le, and D. Lo, "Comparison of static application security testing tools and large language models for repo-level vulnerability detection," *arXiv preprint arXiv:2407.16235*, 2024.

[60] K. Daigle and G. Staff. (2025) Survey: The ai wave continues to grow on software development teams. GitHub. Accessed: 2025-05-29. [Online]. Available: https://github.blog/news-insights/research/survey-ai-wave-grows/?utm_source=chatgpt.com

[61] I. Shani and G. Staff. (2023) Survey reveals ai's impact on the developer experience. GitHub. Accessed: 2025-05-29. [Online]. Available: https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/

[62] S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang, "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*, 2024, pp. 1–17.

[63] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "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, 2023, pp. 31–53.

[64] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," *arXiv preprint arXiv:2404.00971*, 2024.

[65] L. C. Ramírez, X. Limón, Á. J. Sánchez-García, and J. C. Pérez-Arriaga, "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, 2024, pp. 331–339.

[66] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo, "Refining chatgpt-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–26, 2024.

[67] L. Leite, C. Treude, and F. Figueira Filho, "Uedashboard: Awareness of unusual events in commit histories," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 978–981.