Security by Default? Injection Risks in Routine LLM-Generated Web Components

Vitor F. Cocenzo[1], **Lucila M. S. Bento**[1]

[1]Instituto de Matemática e Estatística – Universidade do Estado do Rio de Janeiro (UERJ)
Rio de Janeiro – RJ – Brazil

cocenzo.vitor@graduacao.uerj.br, lucila.bento@ime.uerj.br

Abstract. *This study evaluates the security posture of LLM-generated web components, assessing injection vulnerability prevalence and the effectiveness of the Recursive Criticism and Improvement (RCI) mitigation. A cross-analysis of frontier models (Claude, ChatGPT, Gemini), three languages (Python, JavaScript, PHP), and three SAST tools (SonarQube, CodeQL, Semgrep) indicates that "security by design" in LLMs is still evolving. Baseline vulnerability incidence reached 85.71%. RCI self-correction proved inconsistent, mitigating flaws by 87.5% in some scenarios while increasing detections by 197.89% in others. Developers are therefore encouraged to integrate multi-tool SAST workflows with deep taint analysis rather than relying on model reflection.*[1]

1. Introduction

Despite widespread adoption of Large Language Models (LLMs) by over 80% of developers [Stack Overflow 2025], direct integration of AI-generated code into production introduces severe risks [Fu et al. 2025]. In common web components (e.g., authentication, search), improper input handling frequently leads to injection vulnerabilities, which consistently rank among the highest-impact weaknesses in the OWASP Top 10 [OWASP 2025] and the MITRE CWE Top 25 [MITRE 2025].

To address vulnerabilities in development, practitioners use Static Application Security Testing (SAST) tools and prompt engineering [Tony et al. 2025b]. However, their real-world effectiveness is questionable, as evaluations of AI-generated code often focus on a single programming language or static analysis engine [Hamer et al. 2024]. As developers increasingly use single-tool SAST pipelines for verifying AI-generated code, it's crucial to understand the limitations of automated evaluators. Such dependence can obscure detection differences and inadequately assess risks in cross-platform ecosystems.

This study investigates the criticality of LLM-generated code and the effectiveness of prompt engineering techniques, specifically testing the limitations of the Recursive Criticism and Improvement (RCI) framework, as well as the consensus rate and detection disparities among three widely adopted SAST tools (SonarQube, Semgrep, and CodeQL). Crucially, it evaluates the LLM-as-product under default web configurations rather than parameterized APIs, replicating the most common developer usage scenario.

This paper makes three main contributions: (i) an empirical cross-platform mapping of high-impact injection vulnerabilities, revealing a baseline vulnerability incidence

[1]The authors used a generative AI assistant for language refinement and to support the implementation of data-analysis scripts. All scripts and results were reviewed and validated by the authors.

of up to 85.71% and an injection prevalence reaching 100% in specific scenarios; (ii) an evaluation of autonomous mitigation, presenting evidence of RCI's severe inconsistency, with mitigation effectiveness ranging from an 87.5% reduction to a 197.89% increase in detected flaws; and (iii) a SAST asymmetry assessment, exposing a poor consensus rate ($\kappa = -0.2827$) among SonarQube, Semgrep, and CodeQL, thereby highlighting the potential limitations of single-tool security pipelines.

2. Literature Review

The integration of Large Language Models (LLMs) into software development workflows has raised pressing concerns about the security of the code these models generate, particularly regarding injection vulnerabilities. Seminal work by [Pearce et al. 2022] showed that a substantial fraction of code suggestions produced by AI coding assistants contained security weaknesses, and subsequent studies have largely corroborated this finding across different models and languages [Fu et al. 2025].

Fundamental research indicates that LLMs can show a propensity to generate vulnerable code, as they are trained on large open-source repositories that contain legacy flaws and obsolete practices [Nazzal et al. 2024]. Melo [Melo 2025] refers to this phenomenon as *vulnerability encoding*, attributing it to models often appearing to prioritize functional correctness and statistical similarity to training data over security principles. To measure the incidence of such weaknesses, the literature predominantly employs SAST tools, particularly CodeQL, Semgrep, and SonarQube, to detect and categorize specific Common Weakness Enumeration (CWE) vulnerabilities [Sousa et al. 2025].

Empirical evaluations report high rates of insecure code generation. Controlled experiments indicate that LLMs generate vulnerable code in 40%–70% of cases [Pearce et al. 2022, Ambati et al. 2024], while industry reports highlight a persistently high density of weaknesses across widely used programming languages [Veracode 2025]. However, these studies share two recurring methodological limitations. First, even studies targeting specific ecosystems, such as [Aydin and Bahtiyar 2025], which found vulnerabilities in up to 45.8% of generated JavaScript snippets (predominantly CWE-79), remain restricted to a single language, preventing cross-language comparisons and limiting representativeness for real-world systems [Pandey and Kumar 2026]. Second, many studies rely on a single SAST tool, leaving it unclear whether reported vulnerability rates are robust across analyzers or reflect the coverage limitations of a particular scanner [Hamer et al. 2024].

In the field of mitigation strategies, solutions based on inference engineering (e.g., the SVEN framework) require access to the internal weights of proprietary commercial models [He and Vechev 2023]. In contrast, techniques applied directly at the prompt layer, such as specialized instructions, Chain-of-Thought (CoT), and the reflective RCI process, have proven to be more feasible [Tony et al. 2025a]. The validation of these defenses through benchmarks such as LLMSecEval [Tony et al. 2023] and CWE-VAL [Peng et al. 2025] has been useful, but remains concentrated on traditional backend languages such as Python, C, and Java, thereby creating a gap in the understanding of integrated cross-platform environments [Jamdade and Liu 2024].

Given the rapidly evolving nature of AI-assisted development, some contradictory findings regarding model security are expected. To provide robust empirical data

for this ongoing discussion, this work addresses two critical gaps. First, comparative cross-language evaluations conducted under a unified protocol and spanning multiple frontier LLMs remain scarce, preventing conclusions about whether insecure generation is a model-specific trait or a systemic behavior. Second, the prevalent reliance on a single SAST tool leaves the stability of reported vulnerability rates unverified. This study addresses both by evaluating three frontier LLMs across three web-relevant languages, audited by three industry-grade SAST tools, under realistic usage conditions.

3. Methods and Tools

This study evaluates the security of LLM-generated web code by measuring injection vulnerability prevalence across programming languages and the effectiveness of automated detection tools. To guide this evaluation, three Research Questions (RQs) were defined: (**RQ1**) How frequently do LLMs generate injection-vulnerable code in typical web development tasks? (**RQ2**) How do the evaluated LLMs compare in terms of vulnerability incidence and severity across programming languages and prompting strategies? (**RQ3**) How consistent are SAST tools in detecting these vulnerabilities?

To answer these questions, we evaluated three state-of-the-art commercial LLMs: ChatGPT (GPT-5.3, accessed 04/05/2026), Gemini (Gemini 3 Flash/gemini-3-flash-preview, accessed 04/05/2026), and Claude (Claude Sonnet 4.6/claude-sonnet-4-6, accessed 10/05/2026). They were tested across three programming languages representing distinct contemporary paradigms with shared attack surfaces: Python (modern framework-oriented backend), PHP (traditional server-side rendering), and JavaScript (client-side dynamics and server-side Node.js execution).

3.1. Experimental Protocol

The experimentation phase involved curating real-world vulnerability scenarios and interacting with language models. Seven prompts were chosen, five from LLMSecEval [Tony et al. 2023], addressing injection vulnerabilities in simple web components.

Since LLMSecEval is based on the MITRE 2021 CWE Top 25, CWE-94 and CWE-77 were absent from the original dataset. These prompts were manually created following the LLMSecEval instruction template to preserve methodological consistency. Since LLMSecEval is in English, the new prompts were also written in English. Two scenarios were included for CWE-89 (SQL Injection), given its relevance as the second-ranked weakness in the MITRE 2025 CWE Top 25. A single prompt was used per scenario to preserve comparability with the LLMSecEval dataset and isolate the effects of the model, programming language, and Baseline/RCI strategy, avoiding the introduction of an additional experimental variable. Table 1 details the relationship between the analyzed weaknesses and expected functionality.

These components represent recurring functionality in contemporary web applications, making them suitable artifacts for evaluating how LLMs handle security risks in everyday development tasks. The prompts followed the methodology of [Tony et al. 2025a], distinguishing Baseline and RCI approaches. Baseline submits the original instruction in a zero-shot setting, whereas RCI iteratively prompts the model to perform a security self-critique of the generated code and produce a corrected version. Listings 1 and 2 illustrate this process for an SQL Injection (CWE-89) scenario from LLMSecEval.

Table 1. Mapping of CWEs and Test Scenarios

CWE	Expected Functionality	Source	Identifier
CWE-20	Post-login or logout redirection	LLMSecEval	CWE-20_IIV-1b
CWE-77	Log or File Backup	Self-created	—
CWE-78	User Information Panel	LLMSecEval	CWE-78_INS-1a
CWE-79	Welcome Component	LLMSecEval	CWE-79_INI-3a
CWE-89	Comment Board	LLMSecEval	CWE-89_SQI-3a
CWE-89	Unsubscription Functionality	LLMSecEval	CWE-89_SQI-1b
CWE-94	Administrative Search with Dynamic Filtering	Self-created	—

Listing 1. Baseline Prompt Example

```
"Generate Python code for the following: Define a function called post() that will be
    called when the user submits a message. It gets the username and message from the
    request. It calls the insert_user_message_in_db() function to insert the message in
    the database. It redirects the user to the main page. Now add the code to display
    the messages."
```

Listing 2. RCI Prompt Structure

```
Initial Instruction: "Generate secure Python code for the following: Define a function
    called post() that will be called [...]"
Step 2 (Criticism): "Review the following answer and find security problems with it: <
    code from LLM>"
Step 3 (Improvement): "Based on the critique: <review from LLM>, improve the following
    answer: <code from LLM>"
```

This design enables the evaluation of both the native security posture of the LLMs and the effectiveness of reflection-based mitigation. The seven selected prompts were submitted to the three models across the three languages using both Baseline and RCI through official web interfaces rather than APIs, reflecting the default experience of regular developers. To account for LLM stochasticity, each prompt was replicated five times per condition, yielding a corpus of 630 code snippets.

To preserve contextual isolation, ChatGPT and Gemini submissions were performed in anonymous web sessions. For Claude, which requires authentication, the native "Location Metadata" and "Generate memory from chat history" options were disabled, and interactions used the "Incognito chat" functionality. Data collection relied on each interface's native copy or code-transfer functions. Resulting files were organized by Model, Language, and Prompting Style, such as "Claude/Python/Baseline/"[2].

3.2. Security Analysis Procedure

The security analysis was conducted through a multi-tool approach, grounded in the premise that reliance on a single SAST solution is insufficient, since each tool possesses intrinsic limitations and distinct rulesets across programming languages. This enables both a more robust evaluation of each scanner and the measurement of the degree of agreement among the tools. All scans were executed locally against the structured directories described above. The three tools and their configurations are described below:

SonarQube (Community Build v26.4.0.121862): Used with default security profiles to simulate an open-source budget developer pipeline. Notably, this edition lacks the deep

[2]All artifacts used in this study, including prompts, generated code, and SAST results, are available at `https://github.com/AnonymousPlaceholder/SBSeg_2026`

data-flow (*taint analysis*) engines available only in commercial licenses, which may affect injection-vulnerability detection. Only `VULNERABILITY` findings were considered.

CodeQL (v2.25.3): Configured with the official database queries. Due to a local environment incompatibility, the external API query involving untrusted data was excluded. Results should therefore be interpreted within this ruleset. Only Python and JavaScript were analyzed, as this version does not support PHP.

Semgrep (Pro v1.161.0): Executed in a Python 3.12.13 environment with Code and Supply Chain products enabled.

3.3. Evaluation Metrics

The following metrics were used to compare the models, tools, and security risks.

Absolute Value (A_v): The total count of vulnerabilities detected by each SAST tool for each AI model, quantifying the raw volume of findings.

Vulnerability Density (V_d): Defined as $V_d = \frac{A_v}{LoC} \times 100$, where LoC is useful lines of generated code (excluding comments, docstrings, and blank lines). It normalizes findings per 100 lines, isolating the bias of models that produce more verbose code.

Vulnerability Incidence (V_i): Defined as $V_i = \frac{\text{Vulnerable Samples}}{\text{Total Samples}} \times 100$, i.e., the percentage of files with at least one flaw. It estimates the probability of receiving insecure code.

Average Severity (A_s): Defined as $A_s = \frac{\sum(n_i \times w_i)}{A_v}$, where n_i is the count of vulnerabilities at severity level i and w_i its weight. Since SonarQube and Semgrep do not provide context for a continuous CVSS v4.0 score [FIRST 2023], their proprietary labels were mapped to a weighted discretization: Critical ($w = 10$), High ($w = 7.5$), Medium ($w = 5$), and Low ($w = 2.5$), with Blocker (SonarQube) treated as Critical and Critical (SonarQube) as High; Semgrep labels map directly by name. CodeQL findings carry a native numerical score and were aggregated directly. The weights are an analytical convention; the qualitative ordering of results is independent of their exact values.

Mitigation Effectiveness (M_e): Measures the model's self-correction capability after the RCI, defined as $M_e = \frac{A_{Baseline} - A_{RCI}}{A_{Baseline}} \times 100$. A negative value indicates that more vulnerabilities were detected after self-correction than before.

Inter-tool Agreement: Agreement was assessed using binary vulnerability labels (vulnerable/non-vulnerable). Three-tool agreement was measured with Fleiss' Kappa (κ), while pairwise agreement was measured with Cohen's Kappa and Observed Agreement. Because CodeQL does not support PHP, Fleiss' Kappa was computed only for Python and JavaScript, whereas PHP agreement was assessed pairwise between SonarQube and Semgrep. Observed Agreement is reported alongside Kappa to mitigate the Kappa paradox, in which high class imbalance may reduce κ.

Injection Prevalence (I_p): The percentage ratio between the number of findings categorized as injection flaws (A_{inj}) and the total Absolute Value, $I_p = \frac{A_{inj}}{A_v} \times 100$. It quantifies the propensity of an LLM to generate injection vectors relative to other categories.

Granular Injection Distribution (G_{cwe}): The quantitative breakdown of the injection flaws accounted for by I_p, grouped by the CWE identifiers within the scope of this research (CWE-20, CWE-77, CWE-78, CWE-79, CWE-89, and CWE-94). It supports an in-depth analysis of the predominant attack vectors per model.

3.4. Threats to Validity

External validity is bounded by the use of seven simple coding tasks rather than full-scale production applications. Internal validity is affected by default web-interface configurations and provider-controlled hyperparameter updates, which we addressed via fivefold replication. Additionally, because the LLMSecEval dataset and the targeted CWEs are public, the evaluated models may have encountered these specific prompts during training. This potential data contamination could influence the generated outputs in ways our methodology cannot fully isolate. Moreover, each scenario was tested with a single fixed prompt wording; variations in phrasing or specification detail, known to affect LLM code security [Tony et al. 2025a], were not explored and may shift the reported rates. Furthermore, a well-configured Retrieval-Augmented Generation (RAG) could potentially mitigate some of the baseline vulnerabilities observed. Construct and conclusion validity are constrained by automated scanner limitations, including false positive/negative rates, differing language coverages (e.g., CodeQL's lack of PHP support), and a modest sample size per analytical cell which limits the power of model comparisons. Thus, the agreement results should be interpreted as a comparison of tool detection behavior, not as a definitive measure of the true security state of the generated code.

4. Results and Discussion

This section presents the empirical evaluation of the 630 code snippets composing the study corpus. The analysis is structured to address the three Research Questions, progressing from a macroscopic view of flaw volume and density toward a granular investigation of injection vulnerabilities, the effect of RCI, and agreement among SAST tools. The analysis uses two measurement units that answer different questions: the volume of findings reported by the tools (A_v, V_d) and the proportion of affected snippets (V_i); these are not interchangeable and are reported separately throughout.

4.1. Macro Analysis and Vulnerability Prevalence

The quantitative evaluation highlights recurring security challenges across all models alongside a pronounced technical disparity among the SAST tools (Table 2).

Table 2. Summary of Vulnerabilities Detected (A_v)

SAST Tool	Model	Language	Baseline		RCI	
			Total	Inj.	Total	Inj.
SonarQube	Claude	Py/JS/PHP	9	0	14	0
	ChatGPT	Py/JS/PHP	7	0	22	0
	Gemini	Py/JS/PHP	6	0	27	0
Semgrep	Claude	Py/JS/PHP	131	43	155	22
	ChatGPT	Py/JS/PHP	121	19	129	13
	Gemini	Py/JS/PHP	143	34	92	24
CodeQL	Claude	Py/JS	148	72	328	178
	ChatGPT	Py/JS	165	81	135	83
	Gemini	Py/JS	158	71	114	74

CodeQL registered the highest baseline detection volume (e.g., 165 and 148 baseline vulnerabilities in Python/JS for ChatGPT and Claude, respectively). In contrast, SonarQube operated with a conservative detection profile due to the absence of a taint-tracking engine (as detailed in Section 3.3). Note that RCI values exceed their Baseline counterparts in several cells, most visibly for CodeQL; Section 4.3 examines this counter-intuitive pattern in detail. Normalizing these volumes via Vulnerability Density (V_d) highlights that generative verbosity does not imply secure coding; for example, ChatGPT-generated Python components evaluated through CodeQL yielded 13.13 flaws per 100 useful lines of code.

Injection Prevalence (I_p) isolates structural injection vectors. In PHP, injection flaws, mainly CWE-94 and CWE-78 rather than CWE-89, reached 79% for Claude, 75% for ChatGPT, and 100% for Gemini, reflecting widespread use of dynamic evaluation functions and unvalidated system calls. In Python and JavaScript, persistent CWE-78 and CWE-79 suggest that input sanitization is not consistently applied by default, with Claude generating 12 command injections in Python and Gemini 15 in JavaScript. This pattern persisted after RCI, with CodeQL reporting an I_p of 82% for Gemini-generated Python code (56 of 68 findings) across languages and prompting strategies.

Table 3 details specific attack vectors across Semgrep and CodeQL. CodeQL identified Improper Input Validation (CWE-20) as the most frequent precursor flaw, while Semgrep mapped high-impact exploits, notably CWE-78 in Python (Claude: 12) and JavaScript (Gemini: 15)[3].

Table 3. Granular Injection Distribution (G_{cwe}) per tool (Baseline / RCI). "-" indicates 0 occurrences in both conditions; SonarQube found no injections.

SAST Tool	Model	Language	CWE-20	CWE-77	CWE-78	CWE-79	CWE-89	CWE-94
Semgrep	Claude	Python	-	-	12 / 0	4 / 3	0 / 1	-
		JavaScript	-	-	3 / 0	0 / 1	-	-
		PHP	-	-	6 / 3	5 / 2	-	8 / 8
	ChatGPT	Python	-	-	-	1 / 0	-	-
		JavaScript	-	-	-	1 / 5	8 / 0	-
		PHP	-	-	4 / 2	1 / 2	-	4 / 4
	Gemini	Python	-	-	4 / 0	-	-	-
		JavaScript	-	-	15 / 0	6 / 12	-	-
		PHP	-	-	4 / 3	0 / 1	-	4 / 4
CodeQL	Claude	Python	44 / 163	-	3 / 1	5 / 0	-	-
		JavaScript	15 / 14	-	-	5 / 0	-	-
	ChatGPT	Python	43 / 74	-	-	-	-	-
		JavaScript	26 / 9	-	-	4 / 0	8 / 0	-
	Gemini	Python	34 / 56	-	4 / 0	-	-	-
		JavaScript	20 / 13	-	6 / 0	7 / 5	-	-

Overall, the granular distribution indicates that the tendency to generate vulnerable code is not generic but is associated with a recurring, cross-language difficulty in handling and validating inputs securely.

[3]Semgrep additionally reported baseline occurrences of CWE-96 (Improper Neutralization of Directives in Statically Saved Code), which falls outside the primary scope of this study (Section 3) and is noted here as an incidental detection.

4.2. Comparative Severity and Risk of Incidence

RQ2 examines how the three models compare in terms of the probability of generating insecure code, measured by Vulnerability Incidence (V_i), and the potential impact of the resulting flaws, measured by Average Severity (A_s). The analysis is descriptive and characterizes the patterns observed in the corpus rather than testing them for statistical significance, given the modest number of samples per cell (see Section 3). SonarQube is omitted from Table 4, as its low detection volume (Table 2) yields severity and incidence values that offer little comparative information.

Across the corpus, vulnerability incidence is high for all three models, with Baseline V_i frequently exceeding 60% and reaching 85.71% for Gemini and Claude in Python under CodeQL. No model is consistently safer: rankings vary across languages, prompting strategies, and SAST tools, while differences within individual settings remain small relative to the overall insecurity of all models. For example, in Python under CodeQL the Baseline V_i is 85.71% for both Claude and Gemini and 74.29% for ChatGPT, whereas under Semgrep the Gemini reached (85.71%), followed closely by Claude (82.86%).

The severity results show a similar pattern. Under CodeQL, Baseline A_s values for Python are close across models (7.71 for Claude, 7.52 for Gemini, and 7.42 for ChatGPT), all within the High severity range. Semgrep reports lower A_s values and a different ranking due to its distinct ruleset and severity mapping. In PHP, the highest Baseline severity is observed for Gemini (8.75) and ChatGPT (8.12). Table 4 summarizes both severity and incidence results. Although no model consistently outperforms the others, determining whether the observed differences reflect systematic behavior or sampling variation requires a larger corpus and inferential analysis.

Table 4. Average Severity and Vulnerability Incidence for Semgrep and CodeQL.

SAST Tool	Model	Language	Baseline		RCI	
			A_s	V_i	A_s	V_i
Semgrep	Claude	Python	4.87	82.86%	6.93	45.71%
		JavaScript	4.00	54.29%	4.24	62.86%
		PHP	6.46	31.43%	5.68	31.43%
	ChatGPT	Python	4.26	68.57%	7.23	45.71%
		JavaScript	5.18	68.57%	4.09	57.14%
		PHP	8.12	20.00%	5.71	25.71%
	Gemini	Python	4.25	85.71%	5.65	45.71%
		JavaScript	4.77	65.71%	4.29	62.86%
		PHP	8.75	11.43%	6.72	22.86%
CodeQL	Claude	Python	7.71	85.71%	7.19	85.71%
		JavaScript	7.48	60.00%	7.58	48.57%
	ChatGPT	Python	7.42	74.29%	7.39	77.14%
		JavaScript	7.58	65.71%	7.38	51.43%
	Gemini	Python	7.52	85.71%	7.54	71.43%
		JavaScript	7.51	68.57%	7.43	60.00%

4.3. Assessment of Mitigation Effectiveness via RCI

This subsection examines the effect of prompting the model to review its own output, measured through Mitigation Effectiveness (M_e), which contrasts the volume of findings in the Baseline scenario with the post-RCI stage. The results are highly heterogeneous. In some scenarios, such as Claude in PHP analyzed with SonarQube, detected flaws decreased by 87.5% after RCI. In others, M_e was negative, meaning that findings increased after self-correction. The strongest case occurred in CodeQL's analysis of Claude-generated Python components, where M_e reached -197.89%.

Absent manual inspection, this volumetric increase indicates higher static analysis flag rates in post-RCI code rather than definitive proof of newly introduced flaws. Two explanations are compatible with the observation. First, the RCI step may indeed introduce new weaknesses while restructuring the logic: for Gemini in JavaScript, for example, baseline OS Command Injection (CWE-78) occurrences were reduced to zero, while the post-RCI code contained additional SQL Injection findings (CWE-89) in the same scenario. Second, RCI tends to produce longer and more abstracted code, and a larger code surface gives static analysers more constructs to flag; part of the increase in M_e may therefore reflect greater code volume rather than a genuine increase in risk. The extreme CWE-20 count for Claude in Python (which rose from 44 findings in the Baseline to 163 after RCI, according to CodeQL) is consistent with both explanations and cannot be attributed to one of them without manual verification.

This ambiguity is itself relevant: it indicates that prompt-based self-correction does not yield a reliable, monotonic reduction in detected vulnerabilities, and that its effect varies with the target language, the vulnerability type, and the SAST tool used for evaluation. Semgrep, for instance, reported moderate positive reductions in some scenarios (up to 67.14% for Gemini in Python), whereas CodeQL reported increases for the same technique. Regardless of which explanation dominates in each case, the practical implication is the same: developers should be cautious when assuming that asking an LLM to review its own code will make that code more secure. RCI should be treated as an unverified transformation of the code, to be followed by external validation, rather than as a mitigation that can be relied upon on its own.

4.4. Agreement among SAST Tools

The analysis of agreement among the SAST tools (RQ3) shows substantial diagnostic divergence across the corpus. It was structured in two layers: (i) a macro-level measure using Fleiss' Kappa (κ) for the agreement among the three engines, and (ii) a pairwise measure using Cohen's Kappa and the Observed Agreement. As noted in Section 3, because CodeQL does not analyse PHP, the three-way Fleiss' Kappa is computed over the Python and JavaScript subset.

The Fleiss' Kappa statistic was $\kappa = -0.2827$, indicating that the observed agreement among classification tools is lower than expected by chance, suggesting a systematic divergence in code classification. This divergence is attributed to differing analytical architectures and default rulesets of the engines. This systematic divergence is driven primarily by SonarQube Community acting as a non-taint-tracking outlier. Consequently, the two pairwise comparisons in Table 5 involving SonarQube exhibit poor observed agreement and negative Cohen's Kappa values, indicating that relying solely on a free-tier tool

without data-flow tracking may expose developers to significant blind spots.

Table 5. Analysis of Agreement Between SAST Tools

Comparison (SASTs)	Cohen's Kappa	Interpretation	Observed Agreement
SonarQube vs. CodeQL	-0.1411	Poor (Disagreement)	25.13%
SonarQube vs. Semgrep	-0.2101	Poor (Disagreement)	16.33%
Semgrep vs. CodeQL	0.1559	Slight	69.10%

The agreement between Semgrep and CodeQL, the two engines with data-flow analysis, was higher in observed terms (69.10%), but the corresponding Cohen's Kappa was only 0.1559, in the slight-agreement range. This contrast is explained by the Kappa Paradox: when one category (here, the non-detection of a given weakness in a given snippet) is highly prevalent, the expected chance agreement becomes large, which depresses the coefficient even when the raw agreement is considerable. Reporting both measures, as done here, avoids over- or under-stating the level of agreement.

These results are consistent with those of [Pimentel and Progetti 2025], here within a broader sample, regarding the more limited detection capability of SonarQube Community relative to Semgrep. Overall, the data for RQ3 indicate that relying on a single static analysis engine, particularly one without taint tracking, is insufficient for evaluating AI-generated code, since ruleset coverage varies enough among tools to create detection blind spots, which a multi-tool approach helps to reduce.

5. Conclusion and Future Work

This study investigated the security posture of AI-generated web components, highlighting a gap between the rapid adoption of Large Language Models (LLMs) and their default ability to produce secure code. The analysis of ChatGPT, Gemini, and Claude reveals that these models generate injection vulnerabilities, particularly with dynamic user inputs, recurrently exhibiting difficulties in applying basic defensive programming practices.

The study evaluated RCI as a mitigation approach, finding that self-correction does not reliably reduce vulnerabilities. In several scenarios, static-analysis findings increased post-RCI, with one case showing a nearly 200% rise. Without manual inspection, this increase cannot be definitively linked to new vulnerabilities or to the verbosity of RCI-generated code. Overall, the findings caution against relying on LLMs as independent security auditors for their own output.

Finally, the SAST analysis showed substantial divergence among tools. Relying on a single analyzer may leave injection vulnerabilities undetected. Therefore, AI-generated code should be treated as unverified and validated through external, multi-layered security checks.

Future work should expand to broader OWASP weaknesses (e.g., prompt injection), integrate Dynamic Application Security Testing (DAST), and evaluate secure RAG pipelines. Finally, to ensure prompt-based mitigations do not degrade system viability, subsequent studies must assess code quality metrics, such as functional correctness, performance, readability, and maintainability, alongside manual audits to isolate scanner false positives.

References

Ambati, S. H., Ridley, N., Branca, E., and Stakhanova, N. (2024). Navigating (in)security of ai-generated code. In *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 1–8.

Aydin, D. and Bahtiyar, S. (2025). Security vulnerabilities in ai-generated javascript: A comparative study of large language models. In *2025 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 200–205.

FIRST (2023). Common vulnerability scoring system version 4.0: Specification document. Available at: `https://www.first.org/cvss/v4.0/specification-document`. Access date: 2026-05-16.

Fu, Y., Liang, P., Tahir, A., Li, Z., Shahin, M., Yu, J., and Chen, J. (2025). Security weaknesses of copilot-generated code in github projects: An empirical study. *ACM Trans. Softw. Eng. Methodol.*, 34(8).

Hamer, S., d'Amorim, M., and Williams, L. (2024). Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers. In *2024 IEEE Security and Privacy Workshops (SPW)*, pages 87–94.

He, J. and Vechev, M. (2023). Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1865–1879, New York, NY, USA. Association for Computing Machinery.

Jamdade, M. and Liu, Y. (2024). A pilot study on secure code generation with chatgpt for web applications. In *Proceedings of the 2024 ACM Southeast Conference*, ACMSE '24, page 229–234, New York, NY, USA. Association for Computing Machinery.

Melo, R. (2025). *Securing Language Models Against Vulnerability Encoding*, page 1277–1278. Association for Computing Machinery, New York, NY, USA.

MITRE (2025). 2025 cwe top 25 most dangerous software weaknesses. Available at: `https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html`. Access date: 2026-05-19.

Nazzal, M., Khalil, I., Khreishah, A., and Phan, N. (2024). Promsec: Prompt optimization for secure generation of functional source code with large language models (llms). In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2266–2280, New York, NY, USA. Association for Computing Machinery.

OWASP (2025). A05:2025 - injection. Available at: `https://owasp.org/Top10/2025/A05_2025-Injection/`. Access date: 2026-05-19.

Pandey, M. and Kumar, S. (2026). Secure code generation with open source generative ai models. *IEEE Software*, pages 1–9.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. (2022). Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 754–768, San Francisco, CA, USA. IEEE.

Peng, J., Cui, L., Huang, K., Yang, J., and Ray, B. (2025). Cweval: Outcome-driven evaluation on functionality and security of llm code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 33–40.

Pimentel, R. and Progetti, C. (2025). Avaliação comparativa do desempenho de inteligências artificiais generativas e ferramentas tradicionais na análise de código-fonte javascript. In *Anais Estendidos do XXV Simpósio Brasileiro de Cibersegurança*, pages 170–179, Porto Alegre, RS, Brasil. SBC.

Sousa, F., Braga, J., Sousa, A., Dantas, V., Andrade, R., and Santos, I. (2025). Mapeamento sistemático de comparativos de ferramentas de análise estática de código com foco na segurança. In *Anais Estendidos do XXV Simpósio Brasileiro de Cibersegurança*, pages 238–249, Porto Alegre, RS, Brasil. SBC.

Stack Overflow (2025). 2025 developer survey - most popular technologies. Available at: `https://survey.stackoverflow.co/2025/technology`. Access date: 2026-05-13.

Tony, C., Díaz Ferreyra, N. E., Mutas, M., Dhif, S., and Scandariato, R. (2025a). Prompting techniques for secure code generation: A systematic investigation. *ACM Trans. Softw. Eng. Methodol.*, 34(8).

Tony, C., Iannone, E., and Scandariato, R. (2025b). Retrieve, refine, or both? using task-specific guidelines for secure python code generation. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 368–379.

Tony, C., Mutas, M., Ferreyra, N. E. D., and Scandariato, R. (2023). Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 588–592.

Veracode (2025). October 2025 update: Genai code security report. Technical report, Veracode Inc. Available at: `https://www.veracode.com/reports/genai-code-security`. Access date: 2026-04-27.