Sifting the Noise: A Comparative Study of LLM Agents in Vulnerability False Positive Filtering

YUNPENG XIONG, Monash University, Australia
TING ZHANG*, Monash University, Australia

Static Application Security Testing (SAST) tools are essential for identifying software vulnerabilities, but they often produce a high volume of False Positives (FPs), imposing a substantial manual triage burden on developers. Recent advances in Large Language Model (LLM) agents offer a promising direction by enabling iterative reasoning, tool use, and environment interaction to refine SAST alerts. However, the comparative effectiveness of different LLM-based agent architectures for FP filtering remains poorly understood.

In this paper, we present a comparative study of three state-of-the-art LLM-based agent frameworks, i.e., Aider, OpenHands, and SWE-agent, for vulnerability FP filtering. We evaluate these frameworks using the vulnerabilities from the OWASP Benchmark and real-world open-source Java projects. We further conduct a focused post-cutoff C/C++ study using the strongest configuration to test contamination-free generalization and isolate key agentic capabilities. The experimental results show that LLM-based agents can remove the majority of SAST noise, reducing an initial FP detection rate of over 92% on the OWASP Benchmark to as low as 6.3% in the best configuration. On a real-world Java dataset, the best configuration of LLM-based agents can achieve an FP identification rate of up to 93.3% involving CodeQL alerts. However, the benefits of agents are strongly backbone- and CWE-dependent: agentic frameworks significantly outperform vanilla prompting for stronger models such as Claude Sonnet 4 and GPT-5, but yield limited or inconsistent gains for weaker backbones. On the post-cutoff OSS-Fuzz dataset, SWE-agent with Claude Sonnet 4 identifies 95.5% of FPs while maintaining 95.5% precision, compared with a 36.4% FP identification rate for vanilla prompting. Moreover, aggressive FP reduction can come at the cost of suppressing true vulnerabilities, highlighting important trade-offs. Finally, we observe large disparities in computational cost across agent frameworks.

Overall, our study demonstrates that LLM-based agents are a powerful but non-uniform solution for SAST FP filtering, and that their practical deployment requires careful consideration of agent design, backbone model choice, vulnerability category, and operational cost.

CCS Concepts: • **Security and privacy** → **Software and application security**.

Additional Key Words and Phrases: Static Application Security Testing, False Positives, LLM Agents

ACM Reference Format:
Yunpeng Xiong and Ting Zhang. 2026. Sifting the Noise: A Comparative Study of LLM Agents in Vulnerability False Positive Filtering. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA009 (October 2026), 24 pages. https://doi.org/10.1145/3832100

1 Introduction

Recent years have seen a surge of software vulnerabilities, driven by the rapid growth of software systems and their increasing complexity [2, 3, 33]. The sheer number of reported vulnerabilities poses challenges to software security, as development teams struggle to detect, prioritize, and remediate security issues in a timely manner [13–15, 36].

*Ting Zhang is the corresponding author.

Authors' Contact Information: Yunpeng Xiong, Monash University, Melbourne, Australia, nemo.xiong@monash.edu; Ting Zhang, Monash University, Melbourne, Australia, ting.zhang@monash.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2994-970X/2026/10-ARTISSTA009
https://doi.org/10.1145/3832100

Problem and its importance. Static Application Security Testing (SAST) tools have been widely adopted in both industry and open-source communities to identify vulnerabilities early in the software development lifecycle [12, 37, 45, 48]. These tools analyze source code without execution and can be seamlessly integrated into IDEs and CI/CD pipelines [1, 45, 50]. However, SAST tools are well known for producing a high volume of FPs [5, 23, 31, 35, 59], i.e., warnings that resemble vulnerabilities syntactically but are not exploitable in practice. This noise increases the manual triage burden on developers and often leads to alert fatigue, reduced trust in automated security tools, or even blanket suppression of warnings [5, 23, 25].

In our empirical evaluation on the OWASP Benchmark [46], we observe that this problem is particularly severe: When aggregating findings across multiple mainstream SAST tools, the vast majority of non-vulnerable cases are flagged as suspicious by at least one scanner. Moreover, many FPs are shared across tools, indicating correlated semantic blind spots rather than isolated tool-specific errors. These observations suggest that simply combining or tuning existing SAST tools is insufficient to make their outputs actionable at scale.

Existing techniques. To mitigate the FP problem, prior research has explored learning-based approaches that classify or prioritize static warnings [23, 31]. For example, Transformer-based models have been used to distinguish FPs from true vulnerabilities using learned representations of code and warnings [33]. More recently, LLMs have been applied as semantic inspectors for static analysis results [16, 55]. The state-of-the-art approach, LLM4SA [55], demonstrates that prompting LLMs with SAST warnings and relevant code context can improve precision.

Despite these advances, existing LLM-based techniques primarily rely on *static prompting*, where the model is used as a one-shot classifier that produces a verdict in a single pass [16, 55]. Such approaches cannot actively explore codebases, gather additional evidence, or iteratively refine responses, in contrast to agentic frameworks that can interact with environments and tools [54, 56, 57]. However, these abilities are often essential for accurately assessing whether a reported vulnerability is real or an FP, especially in the presence of complex control flow or object-level semantics[35, 41, 55].

Our insights. LLM-based agents have brought considerable breakthroughs in recent years by extending vanilla LLMs with iterative reasoning, tool use, and environment interaction [54, 56, 57]. In software engineering tasks, agent-based approaches have shown strong effectiveness in areas such as automatic program repair [9, 58], agile effort estimation [11], and code generation [42]. Unlike static prompting, these agents operate in multi-step *perceive–reason–act* loops, enabling them to inspect code, navigate repositories, and validate intermediate assumptions [54, 56, 57].

These capabilities naturally align with how human auditors triage SAST warnings, suggesting that LLM-based agents may be well-suited for vulnerability FP filtering [5, 23, 55]. However, it remains unclear whether agentic frameworks consistently outperform vanilla LLM prompting in this task, how different agent designs compare with each other, and what trade-offs they introduce in terms of accuracy, safety, and computational cost [54–57].

To fill this gap, we present a comparative study of three state-of-the-art LLM agent frameworks, i.e., Aider [17], OpenHands [54], and SWE-agent [56], for filtering FP produced by SAST tools. We evaluate these frameworks with Claude Sonnet 4, DeepSeek Chat, and GPT-5 under a unified task definition: given a static alert and codebase access, determine whether the report corresponds to a real vulnerability or a false alarm. Our full cross-model and cross-agent evaluation covers the OWASP Benchmark [46] and real-world Java alerts from Vul4J [10], using Claude Sonnet 4, DeepSeek Chat, and GPT-5. For this comparative evaluation, we standardize prompts, constrain external tooling, and measure FP filtering performance, true-vulnerability suppression risk, and operational overhead. To address data-leakage and generalization concerns, we further conduct a

focused post-cutoff C/C++ study on OSS-Fuzz [22] using the strongest configuration (SWE-agent with Claude Sonnet 4) together with targeted ablations and stronger non-agentic baselines.

Our results show that agentic approaches substantially reduce SAST noise, but benefits depend on the backbone and weakness category. On OWASP Benchmark, the best configuration reduces the remaining FPR to single digits, with residual errors concentrated in hard weakness families such as cryptography- and policy-oriented CWEs. Across the three models, agentic Claude and GPT match or outperform vanilla zero-shot prompting, whereas DeepSeek performs best without agents. On real-world CodeQL findings, agents remain effective. Some frameworks are more aggressive at removing FPs, while others are more conservative to avoid suppressing true vulnerabilities. The focused OSS-Fuzz study further shows that the strongest configuration generalizes to post-cutoff C/C++ alerts, with gains driven mainly by cross-file navigation and multi-turn interaction rather than larger static context. Finally, computational overhead varies widely across frameworks, revealing a practical cost-effectiveness frontier for deploying agentic triage.

In summary, this work makes three contributions:

- ⋆ **Dimension.** This paper bridges recent advances in LLM-based autonomous agents [54, 56, 57] with a long-standing and practically critical software security problem, i.e., FP filtering in SAST [23, 31]. While prior work has explored LLMs as one-shot semantic classifiers [16, 55], our study is the first to systematically examine whether agentic reasoning, tool use, and environment interaction can meaningfully improve the actionability of SAST results at scale, both on benchmarks and real-world codebases.
- ⋆ **Comparative Study.** We conduct the first comprehensive comparative evaluation of three state-of-the-art LLM-based agent frameworks, i.e., Aider [17], OpenHands [54], and SWE-agent [56], for vulnerability FP filtering. The full cross-model and cross-agent comparison covers the OWASP Benchmark (v1.2) and real-world Java alerts from Vul4J [10], using Claude Sonnet 4, DeepSeek Chat, and GPT-5 as backbone models. We further conduct a focused post-cutoff C/C++ study on OSS-Fuzz [22] using the strongest configuration to evaluate contamination-free generalization, stronger non-agentic baselines, and capability ablations.
- ⋆ **Empirical Insights and Practical Guidelines.** Our results show that LLM-based agents can remove the majority of SAST FPs, reducing an initial FP detection rate of over 92% to 6.3% on OWASP Benchmark in the best configuration. On a sample (n=50) of the Vul4J dataset, the agents show a maximum of 93.3% FP identification rate when given alerts generated by CodeQL. However, we find that the benefit of shifting to an agentic framework is backbone- and CWE-dependent: agentic reasoning improves FP filtering for stronger models, but yields limited or even negative gains for others. Residual FPs are concentrated in policy- and cryptography-related CWEs, while injection-style vulnerabilities are filtered reliably. We further reveal trade-offs between aggressive FP removal, true vulnerability suppression, and operational cost, providing concrete guidance on when and how LLM-based agents should be deployed in practice.

2 Background and Motivation

2.1 Static Application Security Testing Tools

Static Application Security Testing (SAST) tools automatically scan application source code, bytecode, or binaries to identify security vulnerabilities such as injection flaws, XSS, and buffer overflows. These tools facilitate white-box analysis by integrating into IDEs and CI/CD pipelines to detect flaws early in the development lifecycle [35, 45]. By analyzing code without execution, SAST tools transform source text into intermediate representations to identify security violations [37, 55].

Query-based SAST Tools. Query-based SAST tools, exemplified by CodeQL [1], Joern [27], and Semgrep [8], parse a codebase into a relational database representing the program structure. This

database encapsulates information such as abstract syntax trees, control flow graphs, and program dependence graphs [35, 40, 41]. Vulnerability patterns are codified as domain-specific queries, such as CodeQL's object-oriented query language or Joern's graph traversals, which search the database for data-flow paths connecting sources to sinks.

Limitations and FPs. Despite their sophistication, the SAST tools are severely compromised by the trade-off between precision and scalability [35]. Industry leaders like Google, Microsoft, and Meta deploy these tools at scale to maintain code safety [12, 40, 48]. In practice, they often sacrifice precision (context sensitivity and path sensitivity) in favor of efficiency to maintain scalability over millions of lines of code [55].

Moreover, these tools report a high rate of FPs [40]. FPs from SAST tools cause "alert fatigue," resulting in developers losing trust in automated security tooling [25]. When faced with thousands of warnings, manual triage becomes prohibitively expensive. Consequently, developers frequently resort to suppression mechanisms to silence reports. However, evidence shows that 50.8% of suppressions are "useless": they do not actively suppress any warning due to code changes, yet they may even unintentionally mask future legitimate vulnerabilities [25].

2.2 Motivating Example: A Concrete False Positive

In this section, we first present one example `BenchmarkTest00171.java` (Listing 1) from the OWASP Benchmark to show when traditional SAST tools produce an FP on benign code.

```java
// ... (imports)
public void doPost(HttpServletRequest request, HttpServletResponse response) {
    // 1. Taint Source: 'param' receives user input
    String param = request.getHeader("BenchmarkTest00171");
    param = java.net.URLDecoder.decode(param, "UTF-8");

    // 2. Data Flow Obfuscation via HashMap
    java.util.HashMap map40534 = new java.util.HashMap();
    map40534.put("keyA-40534", "a_Value"); // Safe value
    map40534.put("keyB-40534", param);     // Tainted value

    // 3. Retrieval: 'bar' retrieves the SAFE value from keyA
    String bar = (String) map40534.get("keyA-40534");

    // 4. Sink: Command execution using 'bar'
    String cmd = "echo " + bar;
    Runtime.getRuntime().exec(cmd);

    // ...
}
```

Listing 1. An FP example from OWASP benchmark (BenchmarkTest00171.java)

In this scenario, the variable `param` is indeed tainted by user input. The code stores this tainted value in a HashMap under `keyB`, but subsequently retrieves a safe, hardcoded value (`"a_Value"`) from `keyA` to assign to the variable `bar`. Finally, `bar` is used in a system command execution.

- **CodeQL** reports that the "shell command line depends on a user-provided value," failing to distinguish the specific key-value dependencies within the map operations.
- **Semgrep** warns that "Untrusted input might be injected into a command," relying on coarse-grained taint tracking that observes data entering the map and data leaving the map, without precise sensitivity to the string keys.

These tools produce FPs as they lack precise object or path sensitivity to HashMaps, failing to track whether the retrieved value `bar` is semantically different from the tainted `param`. This example

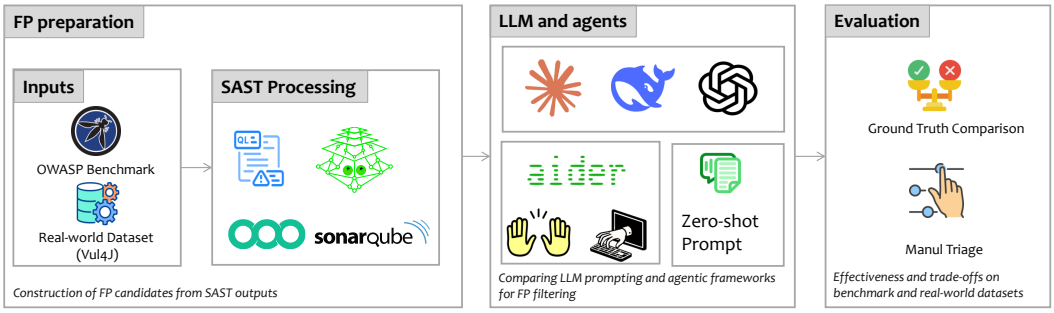


Fig. 1. Overview of our work

highlights a fundamental limitation of existing SAST tools: their inability to precisely reason about object-level semantics, motivating the need for more context-aware filtering approaches. A human auditor can readily deduce that the tainted data flow is severed. This suggests that an LLM-based agent with sufficient semantic reasoning capabilities may be able to reach the same conclusion. As shown by our experimental results (Section 4.1), existing SAST tools produce a high rate of FPs, indicating a strong need for more effective filtering mechanisms.

2.3 LLMs for Vulnerability Analysis: From Oracles to Agents

LLMs as Passive Semantic Oracles. LLMs have demonstrated effectiveness in function-level vulnerability detection by interpreting complex control flows and variable relationships [21, 38, 61]. In the context of static analysis, they often act as semantic oracles to post-process warnings [53]. For instance, LLM4SA demonstrated that feeding SAST findings and code context into an LLM can improve precision by distinguishing true vulnerabilities from FPs [55]. Other approaches have coupled LLM reasoning with program analysis to recover richer context, such as inferring taint specifications [39] or constructing vulnerability-focused slices via code property graphs [34].
Despite these advancements, most prior methods treat LLMs as one-shot classifiers operating on fixed, pre-extracted snippets. As probabilistic text generators, these vanilla LLM approaches lack the structural capability to verify their own outputs or actively navigate multi-file codebases, which limits their performance in rigorous security auditing.
The Emergence of Autonomous Agents. To address these limitations, recent research has shifted toward autonomous agents that operate in a *Perceive-Reason-Act-Observe* loop [57]. Unlike single-pass LLMs, agentic frameworks iteratively collect evidence and use external tools to refine hypotheses before reaching a decision [54, 56]. While agents have shown promising results in repository-level tasks such as program repair [9, 58], their efficacy in filtering FPs produced by SAST tools remains unexamined.
This gap motivates our study. We investigate the effectiveness of LLM-based agents for vulnerability FP triage through a comparative analysis of three representative frameworks. Our evaluation spans multiple backbone models, utilizing both a curated benchmark and real-world projects to provide actionable guidance on the role of agentic triage in software security.

3 Study Design

3.1 Research Questions

Figure 1 shows the overview of our study. Specifically, in this study, we aim to answer the following research questions (RQs):

- **RQ1:** How effective are different LLM-based agent frameworks in filtering FPs generated by SAST tools?
- **RQ2:** How effective are the LLM-based agent frameworks in identifying FPs in real-world Java scenarios?
- **RQ3:** What agentic capabilities drive FP filtering performance, and do the gains generalize to a contamination-free, C/C++ setting?
- **RQ4:** What are the key success drivers and recurring failure modes of LLM-based agents in FP identification?

3.2 Methodology for RQ1

Table 1. Data statistics of the OWASP Benchmark for Java, version 1.2

Vulnerability Area	Instances	Positive	Negative
CWE-78 (Command Injection)	251 (9.16%)	126 (4.60%)	125 (4.56%)
CWE-327 (Weak Cryptography)	246 (8.98%)	130 (4.74%)	116 (4.23%)
CWE-328 (Weak Hashing)	236 (8.61%)	129 (4.71%)	107 (3.91%)
CWE-90 (LDAP Injection)	59 (2.15%)	27 (0.99%)	32 (1.17%)
CWE-22 (Path Traversal)	268 (9.78%)	133 (4.85%)	135 (4.93%)
CWE-614 (Secure Cookie Flag)	67 (2.45%)	36 (1.31%)	31 (1.13%)
CWE-89 (SQL Injection)	504 (18.39%)	272 (9.93%)	232 (8.47%)
CWE-501 (Trust Boundary Violation)	126 (4.60%)	83 (3.03%)	43 (1.57%)
CWE-330 (Weak Randomness)	493 (17.99%)	218 (7.96%)	275 (10.04%)
CWE-643 (XPATH Injection)	35 (1.28%)	15 (0.55%)	20 (0.73%)
CWE-79 (Cross-Site Scripting / XSS)	455 (16.61%)	246 (8.98%)	209 (7.63%)
Total Instances	**2,740 (100.00%)**	**1415 (51.64%)**	**1325 (48.36%)**

Benchmark Dataset. To establish ground truth, we utilized the inherent labels provided by the OWASP Benchmark suite. We compared the aggregated SAST alerts against the benchmark's scorecard to categorize each alert as a True Positive (TP) or False Positive (FP). The OWASP Benchmark Project is a widely recognized Java test suite designed to verify the speed and accuracy of vulnerability detection tools [46], and has been used in numerous industries and for quantitative evaluation of SAST tools [24]. Prior studies on evaluating performance for static analysis, specifically on vulnerability detection, have widely used the OWASP Benchmark [4, 32, 37]. Each test instance within the benchmark is a simple Java EE servlet, while the vulnerabilities within the benchmark are implemented and injected into programs manually [37]. Each test instance is mapped to specific Common Weakness Enumeration (CWE) Weaknesses, making it easier to automate and evaluate against SAST tool outputs. Furthermore, the evaluation scripts for most SAST tools are provided in the source [44], thereby eliminating the possibility of tool misconfiguration. At the time of writing, the latest version of this benchmark is version 1.2. We use this version of the benchmark in our experiment. Table 1 presents the benchmark statistics.

Initial SAST Scanning. We selected four open-source SAST tools to generate an initial set of findings: CodeQL CLI v2.23.7 [19], Semgrep v1.145.0 [49], SonarQube Community v9.9.8 [51], and Joern v4.0.454 [29]. These tools represent different analysis paradigms, including semantic query engines and syntactic pattern matchers. This selection also aligns with previous works [37, 40].

We scanned the benchmark with each tool using default rule sets to gather its alerts. Each file contains only one vulnerable (or no) function within the benchmark framework. If a tool generates an alert on a file and the scope of the finding is within the framework code, we consider the file as a positive, flagged by the tool.

We aggregate the alerts by instance using a union-based approach: an instance is included in the candidate FP set if it is flagged as positive by at least one SAST tool. This aggregation reduces computation costs while maintaining a high recall of potential vulnerabilities for agent verification. Each tool adopts a slightly different running mechanism, specifically (1) **CodeQL.** We consider each "result" reported by `codeql database analyze` ran after `codeql database create` with default rules as a positive finding. (2) **Semgrep.** We consider each "result" reported by `semgrep scan` with default rules as a positive finding. (3) **SonarQube.** We consider each "issue" under "VULNERABILITY" type regardless of its severity, as a positive finding. We do not consider "security hotspot" as a positive finding. (4) **Joern.** We consider each line of "Result" reported by `joern-scan` as a positive finding.

Agentic Frameworks. We then provided these aggregated findings to three LLM-based agent frameworks: Aider v0.86.1 [17, 18], OpenHands Agent SDK v1.5.2 [43, 54], and SWE-agent commit 1d3cfb [52, 56]. They represent three interaction styles: pair-programming assistance (Aider), a developer-like workspace with editor/terminal/browser (OpenHands), and repository-level autonomous repair (SWE-agent). We utilized three state-of-the-art models as the backbone for these systems: Claude Sonnet 4 (`claude-sonnet-4-20250514`), DeepSeek Chat (`deepseek-chat`), and GPT-5 (`gpt-5-2025-08-07`). Each agent was provided with: (1) the specific warning messages generated by the SAST tools for the target file, (2) access to the test code, and (3) a prompt instructing the system to "verify if this static analysis finding represents a real vulnerability or an FP". To ensure fair comparison, we disable browser use and web browsing tools where possible for each agent. We made this design choice for two reasons. First, it keeps the evaluation focused on code reasoning rather than on whether an agent can find useful information on the web. Second, it protects benchmark integrity: unrestricted web access could allow an agent to retrieve benchmark scorecards, issue discussions, fix commits, or other external artifacts that directly reveal the expected ground truth. Such retrieval would change the task from code-based SAST triage to web-based answer lookup. To verify that this constraint did not materially bias our results, we conducted a paired ablation experiment using OpenHands with Claude Sonnet 4. We constructed the 50-case subset from the completed browser-disabled OWASP run: using a fixed Python random seed, we shuffled the completed case list, selected the first 50 cases, and re-ran exactly those instances with browser access enabled. Browser-disabled achieved an 86.0% FP identification rate (43/50), while browser-enabled achieved 82.0% (41/50), indicating that browser access provided no measurable advantage. We report metrics for each combination of model and agent.

Vanilla LLM. We employ zero-shot prompting on the LLMs to serve as a baseline. The prompt establishes a security-oriented reviewer persona and provides the model with necessary context, including the repository structure, the complete source code of the flagged file, and the raw findings from the SAST tool. To maintain consistency with the agentic setups, the prompt specifies a read-only environment and requires a triage decision based on the provided inputs. The complete prompt template and its variants are available in our replication package.

Metrics. We evaluate model performance by comparing the predicted labels against the benchmark ground truth. Results are categorized into True Positives (TP), False Positives (FP), True Negatives (TN), and False Negatives (FN) based on the benchmark ground truth. Specifically, a TP represents a correctly identified vulnerability, while an FP occurs when the tool reports an alert on an instance labeled as negative in the ground truth. Conversely, a TN indicates the correct identification of a non-vulnerable instance, and an FN occurs when the tool fails to detect a known vulnerability.

We also define False Positive Rate (FPR). FPR presents the percentage of non-vulnerable instances that were incorrectly flagged as positive by at least one SAST tool.

3.3 Methodology for RQ2

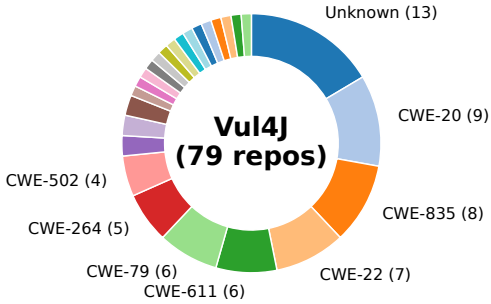


Fig. 2. Distribution of CWE identified in the Vul4J dataset.

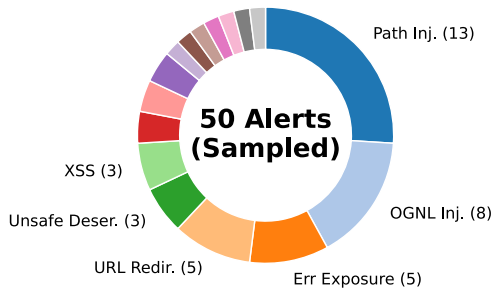


Fig. 3. Distribution of CodeQL rule IDs within the Vul4J sample used for RQ2· ($n = 50$).

Benchmark Dataset. We evaluate RQ2 using the Vul4J dataset [10], which consists of 79 reproducible Java vulnerabilities derived from Project KB [47]. Those vulnerabilities spanned across 51 distinct repositories, spanning a diverse set of real-world Java projects, covering enterprise web frameworks and middleware (e.g., Spring, Struts, CXF), security libraries (e.g., Shiro, ESAPI), data parsing/serialization components (e.g., fastjson, Jackson, XStream), and DevOps infrastructure such as Jenkins and its plugin ecosystem. The vulnerabilities cover 25 CWEs in total. A breakdown of the CWE categories is illustrated in Figure 2. To notice, the category of *Not Mapping* refers to when the authors of the Vul4J dataset failed to retrieve the CWE information for the vulnerabilities.
The dataset is provided as 79 branches in the original dataset hosted on GitHub. Each branch represents a specific vulnerable version of a certain repository in question, which is a data point in the dataset. For evaluation, we checked out each of them and manually copied them as 79 distinct folders. These 79 branches have 6,786,406 source lines of code in total, where the mean and median for each branch are 85,903 and 42,136, respectively. We scanned these 79 cases using CodeQL, resulting in a total of 3,426 alerts, where the mean and median are 43 and 16.
Manual Triage. Due to the large volume of the alerts, we sampled 50 alerts from these for manual triage. We performed the sampling using Python's *random* library with a fixed seed to ensure reproducibility. This sampled set represents 26 projects and covers 16 unique CodeQL rules, accounting for 37% of the total rules observed in the original scan. A breakdown of the CodeQL rules observed in these samples is illustrated in Figure 3.
To establish ground truth labels for these alerts, we followed a two-step process. First, we automatically matched CodeQL alerts against the human-written patches provided in Vul4J. Following previous work [37], we employed method-level matching: any alert falling within a patched method with a matching CWE type was automatically marked as a TP. In our sampled 50 alerts, 2 of them were automatically labeled as TP with this strategy. For the remaining 48 unlabeled alerts, the first author served as the annotator to manually inspect each sample. The annotator has 6 years of experience in the security field and programming. The annotation process can be described as follows: The annotator iterates through sampled alerts. For each alert, the annotator: (1) Browse the project associates with this alert and locate the file; (2) Study the actual CodeQL rule and description and code flow to have an understanding of what this alert is about; (3) Perform analysis including but not limited to: data flow analysis, checking for known insecure patterns, searching for issues mentioning this file, refer to other static analyzers; (4) Consult to other experts if the annotator does not have confidence in the nature of this alert after the annotator perform the above

steps. Otherwise, continue; (5) Label it as a true vulnerability, i.e., TP, if it accurately represented its description and category; otherwise, we labeled it as an FP. The whole process took about 20 man-hours. Finally, among the 50 alerts, 31 are labeled as FPs, and 19 are labeled as TPs, yielding an initial FP identification precision of 62%.

We then provided these labeled alerts to three agent frameworks: Aider, OpenHands, and SWE-agent. Similarly, these frameworks utilized Claude Sonnet 4, DeepSeek Chat, and GPT-5, as their backbone models. Each agent received: (1) the CodeQL warning message, (2) access to the full repository context for cross-file reasoning, and (3) a prompt instructing the agent to "verify if this static analysis finding represents a real vulnerability or an FP", following the procedure in RQ1 3.2. Finally, we compare the results produced by these agents to their backbone model, i.e., the vanilla LLM, with zero-shot prompting, the same as in RQ1 (Section 3.2). We conduct qualitative analysis in Section 5.

Metrics. Unlike RQ1, which reports residual FPR from a tool-output perspective, RQ2 evaluates alert-level triage as a binary classification task. We treat SAST FPs as the positive class because the operational goal is to identify alerts that can be safely filtered. To maintain terminology consistency and address the safety implications of agentic filtering, we define the outcome categories as follows: (1) Identified FP: The agent successfully identifies an SAST FP. (2) Missed FP: The agent fails to identify an SAST FP. (3) Mis-identified FP: A safety violation where the agent incorrectly classifies a true vulnerability as an FP. (4) Correct Retention: The agent successfully identifies and preserves a true vulnerability. Under this framework, recall is the FP identification rate, precision is the probability that an alert classified as an FP is indeed an FP, and F1 summarizes the precision–recall trade-off.

3.4 Methodology for RQ3

Unlike RQ2's full Java comparison, RQ3 is a focused mechanism and generalization study. We use the strongest RQ1 configuration, Claude Sonnet 4 with SWE-agent, and compare it with targeted ablations and non-agentic baselines on OSS-Fuzz. This tests whether agentic gains persist in a contamination-free C/C++ setting and which capabilities explain them.

C/C++ Evaluation. To address concerns about possible data leakage and to extend our evaluation beyond Java, we built a post-cutoff C/C++ alert pool from Google's OSS-Fuzz [22] continuous fuzzing service. Starting from the most recent records in the public OSV export for OSS-Fuzz, we scanned backward through OSS-Fuzz records until we identified recent C/C++ projects whose fixing commits postdated the known training cutoff of Claude Sonnet 4 by 10–12 months and whose vulnerable snapshots could be reconstructed. For each retained OSV record, we used the OSS-Fuzz/OSV metadata to obtain the fixing commit, derived the corresponding vulnerable commit as the single parent of that fixing commit, and reconstructed the corresponding vulnerable repository snapshot. The final sampled alerts cover five projects (vulnerable fixing commit time in bracket): mpv (January 26, 2026), libhevc (February 14, 2026), libical (February 21, 2026), gpsd (March 11, 2026), and quickjs (March 21–23, 2026). We scanned the vulnerable versions of these projects using CodeQL with default C/C++ rules, generating a pool of 804 eligible alerts after filtering out test/generated files and non-security/non-CWE-tagged findings. From this pool, we sampled 50 alerts using a fixed random seed (seed=42), ensuring reproducibility. The sample covers all five projects with the following distribution: mpv (11 alerts), gpsd (16 alerts), libical (9 alerts), libhevc (13 alerts), and quickjs (1 alert). The sampled alerts cover 11 CodeQL C/C++ rule IDs. According to CodeQL's full CWE coverage documentation [20], these rules map to 25 security-relevant CWE IDs. These CWE IDs span high-level weakness types including path traversal and path manipulation (CWE-22/23/36/73/610/642/706), numeric overflow/conversion/calculation (CWE-190/197/681/682/704), access-control and permission weaknesses (CWE-284/285/668/693/732), initialization and nullness

issues (CWE-457/476/665), suspicious comments (CWE-546), unused/dead code (CWE-561/563), and dangerous function use (CWE-676). To establish ground truth, two independent raters, each with 6+ years of programming experience, labeled all 50 alerts. The raters achieved 86.0% raw agreement (43/50 cases), with Cohen's $\kappa = 0.725$, indicating substantial agreement. The 7 disagreements were resolved through discussion, yielding a final ground truth of 22 FPs and 28 TPs.

Experimental Configurations. On this OSS-Fuzz dataset, we evaluated three types of configurations to isolate the sources of agentic gains: **(1) Full agent configurations:** We evaluate the most capable configuration in RQ1 (SWE-AGENT with Claude Sonnet 4). **(2) Ablation variants:** We systematically removed four capabilities from the full configuration: multi-turn interaction, cross-file navigation, tool-based verification (e.g., arithmetic checks), and configuration-file access. These dimensions were motivated by prior work where ReAct emphasizes interleaving reasoning and action, while SWE-AGENT emphasizes repository navigation and tool-enabled interaction and by our own trajectory analysis, which identified configuration validation as a recurring success pattern. **(3) Stronger baselines:** We evaluated three additional baselines beyond vanilla prompting: (a) LLM4SA-style [55] prompting, where we reuse the LLM4SA task instruction and output-label format, fill its *Bug Report* field with the CodeQL alert JSON, and fill its *Code Snippet* field with the complete reported source file, but do not reproduce LLM4SA's separate program-dependence context-extraction pipeline; (b) context-augmented vanilla LLM, which provides cross-file context via heuristic-based searching including call-graph neighbors, direct includes, CodeQL generated code-flow and related locations; and (c) oracle-context vanilla LLM, which receives the exact same files that our full configuration accessed during its successful runs. The oracle-context baseline allows us to isolate the contribution of iterative reasoning from context volume.

Metrics. RQ3 uses the same alert-level FP-filtering metrics as RQ2, with SAST FPs as the positive class. We report accuracy, precision, recall, and F1; recall corresponds to the FP identification rate.

3.5 Methodology for RQ4

To understand the performance gap between agentic frameworks and vanilla prompting, we conduct a manual inspection of execution trajectories. Our analysis focuses on two distinct sets of cases:

Success Analysis: We examine 256 trajectories where the standalone LLM incorrectly predicts a TP but SWE-AGENT (with Claude Sonnet 4) correctly identifies a FP. These cases cover 11 OWASP categories, primarily CWE-330 (126 cases), CWE-78 (35 cases), and CWE-79 (20 cases).

Failure Analysis: We analyze 103 failed trajectories where the agent misses an FP. This set includes 59 cases where both OPENHANDS and SWE-AGENT failed, plus targeted inspection of the dominant failure categories: CWE-327 (47 cases) and CWE-614 (10 cases). For each trajectory, we record the number of steps, the files accessed, and the specific tools utilized. We then categorize the underlying reasoning patterns into success patterns (P1–P4) and failure modes (F1–F3) based on the agent's interaction with the codebase and configuration files.

4 Results

4.1 RQ1: Filtering Performance of LLM-Based Agents

4.1.1 FPs Produced by SAST Tools. Table 2 summarizes initial SAST performance; bold and underline mark the best and second-best results. Figure 4 shows each tool's alert distribution. CodeQL achieves the highest F1 (74.4%) and precision (60.3%), but still produces 904 FPs, covering 33.0% of benchmark cases and 68.2% of ground-truth non-vulnerable cases. Semgrep and SonarQube have lower precision and recall than CodeQL, with recall rates of 90.4% and 95.6%. SonarQube produces the most FPs (1,254; 45.8% of the benchmark) and flags 94.6% of non-vulnerable cases as positive. Joern is conservative, producing only 96 FPs but missing most vulnerabilities, with 8.2%

Table 2. Initial Performance of SAST Tools Across All Instances. The Union represents the candidate pool provided to LLM agents for filtering.

Tool	Accuracy	Precision	Recall	F1	FPR
CodeQL	**65.5%**	**60.3%**	**97.0%**	**74.4%**	68.2%
Joern	49.1%	54.7%	8.2%	14.3%	**7.2%**
Semgrep	58.9%	56.3%	90.4%	69.4%	74.8%
SonarQube	52.0%	51.9%	95.6%	67.3%	94.6%

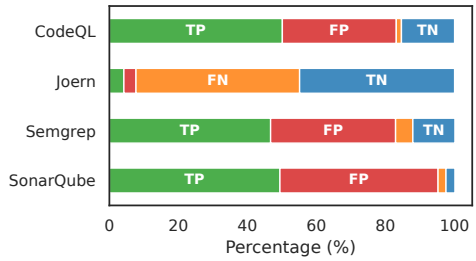
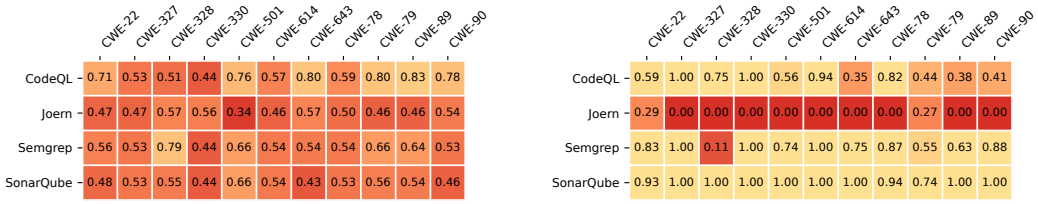


Fig. 4. Distribution of SAST alerts (Overall Scope).

recall. This may stem from Joern's limited default Java query set, which contains 6 rule files and 7 vulnerable-code detection functions [26, 28].



(a) Accuracy across different CWE categories.

(b) FPR across different CWE categories.

Fig. 5. Performance comparison of four SAST tools across various CWEs. (a) Heatmap illustrating accuracy. (b) Heatmap illustrating FPR, where values of 1.00 indicate that tools flag all non-vulnerable instances in those categories.

Figure 5a shows that the detection capabilities vary across tools and vulnerability types. CodeQL consistently demonstrates robust detection performance, achieving the highest accuracy in the majority of categories, specifically peaking at 0.83 for CWE-89 (SQL Injection) and 0.80 for CWE-78 (OS Command Injection) and CWE-643 (XPATH Injection). Semgrep shows competitive performance in specific categories, notably outperforming other tools in CWE-328 (Weak Hashing) with an accuracy of 0.79. Conversely, Joern and SonarQube exhibit moderate to lower accuracy across the board, with Joern dropping as low as 0.34 for CWE-501 (Trust Boundary Violation).

Figure 5b highlights a high volume of FPs generated by traditional SAST tools. In our initial scan, SonarQube and Semgrep exhibit excessively high FPRs. SonarQube, in particular, reaches an FPR of 1.00 across nearly all evaluated CWEs, indicating that for these categories, the tool flags almost every non-vulnerable test instances as a vulnerability. Similarly, Semgrep shows an FPR of 1.00 for multiple categories, including CWE-327 and CWE-330. While CodeQL achieves the highest accuracy, FPR reveals that this comes at the cost of precision. Its FPR reaches 1.00 for CWE-327 and CWE-330, and remains above 0.80 for injection flaws like CWE-78 (0.82). In comparison, Joern maintains an FPR of 0.00 for the majority of CWEs. While this can indicate high precision, its lower accuracy may suggest a conservative analysis strategy that likely suffers from a high false negative rate, missing actual vulnerabilities to avoid false alarms.

To understand whether this noise is tool-specific or correlated, we analyzed the overlap of FP instances across scanners. As shown in Figure 6, FPs are largely shared rather than unique. Only a small fraction of noise is exclusive to a single tool, e.g., 14 instances unique to CodeQL and 11 unique to Semgrep, while SonarQube acts as a near-superset of the FPs reported by other scanners. More importantly, shared-error regions remain large: 36 FP instances are flagged by all four tools, and CodeQL, Semgrep, and SonarQube jointly flag a dominant block of 710 instances.

Table 3. Overall FP reduction performance on the OWASP benchmark (FPR, lower is better).

Model	Agent	FPR (compared to SAST)
Claude Sonnet 4	Aider	14.3% (↓ 84.1%)
	OpenHands	14.9% (↓ 83.5%)
	SWE-agent	6.3% (↓ 92.1%)
	Vanilla LLM	23.0% (↓ 75.3%)
DeepSeek Chat	Aider	13.2% (↓ 85.1%)
	OpenHands	15.8% (↓ 82.6%)
	SWE-agent	13.1% (↓ 85.2%)
	Vanilla LLM	11.2% (↓ 87.1%)
GPT-5	Aider	20.3% (↓ 78.0%)
	OpenHands	16.3% (↓ 82.0%)
	SWE-agent	14.1% (↓ 84.2%)
	Vanilla LLM	20.4% (↓ 78.0%)

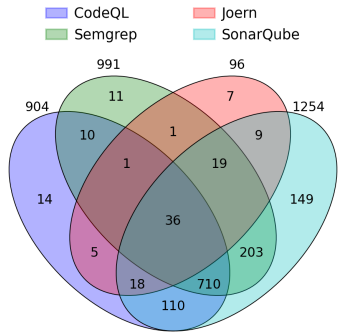


Fig. 6. Distinct FP cases detected by each tool.

4.1.2 Effectiveness of Agentic FP Filtering. We evaluate LLM agents on the union of SAST alerts, where at least one tool flags 1,303 of 1,325 non-vulnerable instances, yielding a 98.3% baseline FP rate. Table 3 reports the remaining FPR after agent verification. Among all configurations, SWE-agent paired with Claude Sonnet 4 achieves the highest filtering performance. This setup reduces the remaining FPR to 6.3%, representing a 92.1% reduction from the original rate. This performance exceeds that of OpenHands at 14.9% and Aider at 14.3%. When using GPT-5, SWE-agent remains the top-performing agent with a 14.1% FPR, though the performance gap among frameworks decreases. For DeepSeek Chat, agent performance ranges between 13.1% and 15.8%, suggesting that increased complexity in agentic loops does not consistently result in improved filtering. Overall gains are strongly backbone-dependent. For Claude Sonnet 4, SWE-agent reduces the remaining FPR from 23.0% with vanilla prompting to 6.3%, demonstrating a clear advantage for iterative logic. For GPT-5, SWE-agent also improves upon the vanilla baseline, reducing the FPR from 20.4% to 14.1%. In contrast, DeepSeek Chat shows no consistent improvement from an agentic workflow; the vanilla baseline already achieves a low remaining FPR of 11.2%, while its agentic counterparts yield comparable or higher residual error rates.

4.1.3 Categorical Analysis across CWEs. A CWE-level analysis indicates that agentic workflows do not provide uniform benefits across different vulnerability types. Figure 7 illustrates the performance difference between the three agents and the vanilla LLM baseline, measured in percentage points (pp) of FP filtering success.

Under Claude Sonnet 4, all three frameworks improve filtering for high-volume injection categories and CWE-330, though the degree of improvement varies. SWE-agent is the most consistently beneficial, improving performance in 10 out of 11 CWEs. Its largest improvements occur in CWE-330, with a 44.4,pp increase from 51.6% to 96.0%, and CWE-78, which increases by 28.5,pp from 69.9% to 98.4%. OpenHands also shows improvements in CWE-78 and CWE-330, yet it exhibits a regression in CWE-614, where performance drops by 41.9,pp from 74.2% to 32.3%, and in CWE-327, which decreases by 26.0,pp from 55.2% to 29.2%. Aider remains more stable on policy and cryptographic categories; for instance, its CWE-614 performance improves to 80.6%, although its gains on high-noise blocks are smaller than those of SWE-agent.

For DeepSeek Chat, the effects of an agentic workflow are complex and polarized. OpenHands and SWE-agent improve injection-heavy categories, particularly CWE-78, with gains of 39.9,pp and 42.3,pp respectively. However, these same agents demonstrate a sharp degradation in policy and

cryptographic categories. Performance in CWE-614 drops by 67.7,pp and 80.6,pp, while CWE-327 success decreases by 69.0,pp and 63.8,pp. Conversely, Aider preserves perfect performance in CWE-614 at 100% and increases CWE-327 success to 100%, but suffers a 12.2,pp regression in CWE-78.

GPT-5 exhibits almost no regressions when an agentic workflow is applied. Both OpenHands and SWE-agent raise CWE-327 filtering success from 1.7% to over 22%, and improve CWE-78 performance by 10.5,pp and 20.3,pp respectively. For categories with high baseline success, such as CWE-79, CWE-89, CWE-614, and CWE-643, all frameworks operate near the performance ceiling, resulting in smaller marginal gains.

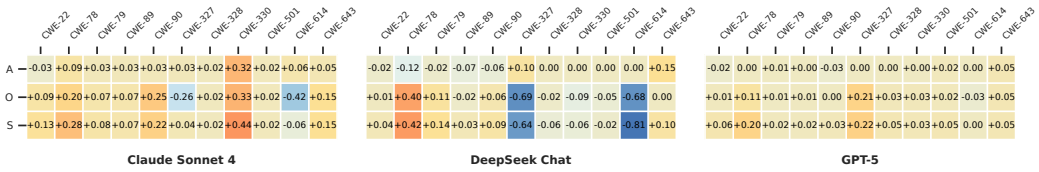


Fig. 7. CWE-level change in FP filtering success over vanilla prompting. Values are percentage-point changes; red indicates gains and blue for regressions. (A=Aider, O=OpenHands, S=SWE-agent.)

Answer to RQ1. LLM-based agents reduce the OWASP residual FPR from 98.3% to as low as 6.3%, but the gain depends on backbone model and CWE. Claude Sonnet 4 and GPT-5 benefit from agentic behaviour, while DeepSeek Chat shows no net gain; residual FPs concentrate in policy-based categories like weak cryptography.

4.2 RQ2: Performance of LLM Agents in Real-World Java Scenarios

Table 4. Performance in selected Vul4J projects (%). Identification metrics refer to the agent's ability to classify SAST FPs correctly while avoiding misidentification of true vulnerabilities.

Model	Agent	Iden. FP	Corr. Ret.	Mis-id. FP	Missed FP	Acc	Prec	Rec	F1
Claude Sonnet 4	Aider	46.9	30.6	8.2	14.3	**77.6**	85.2	76.7	80.7
	OpenHands	42.0	34.0	4.0	20.0	76.0	91.3	67.7	77.8
	SWE-agent	38.3	29.8	10.6	21.3	68.1	78.3	64.3	70.6
	Vanilla LLM	38.0	28.0	10.0	24.0	66.0	79.2	61.3	69.1
DeepSeek Chat	Aider	34.0	34.0	4.0	28.0	68.0	89.5	54.8	68.0
	OpenHands	24.0	38.0	0.0	38.0	62.0	**100.0**	38.7	55.8
	SWE-agent	31.2	35.4	4.2	29.2	66.7	88.2	51.7	65.2
	Vanilla LLM	42.0	16.0	22.0	20.0	58.0	65.6	67.7	66.7
GPT-5	Aider	61.0	14.6	19.5	4.9	75.6	75.8	92.6	**83.3**
	OpenHands	58.3	12.5	25.0	4.2	70.8	70.0	**93.3**	80.0
	SWE-agent	53.2	21.3	19.1	6.4	74.5	73.5	89.3	80.6
	Vanilla LLM	57.1	14.3	19.0	9.5	71.4	75.0	85.7	80.0

Table 4 shows that FP mitigation performance remains highly backbone-dependent. For Claude Sonnet 4, agentic workflows yield a clear advantage over vanilla prompting. Aider achieves the most balanced performance with a recall of 76.7% and precision of 85.2%. OpenHands adopts a more conservative strategy, yielding the highest precision at 91.3% but a lower rate of identified FPs at 67.7%. These results indicate that for Claude, agentic workflows increase the efficacy of noise

reduction, though different framework designs occupy different positions on the spectrum between safety and aggressiveness.

For DeepSeek Chat, the trade-off between noise removal and vulnerability retention is more pronounced. The vanilla LLM baseline achieves a recall of 67.7% but maintains a low precision of 65.6%, indicating a higher frequency of mis-identified FPs. Conversely, OpenHands reaches 100% precision but only identifies 38.7% of the FPs, leaving noise unfiltered. Aider provides a balanced alternative with 89.5% precision and 54.8% recall. These outcomes highlight that a high rate of identified FPs is insufficient if it is achieved by incorrectly discarding actual security flaws.

For GPT-5, the recall is consistently high across all configurations and ranges from 85.7% to 93.3%, which narrows the gap between agentic workflows and vanilla prompting. Aider yields the highest F1-score of 83.3% with a 92.6% recall and 75.8% precision. OpenHands marginally increases the recall to 93.3% but at a lower precision of 70.0%. This suggests that for highly capable backbone models, additional agentic scaffolding may yield diminishing returns in noise mitigation, and the primary differentiator shifts to how effectively the framework avoids mis-identifying true vulnerabilities.

Answer to RQ2. On real-world Java CodeQL alerts, agents can identify up to 93.3% of FPs, but the precision–recall trade-off remains backbone-dependent. Claude benefits most from agentic frameworks, GPT-5 shows smaller gains over vanilla prompting, and DeepSeek trades recall for precision.

4.3 RQ3: Agentic Capabilities and Post-Cutoff Generalization

Table 5 presents the complete results for OSS-Fuzz, including the full SWE-agent, four ablation variants, and three additional baselines, all using Claude Sonnet 4 as the backbone model. On this contamination-free C/C++ dataset, the full SWE-agent identifies 95.5% of FPs while maintaining 95.5% precision, compared with a 36.4% FP identification rate for vanilla prompting.

The ablation results reveal a capability hierarchy rather than a uniform benefit from all agentic behaviour. **Cross-file navigation** is the most important capability: removing it drops accuracy from 96.0% to 44.0% and F1 from 95.5% to 62.5%. Among the 26 full-correct cases missed by this ablation, all become `UNKNOWN`, showing that the agent can no longer collect enough evidence for a grounded verdict. For example: in `mpv@eadba44`, the integer-overflow alert at `audio/filter/af_scaletempo2_internals.c` requires reading `audio/filter/af_scaletempo2_internals.h` to confirm operand types, while the path injection alert at `common/msg.c` requires following `root->stats_path` to `options/options.c` and `options/path.c`. **Multi-turn interaction** is the second key capability: removing it reduces F1 to 56.3%. 12 of the 14 full-correct cases lost by this ablation are FPs for `cpp/unused-static-variable`, where the one-shot variant sees only a declaration window but the full agent searches for later uses. For example, the full agent follows `registry_listener` in `mpv@eadba44/player/clipboard/clipboard-wayland.c` from its declaration at line 272 to a use at line 336, and finds later reads of `g_ai2_ihevc_trans_32_13_815` in `libhevc@8cbcc58/common/arm/ihevc_resi_trans_neon_32x32.c`. **Tool-based verification** has a smaller, localized effect, reducing F1 to 88.9%. The affected cases concentrate in integer-multiplication alerts, where the agent must reconcile typedefs, `sizeof`, and C integer-promotion semantics. **Configuration-file access** has negligible impact: removing it loses no full-correct case, suggesting that these OSS-Fuzz C/C++ alerts depend mainly on source, header, macro, and type evidence rather than external configuration.

The baseline comparison further isolates the source of the agentic gain. The LLM4SA-style prompt-template baseline improves over vanilla prompting, achieving 77.3% FP recall and 70.8% F1, but still falls short of the full agent. The context-augmented baseline achieves 54.5% FP recall and 66.7% F1,

showing that additional static context alone is insufficient. Most importantly, the oracle-context baseline achieves only 36.4% FP recall and 51.6% F1, nearly identical to vanilla prompting despite receiving the same files accessed by SWE-AGENT. This indicates that the advantage comes from iterative evidence gathering and reasoning, not merely from context volume.

Answer to RQ3. On post-cutoff OSS-Fuzz C/C++ alerts, SWE-AGENT with Claude Sonnet 4 identifies 95.5% of FPs with 95.5% precision, versus 36.4% FP identification for vanilla prompting. Ablations show that cross-file navigation and multi-turn interaction drive the gain, while context-only baselines remain below the full agent.

Table 5. OSS-Fuzz C/C++ evaluation results. Metrics are percentages.

Configuration	Iden. FP	Corr. Ret.	Mis-id. FP	Missed FP	Acc	Prec	Rec	F1
SWE-agent (full)	42.0	54.0	2.0	2.0	96.0	95.5	**95.5**	**95.5**
w/o multi-turn	18.0	52.0	2.0	26.0	70.0	90.0	40.9	56.3
w/o cross-file	20.0	24.0	0.0	24.0	44.0	**100.0**	45.5	62.5
w/o tool-based verification	40.0	48.0	6.0	4.0	88.0	87.0	90.9	88.9
w/o config read	42.0	56.0	0.0	2.0	**98.0**	**100.0**	**95.5**	**97.7**
Vanilla LLM	16.0	52.0	2.0	28.0	68.0	88.9	36.4	51.6
Vanilla LLM + LLM4SA	34.0	38.0	18.0	10.0	72.0	65.4	77.3	70.8
Context-augmented Vanilla LLM	24.0	52.0	4.0	20.0	76.0	85.7	54.5	66.7
Vanilla LLM + Oracle Context	16.0	54.0	2.0	28.0	70.0	88.9	36.4	51.6

4.4 RQ4: Qualitative Analysis of Success and Failure Modes

Table 6. Frequency of tool invocation across 256 analysis trajectories.

Tool	# Cases	% Cases	Total Calls
`view`	256	100.0	1,488
`find`	90	35.2	119
`grep`	82	32.0	118
`python3 -c`	39	15.2	45
`javac/java`	4	1.6	4

Table 7. Distribution of failure-mode (FM) patterns by CWE. A trajectory represents one agent run on one case; counts may exceed #Traj. due to multiple patterns per trajectory.

CWE	# Traj.	FM1	FM2	FM3
CWE-327	85	85	35	0
CWE-614	24	15	18	0
CWE-79	8	0	0	8

4.4.1 Capability Analysis: Tool Use and Reasoning. Across the 256 successful trajectories, SWE-AGENT runs a median of 11 steps and views a median of 2 distinct files (source, config). Notably, in 51.2% of cases, the agent reads at least one non-target file beyond the primary benchmark test file, such as helper classes or configuration files. In 25.0% of cases, the final justification explicitly cites evidence found within these non-target files. Table 6 shows a breakdown of how SWE-AGENT utilizes the tools. We categorize the primary reasons for the agent's success into four patterns (P1–P4), as detailed below.

P1: Cross-file Semantic Resolution. Many OWASP Benchmark FPs depend on semantics hidden in helper classes, e.g., wrapper sources, factory-selected implementations. Vanilla prompting an LLM without full context often overapproximates dataflow taint. For example, in `BenchmarkTest00866`, the vanilla LLM labels this instance as TP with high confidence, reasoning that a request parameter flows through a transformation and is concatenated into a file path (`TESTFILES_DIR + bar`), thus enabling a `../` traversal. However, the LLM's evidence is confined to the benchmark test file and

assumes that the wrapper source is user-controlled. In contrast, SWE-AGENT uses tool access to retrieve the missing non-local semantics. It (1) opens `BenchmarkTest00866.java` to identify the dataflow, (2) inspects `SeparateClassRequest.java` and finds that `getTheValue()` is explicitly marked as a *safe source* returning the constant string `"bar"` (hence the "param" is not attacker-controlled), (3) inspects the factory-selected `Thing` implementation (`ThingFactory/Thing2` and `thing.properties`) to confirm the transformation does not reintroduce taint, and (4) checks `Utils.TESTFILES_DIR` to confirm a fixed safe base directory. With these cross-file facts, the agent concludes that the effective filename is constant (base dir + `"bar"`), making traversal impossible, and correctly outputs `FP`.

P2: Constant Folding for Control-flow Disambiguation. A common FP construction in the benchmark is a branch that looks data-dependent but is in fact constant. Vanilla prompting frequently treats the unsafe branch as feasible, concluding TP. SWE-AGENT often resolves this via a lightweight calculator call (e.g., `python3 -c`) and turns the control-flow argument into a verifiable numeric fact. An example for this is `BenchmarkTest00105`, where the vanilla LLM flags SQL injection because `bar` *could* be assigned from a cookie-derived `param`, which actually is a false branch in an unfold constant expression; SWE-AGENT substitute the constant variable and computes that the constant`(7*18)+106 = 232 > 200` is always true, hence `bar` is always the constant `This_should_always_happen` and the SQL query is not attacker-controlled.

P3: Configuration Validation. For crypto and factory-driven behavior, the apparent default value in code can be misleading. Vanilla prompting often assumes the fallback is vulnerable (e.g., default ECB mode), yielding a TP prediction. SWE-AGENT resolves this by locating and reading the relevant `.properties` files, grounding the verdict in actual configuration. This is exemplified by `BenchmarkTest01022`, where the vanilla LLM labels this instance as `TP` and cites two surface cues: (1) a stack-trace print to the HTTP response, and (2) a configuration call of the form `getProperty("cryptoAlg2", "AES/ECB/PKCS5Padding")`, which it interprets as using insecure ECB.

SWE-AGENT avoids this failure via configuration grounding and local semantic checks. First, it inspects the relevant block and notes that the data being encrypted is not attacker-controlled due to a constant branch in `doSomething` (the condition is always true, forcing a constant string). Second, rather than assuming the fallback cipher mode is used, it locates and opens src/main/resources/benchmark.properties and verifies that `cryptoAlg2` is set to `AES/CCM/NoPadding` (a non-ECB mode) in the benchmark configuration. Therefore, the "weak crypto" alert is not supported under the benchmark execution context, and the agent outputs FP.

P4: Direct Verification of Complex Semantics. In a small number of cases (4/256, 1.6%), the agent attempts an even stronger form of grounding: it generates a minimal Java snippet that reproduces the suspicious logic and tries to compile/run it (e.g., list index manipulation). For example, in `BenchmarkTest00265`, the vanilla LLM treats a list-based transformation as propagating user input into a file path. SWE-AGENT creates a miniature program (`test_logic.java`) to validate the effect of `add/remove/get` on the selected element. In our sandbox, `javac` is unavailable, so execution does not succeed; nevertheless, this trajectory suggests that enabling lightweight execution-based checks when feasible could further improve on semantics analysis that are easy to misread but cheap to validate.

4.4.2 Failure Analysis. The inspection of 103 failed trajectories reveals three primary failure modes (FMs). Table 7 presents the distribution of these FMs.

FM1. Incorrect CWE Attribution. Agents often validate a security issue other than the benchmark's target CWE. In all CWE-327 cases, the source code contains `printStackTrace(response.`

`getWriter()`) within a cryptographic exception handler. This pattern may reveal sensitive application information to users. The trajectories indicate that agents identify this as a true vulnerability related to the cryptographic context, even though it does not constitute a CWE-327 violation. Furthermore, 7 out of 10 CWE-614 cases are marked as TP because the agents conclude that the cookie value is user-controlled and susceptible to CRLF injection (CWE-113) due to a lack of sanitization, rather than focusing on the missing Secure flag.

FM2. Overly Conservative Threat Modeling. Agents frequently adopt a worst-case threat model, which prioritizes recall but reduces precision in FP triage. Even when agents focus on the SAST alerts, they often conflate hardening guideline violations with exploitable vulnerabilities. In 4 out of 10 CWE-614 cases, agents report a TP solely because the `SameSite` attribute is absent. The agents reason that this absence could permit a CSRF attack despite hardcoded test inputs, arguing that the code pattern represents a production security weakness without attempting to construct a feasible payload. Additionally, in 12 out of 47 CWE-327 cases, agents identify weak cryptographic algorithms such as `"AES/ECB/PKCS5Padding"` that are unreachable in the execution flow. The reasoning follows a static-like pattern: the existence of the weak algorithm justifies reporting a vulnerability.

FM3. Surface-Level Pattern Matching. Agents may rely on surface cues, such as specific API names and suspicious sinks, and terminate analysis prematurely without deep semantic validation. In all 4 CWE-79 cases, both agents conclude a TP based on the presence of `format()` and output APIs. This combination is often associated with format-string or injection vulnerabilities. However, the data returned in these cases is already HTML-escaped. The remaining concern involves software robustness rather than a functional XSS primitive.

Answer to RQ4: Agentic workflows improve FP filtering through cross-file semantic resolution and control-flow disambiguation, with 51.2% of successful cases relying on evidence from non-target files such as helper classes and configurations. However, performance is limited by conservative threat modeling and incorrect CWE attribution, where agents often prioritize hardening guidelines or side-channel leaks over the specific benchmark ground truth.

5 Discussion

5.1 Cost Analysis

Table 8. Average resource consumption and operational metrics per task across different models and agents.

Model	Agent	Avg. Rounds	Avg. Tokens	Avg. Cost (USD)
Claude Sonnet 4	Aider	1.0	13,321	0.0469
	OpenHands	20.5	219,299	0.1867
	SWE-agent	9.0	130,846	0.1501
DeepSeek Chat	Aider	1.0	11,036	0.0028
	OpenHands	27.1	269,260	0.0252
	SWE-agent	13.6	189,937	0.0215
GPT-5	Aider	1.0	12,564	0.0311
	OpenHands	7.4	75,205	0.0599
	SWE-agent	8.3	125,106	0.1052

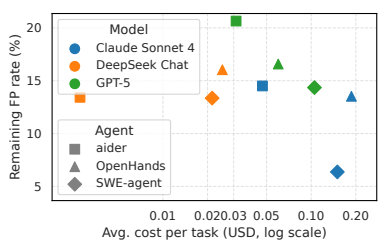


Fig. 8. Cost effectiveness of agent frameworks (OWASP FP Reduction). Lower left is better.

Table 8 and Figure 8 show that agent frameworks differ sharply in interaction cost. OpenHands is the most iterative, averaging 20.5 rounds with Claude Sonnet 4 and 27.1 with DeepSeek Chat, while Aider finishes in one round and behaves closer to a prompting engine. SWE-agent uses fewer rounds than OpenHands but much heavier context per round, consuming 130,846 tokens per Claude run. Cost follows these patterns: OpenHands+Claude is the most expensive per task ($0.1867), SWE-agent+GPT-5 is costly due to token load ($0.1052), and Aider+DeepSeek is the cheapest ($0.0028). These results show a practical cost–effectiveness frontier with no uniformly dominant agent.

5.2 True-Positive Retention

While RQ1 focuses on filtering FPs among SAST alerts, an equally important concern is whether these agents suppress true vulnerabilities. To quantify this risk, we conducted a TP-retention experiment on the OWASP Benchmark positives using the best-performing configuration from Table 3, i.e., Claude Sonnet 4 paired with SWE-agent. Among the 1,415 vulnerable test cases, the agent successfully completed 1,411 runs, achieving a 99.7% completion rate. The agent incorrectly labeled 314 true vulnerabilities as FPs, resulting in a TP retention of 77.7% and a miss rate of 22.25%. These results suggest that despite strong FP reduction on non-vulnerable instances, the agent may still hide a non-trivial fraction of real vulnerabilities if its decisions drive automatic suppression.

Our per-CWE analysis shows that the miss rate remains low for classic data-flow-driven injection vulnerabilities. Specifically, the miss rate is 2.38% for CWE-78, 1.10% for CWE-89, 0.41% for CWE-79, and 0% for both CWE-90 and CWE-643. This indicates that the agent is generally reliable when reasoning about explicit taint-flow evidence in local code. In contrast, we observe severe TP suppression for weaknesses that are less obvious to straightforward data-flow reasoning. For instance, the miss rate rises to 77.17% for CWE-327 (Weak Cryptography), 84.50% for CWE-328 (Weak Hashing), 77.11% for CWE-501 (Trust Boundary Violation), and 50.00% for CWE-614 (Secure Cookie Flag). These categories often require domain knowledge such as algorithm strength, configuration context, or threat modeling. In these cases, the agent may dismiss alerts as best-practice warnings rather than exploitable vulnerabilities, or infer mitigations that do not exist.

5.3 Data Contamination Check

A potential concern is that the models may have seen our evaluation datasets during training (data leakage), inflating performance through memorization rather than genuine reasoning capability. We address this through three complementary checks. First, we conducted a filename-only leakage probe on the OWASP Benchmark: we provided only the case identifier (e.g., `BenchmarkTest00171`), then asked each model to predict whether the case was vulnerable or non-vulnerable. All three models remained at chance level (Claude Sonnet 4: 51.6% accuracy, GPT-5: 51.0%, DeepSeek Chat: 49.8%), indicating no label memorization. Second, we scanned all 15,636 OWASP agent trajectories for explicit prior-exposure language audit such as "training data," "memorized," or "I've seen this before." No trajectory contained claims of prior benchmark, file, or answer exposure. Third, we conducted an evaluation on five OSS-Fuzz projects: mpv, libhevc, libical, gpsd, and quickjs. We selected them by scanning backward from the most recent OSS-Fuzz records in the OSV export, deriving vulnerable commits from OSV fixing commits, constructing vulnerable repository snapshots, and running CodeQL. The selected vulnerable and fixing commits postdate the evaluated model's known training cutoff by 10–12 months. On this contamination-free C/C++ dataset, the agent still achieves 95.5% F1, indicating that the gains generalize beyond public Java benchmarks.

5.4 Scope Drift in Weaker Models

We first observe a counterintuitive degradation for DeepSeek Chat in RQ1: the vanilla baseline achieves the lowest remaining FPR among DeepSeek configurations, whereas adding SWE-agent increases it. To understand why, we conduct a targeted trajectory audit rather than attributing the result to generic tool-use noise. The audit compares 1,303 paired OWASP runs for which both vanilla DeepSeek and DeepSeek + SWE-agent completed. We identify 145 cases where vanilla DeepSeek is correct but DeepSeek + SWE-agent is wrong. All 145 are ground-truth non-vulnerable cases, and in every case the agent changes the classification to TP. Thus, the degradation is a precision failure: additional agentic behaviour causes DeepSeek to over-report FPs as vulnerabilities. We then audit the final trajectory rationales for these 145 cases; the categories overlap, but the pattern is clear: 80

trajectories mention stack-trace or `printStackTrace` exposure, 93 mention information or error disclosure, 54 mention cookie-hardening issues such as `HttpOnly` or `SameSite`, 23 mention carriage-return/line-feed or response-splitting concerns, 35 mention weak cryptography, and 97 mention exception-handling concerns such as `IOException`. Three example cases illustrate this scope drift. In `BenchmarkTest00054`, vanilla DeepSeek correctly returns an FP verdict because the cross-site scripting path is HTML-encoded, whereas SWE-AGENT returns a TP verdict by aggregating stack-trace exposure, missing cookie attributes, and unhandled `IOException`. In `BenchmarkTest00089`, the agent converts a secure-cookie FP verdict into a speculative carriage-return/line-feed response-splitting TP verdict. In `BenchmarkTest00225`, it escalates ordinary servlet exception propagation into resource-leak, denial-of-service, and information-disclosure concerns. These cases lead to a scope-drift explanation: for DeepSeek, the agentic loop shifts from alert verification to broader vulnerability hunting. The extra context therefore amplifies speculative hypotheses instead of correcting them, degrading precision despite increased exploration.

5.5 Lessons Learnt

Lesson 1: LLM-based agents can remove most SAST FPs, but only for certain vulnerability categories. *Evidence.* Section 4.1 shows that LLM-based agents can reduce SAST noise on the OWASP Benchmark: the best configuration, i.e., Claude Sonnet 4 with SWE-AGENT, lowers the remaining FPR from over 98% to 6.3%, eliminating more than 92% of FPs (Table 3). However, the CWE-level analysis in Section 4.1.3 reveals a strong skew: residual FPs are concentrated in a few categories, particularly CWE-327, CWE-330, and CWE-614, while injection-style vulnerabilities (e.g., CWE-78, CWE-79, CWE-89) are filtered almost completely. *Lesson.* LLM-based agents are highly effective FP filters for data-flow-driven vulnerabilities, but their effectiveness drops sharply for policy- and cryptography-related weaknesses that require domain knowledge or threat modeling.
Lesson 2: The value of agentic reasoning depends more on the backbone model than on the agent framework. *Evidence.* The comparison in Section 4.1 shows that agentic frameworks consistently outperform vanilla LLM prompting for stronger backbone models such as Claude Sonnet 4 and GPT-5, reducing the remaining FPR from 23.0% to 6.3% for Claude. In contrast, for DeepSeek Chat, vanilla prompting already achieves a low remaining FPR (11.2%), and adding agentic scaffolding provides no consistent benefit and can even degrade performance. This backbone dependency is further confirmed by the per-CWE agentic behaviour gain/loss analysis in Section 4.1.3 (Figure 7). *Lesson.* Agentic reasoning amplifies the strengths of capable backbone models but does not compensate for weaker ones; backbone selection matters more than agent framework choice.
Lesson 3: Aggressive FP suppression risks hiding real vulnerabilities and should not be fully automated. *Evidence.* While Section 4 focuses on FP reduction on non-vulnerable cases, the TP retention analysis in Section 5.2 shows that even the best-performing configuration incorrectly suppresses 22.25% of real vulnerabilities on the OWASP Benchmark positives. This risk is highly CWE-dependent: miss rates are negligible for injection-style vulnerabilities but exceed 50% for cryptography- and policy-related categories such as CWE-327, CWE-328, CWE-501, and CWE-614. *Lesson.* LLM-based agents are unsuitable for unconditional, automatic suppression of SAST warnings; they should instead be deployed as decision-support tools, especially for vulnerability categories with high safety risk.

5.6 Threats to Validity

Threats to Internal Validity. The first threat stems from the inherent non-determinism of LLMs, which may compound across multiple reasoning steps in agentic frameworks and affect the reproducibility of this study. We mitigate this by setting the temperature to zero, which promotes, but does not guarantee, deterministic results [7]. Another threat concerns the size and labeling process

of the real-world alert samples. For Vul4J, our RQ2 evaluation includes 50 sampled CodeQL alerts due to limited manual triage capacity, and most labels were assigned by the first author after patch matching. Although we use a fixed random seed to ensure reproducibility, the sample may not fully represent the original alert distribution, and the measured performance can be affected by the rule mix (some CodeQL rules are inherently noisier than others). For OSS-Fuzz, we mitigate this concern through an independent dual-rater protocol with substantial agreement, but the dataset is still a focused 50-alert validation set rather than a comprehensive C/C++ benchmark. Also, our evaluation follows prior work on static-analysis warning validation, where the reported static-analysis output (alert) is the unit of analysis and each warning is judged against an alert-specific TP/FP label [30, 31, 60]. This design matches the SAST triage setting: developers must decide whether to act on the reported alert, not whether the surrounding code contains any possible security issue. However, this alert-specific framing has a limitation: an agent may correctly reject the reported vulnerability rationale while identifying a different real security issue in the same code path. Under our current metric definition, such cases count as incorrect because correctness is defined with respect to the specific labeled alert rather than broader security utility. We acknowledge that this binary framing may undercount cases where the agent provides actionable security value beyond the scope of the original alert.

Threats to External Validity. To enhance the generalizability of our findings, our evaluation combines a controlled Java benchmark, real-world Java projects, and a focused post-cutoff C/C++ validation set. However, the full cross-model and cross-agent comparison is conducted on OWASP and Vul4J, while OSS-Fuzz is used specifically to test contamination-free generalization and isolate agentic capabilities for the strongest configuration. Therefore, the OSS-Fuzz results should not be interpreted as a complete ranking of all agents or backbone models on C/C++. Second, although our real-world evaluation primarily utilizes CodeQL alerts, we mitigate tool-specific bias by first analyzing the correlation of FPs across multiple industrial scanners (Semgrep and SonarQube), ensuring that our agent systems target shared semantic blind spots rather than isolated analyzer errors. Finally, we evaluate three state-of-the-art models and agent frameworks to reduce selection bias.

6 Conclusion and Future Work

In this study, we provide a comparative evaluation of three LLM-based agent frameworks, i.e., Aider, OpenHands, and SWE-agent, for filtering FPs generated by SAST tools. Experimental results indicate that agentic reasoning is effective. In the best-performing configuration, using SWE-agent with Claude Sonnet 4, the agentic workflow reduced the initial FP rate of 98.3% on the OWASP Benchmark to 6.3%, effectively removing over 92% of the noise. Similarly, in real-world scenarios involving CodeQL alerts on Java, the agents achieved an FP identification rate of up to 93.3%. Further evaluation on unseen C/C++ vulnerabilities sampled from OSS-Fuzz shows that this advantage is not explained solely by memorization of public Java benchmarks: on these projects, SWE-agent with Claude Sonnet 4 achieves 96.0% accuracy and 95.5% F1, substantially outperforming vanilla, LLM4SA-style, context-augmented, and oracle-context baselines. The effectiveness of these agents is non-uniform and depends on both the backbone model and the vulnerability category. Claude Sonnet 4 and GPT-5 improved filtering performance when deployed within an agentic framework compared to vanilla zero-shot prompting, reducing residual FPR from 23.0% to 6.3%, whereas weaker backbones showed no consistent gain from added agentic loops. However, this reliability is concentrated in data-flow-dependent injection categories, whereas weakness families that require domain-specific heuristics or policy understanding retain higher residual FPs and carry a substantially greater risk of suppressing true positives.

In the future, we plan to develop CWE-aware, cost-adaptive agent strategies and human-in-the-loop auditing mechanisms to maintain high FP filtering rates while minimizing the loss of TPs, and to evaluate their generalizability across additional programming languages and SAST tools.

Data-Availability Statement

To ensure the reproducibility of our results and to provide transparency in our research, we have made all related scripts and data publicly available. All resources can be accessed as part of our anonymized artifact, which is available at https://doi.org/10.5281/zenodo.18420284 [6].

References

[1] 2025. CodeQL. https://codeql.github.com/. Accessed: 2025-12-14. https://codeql.github.com/

[2] 2025. National Vulnerability Database (NVD). https://www.nist.gov/itl/nvd. Accessed: 2026-01-30. https://www.nist.gov/itl/nvd

[3] 2026. NVD Vulnerability Visualizations: CWE Over Time. https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time. Accessed: 2026-01-30. https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cwe-over-time

[4] Vishwanath Akuthota, Raghunandan Kasula, Sabiha Tasnim Sumona, Masud Mohiuddin, Md Tanzim Reza, and Md Mizanur Rahman. 2023. Vulnerability Detection and Monitoring Using LLM. In *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*. 309–314. doi:10.1109/WIECON-ECE60392.2023.10456393

[5] Bushra Aloraini, Meiyappan Nagappan, Daniel M. German, Shinpei Hayashi, and Yoshiki Higo. 2019. An empirical study of security warnings from static application security testing tools. *J. Syst. Softw.* 158, C (Dec. 2019), 25 pages. doi:10.1016/j.jss.2019.110427

[6] Anonymous. 2026. *ISSTA 2026 Artifact Package for Sifting the Noise: A Comparative Study of LLM Agents in Vulnerability False Positive Filtering*. doi:10.5281/zenodo.21282004

[7] Merve Astekin, Max Hort, and Leon Moonen. 2024. An Exploratory Study on How Non-Determinism in Large Language Models Affects Log Parsing. In *Proceedings of the ACM/IEEE 2nd International Workshop on Interpretability, Robustness, and Benchmarking in Neural Software Engineering* (Lisbon, Portugal) *(InteNSE '24)*. Association for Computing Machinery, New York, NY, USA, 13–18. doi:10.1145/3643661.3643952

[8] Gareth Bennett, Tracy Hall, Emily Winter, and Steve Counsell. 2024. Semgrep*: Improving the Limited Performance of Static Application Security Testing (SAST) Tools. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 614–623. doi:10.1145/3661167.3661262

[9] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. 2025. RepairAgent: An Autonomous, LLM-Based Agent for Program Repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 2188–2200. doi:10.1109/ICSE55347.2025.00157

[10] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 464–468. doi:10.1145/3524842.3528482

[11] Thanh-Long Bui, Hoa Khanh Dam, and Rashina Hoda. 2025. An LLM-based multi-agent framework for agile effort estimation. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1032–1043. doi:10.1109/ASE63991.2025.00090

[12] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. 2015. Moving Fast with Software Verification. In *NASA Formal Methods*, Klaus Havelund, Gerard Holzmann, and Rajeev Joshi (Eds.). Springer International Publishing, Cham, 3–11. doi:10.1007/978-3-319-17524-9_1

[13] Yiran Cheng, Ting Zhang, Lwin Khin Shar, Zhe Lang, David Lo, Shichao Lv, Dongliang Fang, Zhiqiang Shi, and Limin Sun. 2026. Fixseeker: An Empirical Driven Graph-based Approach for Detecting Silent Vulnerability Fixes in Open Source Software. *ACM Trans. Softw. Eng. Methodol.* (April 2026). Just Accepted. doi:10.1145/3803410

[14] Yiran Cheng, Ting Zhang, Lwin Khin Shar, Shouguo Yang, Chaopeng Dong, David Lo, Shichao Lv, Zhiqiang Shi, and Limin Sun. 2026. Vercation: Precise Vulnerable Open-Source Software Version Identification Based on Static Analysis and LLM. *IEEE Transactions on Software Engineering* 52, 2 (2026), 376–394. doi:10.1109/TSE.2025.3599581

[15] Nesara Dissanayake, Asangi Jayatilaka, Mansooreh Zahedi, and M. Ali Babar. 2022. Software security patch management - A systematic literature review of challenges, approaches, tools and practices. *Information and Software Technology* 144 (2022), 106771. doi:10.1016/j.infsof.2021.106771

[16] Xueying Du, Kai Yu, Chong Wang, Yi Zou, Wentai Deng, Zuoyu Ou, Xin Peng, Lingming Zhang, and Yiling Lou. 2025. Minimizing False Positives in Static Bug Detection via LLM-Enhanced Path Feasibility Analysis. arXiv:2506.10322 [cs.SE] https://arxiv.org/abs/2506.10322

[17] Paul Gauthier. 2025. Aider - AI Pair Programming in Your Terminal. https://aider.chat/. Accessed: 2025-12-14. https://aider.chat/

[18] Paul Gauthier. 2026. Aider, release v0.86.1. https://github.com/aider-AI/aider/releases/tag/v0.86.1. Accessed: 2026-05-21. https://github.com/aider-AI/aider/releases/tag/v0.86.1

[19] GitHub. 2026. CodeQL CLI Binaries, release v2.23.7. https://github.com/github/codeql-cli-binaries/releases/tag/v2.23.7. Accessed: 2026-05-21. https://github.com/github/codeql-cli-binaries/releases/tag/v2.23.7

[20] GitHub. 2026. CodeQL Full CWE Coverage. https://codeql.github.com/codeql-query-help/full-cwe/. Accessed: 2026-05-21. https://codeql.github.com/codeql-query-help/full-cwe/

[21] Damian Gnieciak and Tomasz Szandala. 2025. Large Language Models Versus Static Code Analysis Tools: A Systematic Benchmark for Vulnerability Detection. *IEEE Access* 13 (2025), 198410–198422. doi:10.1109/ACCESS.2025.3635168

[22] Google. 2016. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz. Accessed: 2026-04-17.

[23] 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 Trans. Softw. Eng.* 49, 12 (Dec. 2023), 5154–5188. doi:10.1109/TSE.2023.3329667

[24] Juan R. Bermejo Higuera, Javier Bermejo Higuera, Juan A. Sicilia Montalvo, Javier Cubo Villalba, and Juan José Nombela Pérez. 2020. Benchmarking Approach to Compare Web Applications Static Analysis Tools Detecting OWASP Top Ten Security Vulnerabilities. *Computers, Materials & Continua* 64, 3 (2020), 1555–1577. doi:10.32604/cmc.2020.010885

[25] Huimin Hu, Yingying Wang, Julia Rubin, and Michael Pradel. 2025. An Empirical Study of Suppressed Static Analysis Warnings. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE014 (June 2025), 22 pages. doi:10.1145/3715729

[26] joern.io. 2025. Documentation of Joern Scan. https://docs.joern.io/scan/. Accessed: 2025-12-14. https://docs.joern.io/scan/

[27] joern.io. 2025. Joern — The Bug Hunter's Workbench. https://joern.io/. Accessed: 2025-12-14. https://joern.io/

[28] joern.io. 2025. Preshipped Joern Scan Ruleset for Java. https://github.com/joernio/joern/tree/master/querydb/src/main/scala/io/joern/scanners/java. Accessed: 2025-12-14. https://github.com/joernio/joern/tree/master/querydb/src/main/scala/io/joern/scanners/java

[29] joern.io. 2026. Joern, release v4.0.454. https://github.com/joernio/joern/releases/tag/v4.0.454. Accessed: 2026-05-21. https://github.com/joernio/joern/releases/tag/v4.0.454

[30] Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. 2021. Validating static warnings via testing code fragments. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 540–552. doi:10.1145/3460319.3464832

[31] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting false alarms from automatic static analysis tools: how far are we?. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 698–709. doi:10.1145/3510003.3510214

[32] Avishree Khare, Saikat Dutta, Ziyang Li, Alaia Solko-Breslin, Rajeev Alur, and Mayur Naik. 2025. Understanding the Effectiveness of Large Language Models in Detecting Security Vulnerabilities. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 103–114. doi:10.1109/ICST62969.2025.10988968

[33] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, and Neel Sundaresan. 2022. Learning to reduce false positives in analytic bug detectors. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1307–1316. doi:10.1145/3510003.3510153

[34] Ahmed Lekssays, Hamza Mouhcine, Khang Tran, Ting Yu, and Issa Khalil. 2025. LLMxCPG: Context-Aware Vulnerability Detection Through Code Property Graph-Guided Large Language Models. In *Proceedings of the 34th USENIX Security Symposium*. 489–507. https://www.usenix.org/conference/usenixsecurity25/presentation/lekssays

[35] Valentina Lenarduzzi, Fabiano Pecorelli, Nyyti Saarimaki, Savanna Lujan, and Fabio Palomba. 2023. A critical comparison on six static analysis tools: Detection, agreement, and precision. *J. Syst. Softw.* 198, C (April 2023), 19 pages. doi:10.1016/j.jss.2022.111575

[36] Frank Li and Vern Paxson. 2017. A Large-Scale Empirical Study of Security Patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Association for Computing Machinery, Dallas, TX, USA, 2201–2215. doi:10.1145/3133956.3134072

[37] Kaixuan Li, Sen Chen, Lingling Fan, Ruitao Feng, Han Liu, Chengwei Liu, Yang Liu, and Yixiang Chen. 2023. Comparison and Evaluation on Static Application Security Testing (SAST) Tools for Java. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 921–933. doi:10.1145/3611643.3616262

[38] Yikun Li, Ngoc Tan Bui, Ting Zhang, Chengran Yang, Xin Zhou, Martin Weyssow, Jinfeng Jiang, Junkai Chen, Huihui Huang, Huu Hung Nguyen, et al. 2025. Out of Distribution, Out of Luck: How Well Can LLMs Trained on Vulnerability Datasets Detect Top 25 CWE Weaknesses? *arXiv preprint arXiv:2507.21817* (2025). doi:10.48550/arXiv.2507.21817

[39] Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities. In *Proceedings of the International Conference on Learning Representations (ICLR)*. https://openreview.net/forum?id=9LdJDU7E91

[40] Zongjie Li, Zhibo Liu, Wai Kin Wong, Pingchuan Ma, and Shuai Wang. 2024. Evaluating C/C++ Vulnerability Detectability of Query-Based Static Application Security Testing Tools. *IEEE Trans. Dependable Secur. Comput.* 21, 5 (Sept. 2024), 4600–4618. doi:10.1109/TDSC.2024.3354789

[41] Zhen Li, Ning Wang, Deqing Zou, Yating Li, Ruqian Zhang, Shouhuai Xu, Chao Zhang, and Hai Jin. 2024. On the Effectiveness of Function-Level Vulnerability Detectors for Inter-Procedural Vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 157, 12 pages. doi:10.1145/3597503.3639218

[42] Feng Lin, Dong Jae Kim, and Tse-Hsun (Peter) Chen. 2025. SOEN-101: Code Generation by Emulating Software Process Models Using Large Language Model Agents. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. IEEE Press, 1527–1539. doi:10.1109/ICSE55347.2025.00140

[43] OpenHands. 2026. OpenHands Software Agent SDK, release v1.5.2. https://github.com/OpenHands/software-agent-sdk/releases/tag/v1.5.2. Accessed: 2026-05-21. https://github.com/OpenHands/software-agent-sdk/releases/tag/v1.5.2

[44] OWASP Foundation. 2025. OWASP Benchmark for Java (Source Repo). https://github.com/OWASP-Benchmark/BenchmarkJava. Accessed: 2025-12-14. https://github.com/OWASP-Benchmark/BenchmarkJava

[45] OWASP Foundation. 2025. Source Code Analysis Tools | OWASP Foundation. https://owasp.org/www-community/Source_Code_Analysis_Tools. Accessed: 2025-12-14. https://owasp.org/www-community/Source_Code_Analysis_Tools

[46] OWASP Foundation. 2025. The OWASP Benchmark Project. https://owasp.org/www-project-benchmark/#div-java_test_cases. Accessed: 2025-12-14. https://owasp.org/www-project-benchmark/#div-java_test_cases

[47] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 383–387. doi:10.1109/MSR.2019.00064

[48] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 598–608. doi:10.1109/ICSE.2015.76

[49] Semgrep, Inc. 2026. Semgrep, release v1.145.0. https://github.com/semgrep/semgrep/releases/tag/v1.145.0. Accessed: 2026-05-21. https://github.com/semgrep/semgrep/releases/tag/v1.145.0

[50] SonarSource. 2025. SonarQube: Code Quality & Security | Static Analysis Tool. https://www.sonarsource.com/products/sonarqube/. Accessed: 2025-12-14. https://www.sonarsource.com/products/sonarqube/

[51] SonarSource. 2026. SonarQube Community Docker image, version 9.9.8-community. https://hub.docker.com/layers/library/sonarqube/9.9.8-community/. Accessed: 2026-05-21. https://hub.docker.com/layers/library/sonarqube/9.9.8-community/

[52] SWE-agent Team. 2026. SWE-agent, commit 1d3cfb798a2c257a0fc6094f1d35ef084d9919e1. https://github.com/SWE-agent/SWE-agent/commit/1d3cfb798a2c257a0fc6094f1d35ef084d9919e1. Accessed: 2026-05-21. https://github.com/SWE-agent/SWE-agent/commit/1d3cfb798a2c257a0fc6094f1d35ef084d9919e1

[53] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2024. CORE: Resolving Code Quality Issues using LLMs. *Proc. ACM Softw. Eng.* 1, FSE, Article 36 (July 2024), 23 pages. doi:10.1145/3643762

[54] 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. 2025. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=OJd3ayDDoF

[55] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Trans. Knowl. Discov. Data* 18, 7, Article 168 (June 2024), 34 pages. doi:10.1145/3653718

[56] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: agent-computer interfaces enable automated software engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 1601, 125 pages.

[57] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/2210.03629

[58] Zheng Yu, Ziyi Guo, Yuhang Wu, Jiahao Yu, Meng Xu, Dongliang Mu, Yan Chen, and Xinyu Xing. 2025. Patchagent: A practical program repair agent mimicking human expertise. In *Proceedings of the 34th USENIX Security Symposium (USENIX Security'25), Seattle, WA, USA.*

[59] Ting Zhang, Yikun Li, Chengran Yang, Ratnadira Widyasari, Yue Liu, Ngoc Tan Bui, Phuc Thanh Nguyen, Yan Naing Tun, Ivana Clairine Irsan, Huu Hung Nguyen, Huihui Huang, Jinfeng Jiang, Lwin Khin Shar, Eng Lieh Ouh, David Lo, Hong Jin Kang, Yide Yin, and Wen Bin Leow. 2026. TitanCA: Lessons from Orchestrating LLM Agents to Discover 100+ CVEs. arXiv:2604.17860 [cs.CR] doi:10.48550/arXiv.2604.17860

[60] Yunhui Zheng, Saurabh Pujar, Burn Lewis, Luca Buratti, Edward Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 111–120. doi:10.1109/ICSE-SEIP52600.2021.00020

[61] Xin Zhou, Ting Zhang, and David Lo. 2024. Large Language Model for Vulnerability Detection: Emerging Results and Future Directions. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results* (Lisbon, Portugal) *(ICSE-NIER'24)*. Association for Computing Machinery, New York, NY, USA, 47–51. doi:10.1145/3639476.3639762

Received 2026-01-30; accepted 2026-06-25