Evaluating the Efficiency of LLM-Generated Software in Resisting Malicious Attacks

Dominic Niceforo, Haydar Cukurtepe
Department of Computing and Information Sciences,
Valparaiso University, Valparaiso, Indiana 46383-7123

Abstract—**This study introduces a structured framework for evaluating the security of Java applications generated by large language models (LLMs) and presents the results from its implementation across three models: DeepSeek, GPT-4, and Llama 4. The framework integrates Open Web Application Security Project (OWASP)-supported tools, such as SpotBugs with FindSecBugs, OWASP Dependency Check, and OWASP Zed Attack Proxy (ZAP), alongside the NIST Risk Management Framework. These tools and standards were selected for being publicly available, allowing this process to be replicated and extended without proprietary licensing, and for their alignment with widely adopted industry benchmarks. The testing methodology for generated Java applications includes static code analysis, third-party dependency checking, and dynamic attack simulation. Each of the specified tools for this study corresponds to identifying a specific category of critical vulnerabilities. Identified vulnerabilities are then evaluated against NIST risk analysis standards to characterize their threat sources, likelihoods, and impacts, as well as their implications for the overall security risk profile of each application. The effect of prompt design is also explored by comparing a neutral prompt against a security-emphasized prompt incorporating OWASP best practices. Results varied considerably across models: GPT-4 showed noticeable improvements across critical and high-severity vulnerabilities, with 33.3% and 53.8% reductions, respectively. However, Llama 4 and DeepSeek saw an increase in vulnerabilities from the neutral to the secure prompt. Llama 4 had a general increase of 10-15% across critical, high, and medium-severity vulnerabilities, while DeepSeek saw no change in high-severity vulnerabilities and a 40% increase in low-severity vulnerabilities. The framework presented provides a structured process for evaluating LLM-generated code against established software development and security standards, while identifying present limitations and possible directions for future work.**

Keywords—Artificial intelligence; cyber security, large language models (LLMs); software security evaluation

I. Introduction

A core goal in computer security is to reduce software vulnerabilities, both by identifying existing flaws and preventing new ones from emerging. Yet, security problems are still common in software systems. For instance, a report shows that 48% of tested software contains critical, high-risk vulnerabilities [1]. Another report indicates the common vulnerabilities and exposures (CVEs) volume for 2025 is rapidly increasing, with a significant jump from previous years, 16-18% by mid-year. This change is driven by faster reporting and an increased focus on software flaws [2]. Even as advancements in AI offer new ways to aid software development, these security concerns remain, potentially creating new attack vectors in the quality of code generated.

In [3], it is noted that more than 800 million users actively engage with ChatGPT [4] each week, collectively generating over 5.8 billion queries per month. LLMs have been trained to work in a wide variety of domains and have also been integrated into software development environments. The growing use of LLMs for code generation presents both opportunities and significant challenges. As models like GPT-4 [4], DeepSeek [5], and Llama 4 [6] become better trained at generating code, there is a growing interest in handing over parts of the development process to LLMs [7]. These reasons include streamlining coding tasks, enhancing productivity, and reducing development time. While LLMs can speed up the process of writing functional code, this convenience comes with substantial risks to software quality, security, and developer accountability. A growing reliance on these models to handle code generation does not address the challenges, particularly regarding security, which remains largely untested in real-world scenarios. Traditional software development methodologies prioritize secure coding practices, such as threat modeling, code reviews, and penetration testing, to identify and address potential vulnerabilities. In contrast, LLMs face limitations during training, where models lack proper optimization for secure code generation, and during deployment, when LLMs are used for more sensitive tasks without appropriate oversight or validation. As a result, this raises significant concerns about the overall security of LLM-generated solutions. Some studies indicate that the use of LLMs negatively affects code generation. In [8], the authors conducted a controlled experiment with two groups, one of which worked with the help of AI code generators, and the other without their assistance. The authors observed that participants with access to an AI assistant wrote more insecure solutions than those without it. Similarly, Sarkar et al. [9] observed that LLM-supported programming assistants have a negative influence on code development. They also found that inexperienced users often trust code generated by AI tools without much consideration. The survey [10] also highlights the importance of clean and healthy code during LLM training.

Understanding the code-generation capabilities of LLM tools will benefit all relevant groups. This research examines the extent to which the generated code adheres to current security standards and performs under structured vulnerability assessments. Our study integrates established frameworks, such as the OWASP Top 10 [11] and the NIST Risk Management Framework [12], to evaluate Java applications generated by different LLMs. Using tools such as SpotBugs, FindSecBugs, OWASP Dependency Check, and OWASP ZAP, we conduct both static and dynamic testing, as well as third-party dependency analysis. We then analyze, classify, and score the findings using the NIST framework to produce quantifiable risk assessments.

This research aims to answer multiple key questions about the capabilities of commonly used LLM tools:

- How effective are LLMs in generating code that can resist specific cyberattacks?
- What security benchmarks can accurately evaluate the resilience of this code?
- What attack vectors are these applications most vulnerable to?

This study hypothesizes that secure prompting will influence the security profile of generated applications, with the direction and degree of change expected to vary across the different models, given potential differences in architecture and training.

By exploring these questions, this research establishes a replicable framework for evaluating the security of code generated by LLMs for a structured application. Additionally, it will provide valuable insights into how prompt design can influence more secure code output.

The following sections organize the remainder of this study: Section II provides a brief overview of the background of AI-assisted software development, along with related works, with a special emphasis on performance evaluations and security concerns. Section III expands on how the evaluation process will be implemented within the proposed framework. Section IV presents experimental results, performance comparisons, and discusses the key findings. Lastly, the conclusion in Section V discusses the current limitations of the research, possible future work, and concludes the study.

II. Background and Related Work

This study aims to investigate the efficiency of LLMs in software development, specifically their ability to generate code that withstands commonly known attack types. As AI-powered tools have become increasingly integrated into the general population, their ability to create high-quality code has been the subject of numerous studies.

The survey [10] highlights the powerful capability of LLMs, especially for code generation, when high-quality, clean, and healthy code is used for training. The authors systematically review the use of LLMs in software security. They break down software security into stages and analyze how LLMs can be utilized in each stage. The authors emphasize the powerful potential of LLMs, especially when pretrained with clean data, fine-tuned for specific tasks, and carefully crafted input prompts. A comprehensive survey of LLMs for Software Engineering (SE) is given in [13]. This survey concludes that traditional SE methods, combined with LLMs, play a critical role in enabling reliable and efficient LLM-based SE. The surveys [13], [14] provide AI-assisted development and open technical challenges that highlight their capabilities, limitations, potentials, and security concerns. In [14], the authors reviews how embeddings can be utilized for detecting security vulnerabilities by comparing them to known code patterns. They also suggest that the transformers can be trained to detect security vulnerabilities by learning from labeled data. In [15], the authors provide a survey on the usability of AI tools to assist in software development, with a focus on Copilot and GitHub. The study is based on a survey of a diverse population of developers. The authors observe that developers are motivated to use AI programming assistants; however, 54% are reluctant to do so due to security concerns. In [16], the authors provide a comprehensive survey on LLMs for security. The authors investigate the use of LLMs for identifying possible vulnerabilities in natural language descriptions and source code, as well as generating security-related code, such as patches and exploits. In [17], the authors provide a survey on the security aspects of code generated by LLMs. The authors emphasize the importance of employing robust tools and approaches to enhance code quality.

In [18], the authors provide an analysis of the quality of solutions generated by AI tools, including code writing, testing, reviewing, and documentation. The study is conducted using Claude 3 Opus and GitHub Copilot. The performance of the tools is compared with three security factors: “Provided security configuration from the start”, “Filter out malicious input”, “Up to date on libraries information”, but it does not provide a comprehensive analysis of vulnerabilities. In [19], the authors focus on generating Solidity code with GitHub Copilot. AI-generated code is scanned for vulnerability using a static analysis framework for smart contracts. Despite the advantages of using AI, there are significant challenges, particularly concerning the security, reliability, and quality of the generated code. In [20], [21], the authors focus on developer use of AI tools. In [22], the authors researches the security concerns of developers.

There are many studies evaluating the correctness and usability of the LLM-generated code [23], [24], [25]. The study [26] analyzes developments in AI-supported code generation, focusing on safety, reliability, and code quality. In [27], the authors evaluates the code generated from GitHub Copilot using MITRE vulnerability standards. The authors suggest pairing Copilot with appropriate security-aware tooling during both training and generation to minimize the risk of introducing security vulnerabilities. The study [28] proposes a controlled code generation method (svGen) using LLMs. The authors train the model with secure-accepted code and evaluate the results using a model proposed by [27]. The authors employ this approach to improve the security of LLM-generated code from 59% (for considered cases) to 92%. The evaluation model used in this study needs to be aligned with industry standards and repeatable.

In [29], the authors tested CodeBERT, GPT-3.5, and CodeX for their code generation capabilities and security concerns related to 10 different vulnerabilities, which were selected from common CVEs, Common Weakness Enumerations (CWEs), NIST vulnerabilities, and OWASP’s Top 10 web application security risks. The authors conclude that ChatGPT excels at explaining potential security issues and suggest that all AI models should be integrated with cybersecurity practices. Our study differs from this study in its comprehensive analysis of vulnerabilities, repeatability of implementation framework, modern LLM implementations, and NIST evaluations. Berabi et al. [30] investigate the effectiveness of LLMs for solving code-repair tasks. The authors present a new learning model that trains LLMs to fix coding errors and vulnerabilities. The authors concluded that improving code quality and security is difficult, as it requires large amounts of high-quality, labeled training data. Another type of risk associated with using AI tools in code generation is highlighted by [31]. The study emphasizes that an attacker can poison the data, infecting AI code generators and leading them to generate code that contains vulnerabilities and security defects. The studies [9], [22] indicate that, although developers have serious security concerns about AI-generated code, AI use will continue to increase for productivity purposes.

Prior studies in this area can be generally categorized into three groups: the first focuses on performing static analysis on LLM-generated code, such as [27] and [19]. The second group examines developer behavior and potential concerns arising from the use of AI tools in practice, such as [9], [22], and [15]. The third group explores the broader capabilities and generation approaches, such as [29] and [28].

The present study differs from all three categories by combining static analysis, third-party dependency checking, and dynamic attack simulation within a single evaluation pipeline and assessing it against the NIST Risk Management Framework. This reflects the multi-faceted nature of real-world security threats, where no single analysis method captures the full risk profile of an application. By integrating the specializations of prior work into a unified framework, this study provides a more complete picture of security vulnerabilities in a given code-generation scenario.

III. Methodology

This section details the implementation of the evaluation process and proposes a framework that integrates software generation, security benchmarking, and attack testing across selected LLMs.

The attack vectors tested in this study are based on the OWASP Top Ten [11], an openly accessible document that outlines the most common and critical security risks faced by web applications. These include:

- Broken Access Control
- Cryptographic Failures
- Injection
- Insecure Design
- Security Misconfiguration
- Vulnerable and Outdated Components
- Identification and Authentication Failures
- Software and Data Integrity Failures
- Security Logging and Monitoring Failures
- Server-Side Request Forgery

The OWASP Top Ten list is globally recognized for providing a thorough, systematic approach to identifying, prioritizing, and classifying vulnerabilities relevant to the type of application generated in this study. We specifically chose this benchmark as our primary reference because of its widespread acceptance in academia and industry [32], [33], [34]. While the OWASP Top Ten is updated regularly to keep pace with new threats and research, using the current version at the time of our study, OWASP Top Ten 2025, ensures our methods are transparent, consistent, and reproducible. By clearly documenting these categories and their integration into our testing, our framework directly aligns with current security standards, making it more credible and applicable. The framework also incorporates the NIST Risk Management Framework to deliver clear, quantifiable assessments of potential security risks.

The proposed framework comprises seven primary components, which form a sequential process for evaluating LLM-generated code from a security perspective, as illustrated in Fig. 1. Details about its implementation and varying components are explained in the following subsections.

A. LLMs and Prompts

LLMs are widely adopted and used intensively for many purposes, including software development assistance [12].

1) LLM selection: In this study, we have selected three distinct large language models: GPT-4, DeepSeek, and Llama 4. Each LLM offers a different perspective on what comprises the evolving landscape of AI. GPT-4 stands as the industry standard, well-known, widely trusted, and high-performing. DeepSeek is a newer model, and its inclusion in this study prompts a different methodological model design. Lastly, Llama 4, which is partially open-source, invites innovation and transparency. Together, these choices form a well-rounded representation of the variety expected of available models and, therefore, an accurate basis for comparison in this study.

2) Prompt preparation: This step involves drafting comprehensive project specifications for an LLM to generate a Java-based e-commerce web application. The primary goal is to ensure the instructions are easy to understand and consistently reproducible across different models. Additionally, by adding intentional complexity, the prompt acts as a controlled environment for testing how the model handles various categories of security vulnerabilities.

The neutral prompt, which serves as the baseline for creating the project, is as follows:

"Create a simple Java web application for an e-commerce platform. Include the following features: user registration and login, a product catalog with basic search functionality, an admin page to add products, and file upload support for product images. Keep the structure as simple as possible, and make sure the core features are implemented. Use H2 as the database, Spring MVC for routing, and Thymeleaf templates for the UI. Structure the application as a Maven project and include a working pom.xml with necessary dependencies to build and run the app."

In comparison, the second prompt is meant to introduce a greater focus on security. For the context of this study, only one secure prompt is used. However, a greater number of prompts and variance in its security emphasis can be beneficial, as mentioned in the Limitations section of this study. While the secure prompt is identical in its project specifications, this prompt incorporates safe coding practices that would typically be followed in a traditional development cycle, as well as types

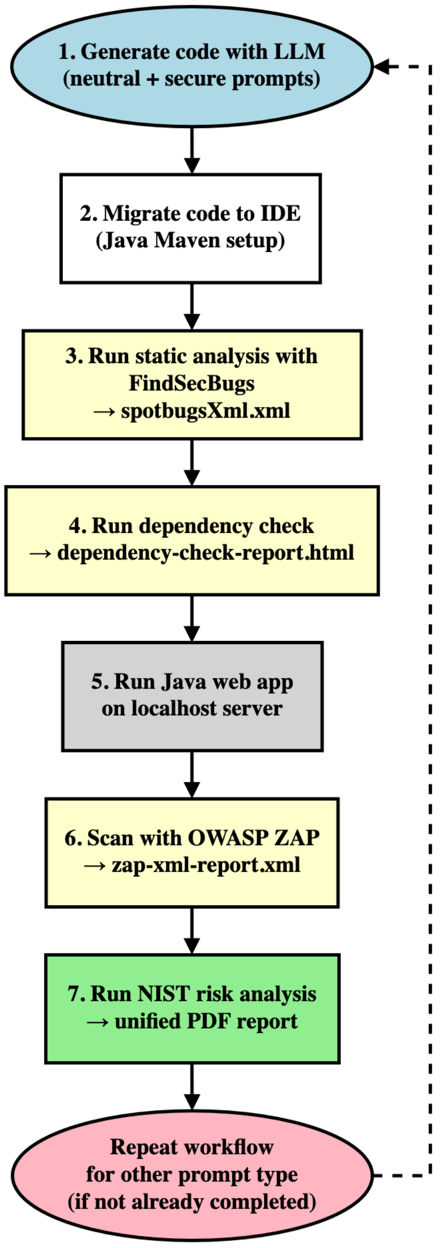

Fig. 1. Overview of the primary components within the proposed framework for evaluating LLM-generated applications.

of potential vulnerabilities to avoid when implementing core features.

“Implement the following OWASP best security practices: use the latest version of dependencies, input validation and output encoding, use prepared statements for database access, hash passwords using BCrypt, implement access control (e.g., admin vs user), validate uploaded files and store them safely, avoid hard coded secrets (use application.properties), handle errors securely (do not expose stack traces), and include comments explaining your security-related decisions.”

B. Code Migration

In order to evaluate the security of LLM-generated applications, a practical testing environment is created by initializing a new Java project in an Integrated Development Environment (IDE). This approach closely follows a real-world developer workflow, enabling an accurate testing environment that simulates realistic development and security assessments. In the current version of the study, code is migrated manually from the LLM’s output into a Maven project. However, automated integration, as an extension of this study, is considered in the Limitations section.

Within the project, the security tools explained in the following subsections are integrated via the Maven pom.xml configuration file.

C. Static Vulnerability Analysis

Static code analysis is essential in preventing costly security breaches by identifying vulnerabilities early, saving time and resources. It also helps improve code quality by enforcing standards.

SpotBugs and FindSecBugs are commonly used tools for static analysis of the source code, without actually running the code. SpotBugs checks for over 400 bug patterns, categorized by severity [35]. This tool is particularly effective in identifying issues such as hard-coded credentials, directory traversal, SQL injection, and weak cryptographic implementations. These vulnerabilities align with multiple OWASP categories, including Broken Access Control, Injection, and Cryptographic Failures. In this study, we utilized SpotBugs version 4.9.2.0 and FindSecBugs version 1.12.0 to identify vulnerabilities.

The FindSecBugs tool is compatible with the Java development environment. After configuring the tools, the IDE performs automated scans and generates detailed reports. Exported in both HTML and XML formats, these reports are stored for further review.

D. Dependency Analysis

Dependency analysis is another essential component of the study methodology, as it mitigates vulnerabilities caused by outdated dependencies and libraries in the generated application.

OWASP Dependency Check is a command-line tool that scans for known vulnerabilities in project dependencies by comparing them against databases like the National Vulnerability Database (NVD) [36]. It is used to identify vulnerabilities in third-party libraries. This tool scans through the project’s dependencies and maps them to known CVEs from public vulnerability databases. As one of the most prevalent risks for web applications, using Dependency Check in this testing methodology addresses the OWASP category of Vulnerable and Outdated Components. In this study, we used version 6.5.3 to identify vulnerabilities.

OWASP Dependency-Check tool is compatible with the Java development environment. After configuring the tools, the IDE performs automated scans and generates detailed reports. Exported in both HTML and XML formats, these reports are stored for further review.

E. Running Code on a Server

Once the static analysis and dependency checks are complete, the application is deployed locally to a web server from the IDE. This step enables dynamic analysis, which involves observing and testing the application while it is running.

F. Dynamic Vulnerability Analysis

OWASP ZAP is used at this point to scan the live application, conducting various vulnerability tests that mimic possible attack vectors by an external attacker.

OWASP ZAP (Zed Attack Proxy) simulates live attacks against an application while running on a web server for dynamic testing [37]. ZAP intercepts and manipulates HTTP requests, helping to satisfy the OWASP categories of Broken Access Control, Cross-Site Scripting (XSS), and Authentication Failures. These dynamic tests are insightful for better understanding how this application will behave under real-world conditions, where it may be subject to more varied testing than provided by static analysis or unit testing. In this study, we used version 2.16.1 to identify vulnerabilities.

G. NIST Framework Analysis

The NIST Cybersecurity Framework serves as a comprehensive and valuable resource for organizations to manage and mitigate cybersecurity risks. It offers a structured, flexible, and repeatable methodology for identifying, assessing, and addressing security vulnerabilities across diverse operational contexts [12]. By adopting this framework, organizations can not only improve their alignment with regulatory and industry compliance requirements but also substantially reduce their exposure to cyber threats. Moreover, the framework supports the development of a resilient and proactive security posture, ultimately contributing to the organization's long-term stability and risk governance.

The NIST Risk Management Framework is used to convert raw output from the OWASP vulnerability assessments into quantifiable risk levels. This step provides greater structure to the security findings, supporting risk-based decision-making for individual developers and organizations throughout the software development process.

Data collected from the OWASP tools, such as SpotBugs, FindSecBugs, Dependency Check, and ZAP, is extracted and standardized. These include information such as CWEs, Common Vulnerabilities and Exposures (CVEs), and severity scores as reported by the tools.

Additionally, Python scripts are used to automate the process of querying the National Vulnerability Database (NVD) API. These scripts retrieve CVSS (Common Vulnerability Scoring System) data for each CVE, helping to determine the likelihood and impact of each vulnerability using NIST's criteria:

- Likelihood is determined by factors such as exploitability, prevalence, and exposure, referenced from CVSS subscores and vulnerability history.
- Impact is evaluated based on the nature of the affected component and the sensitivity of the data it handles.

Final risk levels are calculated by multiplying the likelihood and impact weights. They are categorized as Low, Moderate, or High as outlined by NIST standards.

IV. Experimental Results

This section presents the experimental results from implementing the proposed framework with two prompts: a *neutral* prompt, where security requirements are not emphasized, and a *secure* prompt, where the LLM is explicitly asked to create secure code that avoids specific vulnerability groups. Results are contextualized by the framework with an additional focus on implementation details that required manual intervention.

TABLE I. Security Findings by Severity for Each LLM and Prompt.

LLM	Prompt	Critical	High	Medium	Low	Info
DeepSeek	Neutral	3	13	10	11	3
DeepSeek	Secure	2	13	14	14	4
GPT-4	Neutral	3	13	10	8	3
GPT-4	Secure	2	6	15	6	3
Llama 4	Neutral	10	9	6	11	3
Llama 4	Secure	11	10	7	11	3

A. LLM and Prompt Behavior

During the code generation process, each LLM's behavior is evaluated based on the functional quality of its output, responsiveness to the given prompt, and the degree of manual refinement required to produce a working Java application from the model (see Table I).

1) GPT-4: The neutral prompt generated a working Spring Boot web application that included basic controllers, models, and template views. Fig. 2 depicts the directory structure produced along with the source files. However, GPT-4's implementation introduced several security vulnerabilities, including 34 in total: 3 Critical, 13 High, 10 Medium, and 8 Low. These include nine significant issues reported by SpotBugs/FindSecBugs, such as a potential path-traversal issue (CWE-22) caused by improper file handling, and a failure to check the return value of the mkdirs() method (CWE-253).

OWASP ZAP reported five issues, including the absence of a Content Security Policy (CSP) header being set (CWE-693) and the use of cookies without the SameSite attribute (CWE-1275), both of which increase the web application's susceptibility to client-side attacks. However, most of the reported issues originated from OWASP Dependency Check, which identified all three vulnerabilities marked as critical. These include SnakeYAML, Spring Boot Starter Security, and Tomcat Embed Core. One or multiple published CVEs could reference each of these outdated dependencies. For example, the SnakeYAML version was linked to CVE-2022-1471, involving a deserialization vulnerability. On the other hand, Tomcat Embed Core was linked with 7 CVEs, ranging from improper input validation to path equivalence and the improper neutralization of escape sequences. Each of these vulnerabilities matches with one or more OWASP Top Ten vulnerabilities [11].

The secure prompt reduced critical-rank vulnerabilities by 1 to 2 and High-rated vulnerabilities by 7 to 6. These results

```
ecommerce-webapp/
├── pom.xml
├── src/
│   ├── main/
│   │   ├── java/
│   │   │   └── com/example/ecommerce/
│   │   │       ├── EcommerceApplication.java
│   │   │       ├── controller/
│   │   │       │   ├── AdminController.java
│   │   │       │   ├── AuthController.java
│   │   │       │   └── ProductController.java
│   │   │       ├── model/
│   │   │       │   ├── Product.java
│   │   │       │   └── User.java
│   │   │       ├── repository/
│   │   │       │   ├── ProductRepository.java
│   │   │       │   └── UserRepository.java
│   │   │       └── config/
│   │   │           └── SecurityConfig.java
│   │   └── resources/
│   │       ├── static/
│   │       ├── templates/
│   │       │   ├── login.html
│   │       │   ├── register.html
│   │       │   ├── catalog.html
│   │       │   └── admin.html
│   │       └── application.properties
└──
```

Fig. 2. GPT-4 responding to the neutral prompt with a directory structure of files necessary for generating the Java-based web application.

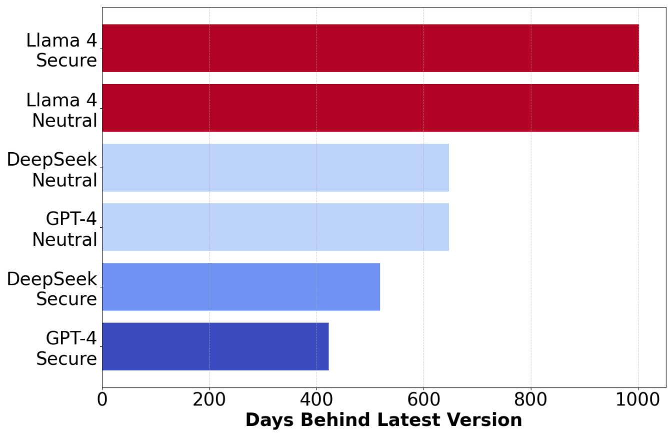

Fig. 3. Bar chart showing the number of days that the Tomcat dependency used in each application lagged behind the latest available version at the time of testing. Lower bars represent a more recent version of the dependency.

include the SnakeYAML dependency issue, which was no longer reported by Dependency Check. However, new vulnerabilities were also introduced, including disabling cross-site request forgery (CSRF) protection in the security configuration (CWE-352) and omitting an anti-clickjacking header (CWE-1021). Additionally, some vulnerabilities classified as high were downgraded to medium, contributing to the increase in that category by 5 to 15 when compared to the neutral prompt. The cumulative total for this version was 29 vulnerabilities. These results suggest that while the secure prompt made modest improvements in its use of dependencies, it did not consistently address all vulnerabilities and also introduced new configuration risks.

2) DeepSeek: Overall, a comparison of the outputs from the two prompts shows that DeepSeek produced a more complex implementation of the Java-based web application than GPT-4. The applications generated by DeepSeek encompassed not only the configuration, controller, model, and repository classes, like with GPT-4, but also added additional files that enhanced functionality and structure. For example, the neutral prompt introduced the Data Transfer Object (DTO) and service classes for handling data manipulation and application logic.

After running the code through the OWASP security tools, the neutral prompt revealed 37 total vulnerabilities, including 3 Critical, 13 High, 10 Medium, and 11 Low. While these results were similar to those from the GPT-4 model, some notable exceptions warrant highlighting. One medium-level vulnerability was associated with version 3.1.2 of Spring Boot DevTools, as detected by OWASP Dependency Check (CVE-2023-34055). Additionally, including the StorageService class by DeepSeek introduced another medium-level Potential Path Traversal vulnerability, in addition to what was already detected by SpotBugs with GPT-4 (CWE-22).

Looking into the secure prompt for DeepSeek reveals a greater degree of complexity than even its neutral variant. The changes included implementation classes for ProductService and UserService, which were separated from their interfaces to better align with the separation of concerns. A method call was also identified within the UserController that passed a null value to a non-null parameter during the user account registration process. This vulnerability, which involves input validation and error handling, is classified as CWE-476 in the public database. In addition, the secure prompt analysis flagged additional outdated dependencies, including version 1.70 of the Bouncy Castle Provider, which is used for cryptographic implementations. This issue was linked to five CVEs with low confidence. The secure prompt ultimately led to more vulnerabilities across all severity levels, highlighting the unintentional consequences that can arise from overengineering a secure prompt. Although there was one less critical vulnerability than in the neutral version, testing revealed four additional medium-level vulnerabilities and three new low-level vulnerabilities. The cumulative total for this version was 43 vulnerabilities.

3) Llama 4: Llama 4 exhibited less effective performance under both prompt conditions, showing minimal benefit from the introduction of secure coding practices. In both the neutral and secure variants, the model produced approximately 10 critical-rated vulnerabilities, the highest among the three LLMs tested. No measurable improvements were observed between prompts, with the secure version yielding an equal or greater quantity of vulnerabilities. These additional risks were mainly identified by OWASP Dependency Check. As shown in Fig. 3, the version of Tomcat Embed Core used for each model and prompt is graphed based on how far it lags behind the latest release available during testing in April 2025. Both outputs for Llama 4 included Tomcat Embed Core version 9.0.65, released in July 2022, in their pom.xml files. Lagging by over 2.5 years, this model's output is a relative outlier because it uses outdated

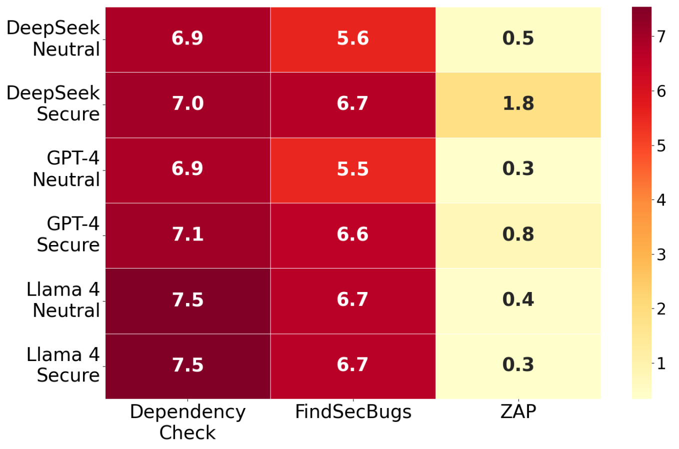

Fig. 4. Heatmap matrix showing average CVSS scores by OWASP tool across different LLMs and prompt types. Higher scores, visualized by darker colors, indicate more severe vulnerabilities.

components. This reliance reduces the application's resilience to withstand modern exploits and cyberattacks.

B. Overall Assessment

The results across all three models suggest that while using more secure prompts can influence improvements, such as using newer versions of dependencies and reducing the number of vulnerabilities, their impact is largely inconsistent and model-dependent. For example, GPT-4 responded most effectively to secure prompting, showing notable reductions in high-severity vulnerabilities. DeepSeek took a more detailed approach to its structure and implementation, but did not implement any consistent security improvements. And lastly, Llama 4 displayed no measurable improvements, continuing to use severely outdated components such as Tomcat Emded Core from mid-2022. The most critical issues across all models were identified by OWASP Dependency Check, highlighting common vulnerabilities in the versions of third-party libraries in use. Additionally, the secure prompts exhibited some natural variance in the generation process, including misconfigurations introduced from the added complexity (e.g., disabled CSRF protection). Fig. 4 presents a heatmap showing average CVSS scores by OWASP tool across different LLMs and prompt types.

C. NIST Risk Analysis

Before any risk could be scored using the NIST Risk Management Framework, it was necessary to normalize and enrich the raw vulnerability data generated from the OWASP security tools. Results from SpotBugs, FindSecBugs, OWASP Dependency Check, and OWASP ZAP were exported in HTML or XML, depending on the specific tool and configuration used. These saved reports contained valuable information for risk evaluation purposes, but required additional steps to easily extract identifiers such as CWEs and CVEs (see Fig. 5).

Custom Python scripts were used to parse for this type of information in each of the output files. For example, the XML-formatted reports from both SpotBugs and ZAP listed vulnerabilities with any CWEs associated with code smells

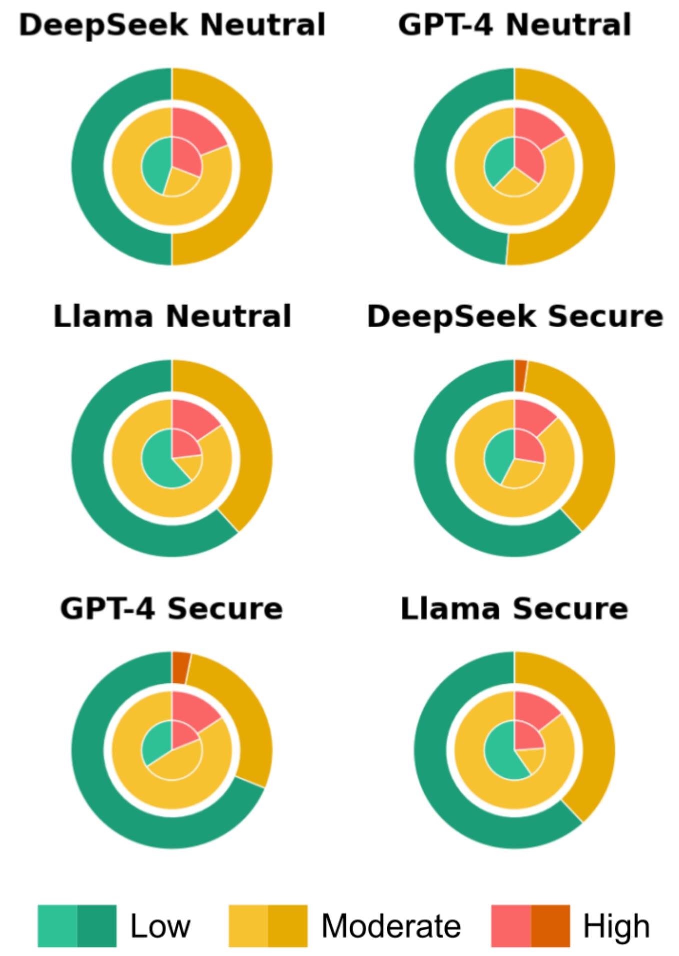

Fig. 5. Concentric pie charts visualizing the distribution of likelihood (inner ring), impact (middle ring), and overall risk (outer ring) for each LLM under neutral and secure prompt conditions. Severity levels are color-coded as Low (green), Moderate (amber), and High (red).

or dynamic flaws. Dependency Check, on the other hand, linked its findings directly to CVEs for a specific outdated and vulnerable component. The format and structure of the outputs generated by these tools served as the foundation for mapping findings to standardized definitions of known security issues.

V. Conclusion

In this study, we have examined LLM-generated code against modern security standards to better understand its resiliency to malicious attacks. We used the established OWASP Top Ten framework to categorize attack types, and the NIST Risk Management Framework to evaluate the risks associated with identified vulnerabilities. Using tools such as SpotBugs with FindSecBugs, OWASP Dependency Check, and OWASP ZAP, we conducted static analysis, third-party dependency scanning, and dynamic testing. Across the various testing configurations, all outputs, collected in either HTML or XML formats, were passed through the NIST Risk Management Framework. This necessary step standardized the findings and aligned each identified weakness, as outlined by Common Weakness Enumerations (CWEs), with Common Vulnerabilities and Exposures (CVEs) from the National Vulnerability Database. By linking the identified weaknesses to known vulnerabilities already published, we have established a quantifiable basis for assessing security in applications influenced by large language models.

A. Research Limitations

Several limitations of this study should be acknowledged. First, all experimental results were obtained from testing a single application type, a Java-based e-commerce web application. This constrains how any conclusions about prompt-driven security improvements can be generalized. It also introduces a potential bias not only from the application type but also from the programming language used, as models may generate code differently to satisfy project requirements, which could inadvertently alter the output's security profile. Projects that vary in codebase size also pose a concern, as vulnerability counts in this study are reported in absolute terms without normalization. Overall, future work of this study should investigate how differences in programming languages, normalized project sizes, and application types affect a model's adherence to security-emphasized prompting.

Second, each prompt-model combination was evaluated using a single generated output. LLMs are inherently stochastic, meaning that repeated generations from the same prompt can yield different solutions, varying in how the code is structured, implemented, and supplemented with security requirements. Reproducibility was further hindered by reliance on human intervention during testing, including manual code migration into the development environment and error resolution. Without automation, replication across runs in the testing process cannot be guaranteed, as human intervention can influence the outcomes.

Third, this study used only two prompt conditions: a neutral prompt to establish the project requirements and a security-emphasized prompt incorporating OWASP best practices for secure code generation. While this binary design simplifies the study and allows for a controlled comparison, it does not capture the full range of outcomes that could be observed across a gradient of security emphasis. Exploring these variables within prompt design would provide valuable insights beyond the current state of this research.

B. Future Work

Experimental results were obtained using the latest version of the LLMs at the time of the study. This field of AI is highly dynamic and continuously improving. A future study can be designed to test the performance of newer versions to observe their improvements, specifically in resisting malicious attacks.

Also, the study can be designed to implement iterative code enhancements to exploit the refined reasoning ability of LLMs. Applications generated by LLMs would undergo an automated evaluation process using the OWASP tools and the NIST Risk Management Framework, with the results informing any necessary changes in subsequent versions of the application. This loop would involve multiple cycles of code generation, vulnerability scanning, and risk assessment, with each cycle feeding back into the prompt or model configuration to produce a more secure version. Over multiple iterations, LLMs could learn to avoid common vulnerabilities and generate code with a more secure profile. Such an implementation would not only automate vulnerability detection but also enable continuous security improvement, paving the way for LLM-driven solutions to create software that is production-ready and compliant with established policies.

In this rapidly evolving field of study, this work, despite its shortcomings, opens doors and guides research in different directions.

References

[1] "Software Vulnerability Snapshot 2024" Report by Black Duck, Retrieved from http://www.blackduck.com

[2] Deepstrike (2025, Oct 08), "Vulnerabilities Statistics 2025: Record CVEs, Zero-Days & Exploits", Retrieved from https://deepstrike.io/blog/vulnerability-statistics-2025

[3] S. Singh, (2025, May 19) ChatGPT Statistics 2025 – DAU & MAU Data (Worldwide) https://www.demandsage.com/chatgpt-statistics/ Accessed 22 May 2025

[4] ChatGPT, GPT-4, OpenAI, chat.openai.com, Accessed 15 June 2025

[5] DeepSeek, https://www.deepseek.com/, Accessed 15 June 2025

[6] Llama-4, https://www.llama.com/models/llama-4/, Accessed 15 June 2025

[7] Stack Overflow. (2024). *2024 Developer Survey: AI and developer tools*. https://survey.stackoverflow.co/2024/ai Accessed 06 June 2025

[8] Perry, Neil, et al. "Do users write more insecure code with AI assistants?" Proceedings of the 2023 ACM SIGSAC conference on computer and communications security. 2023.

[9] Sarkar, Advait, et al. "What is it like to program with artificial intelligence?." arXiv preprint arXiv:2208.06213 (2022).

[10] Zhu, Xiaogang, et al. "When software security meets large language models: A survey." IEEE/CAA Journal of Automatica Sinica 12.2 (2025): 317-334.

[11] OWASP Foundation. (2021). OWASP Top 10: 2021. https://owasp.org/www-project-top-ten/2021

[12] National Institute of Standards and Technology. (2018). Risk Management Framework for Information Systems and Organizations: A System Life Cycle Approach for Security and Privacy. U.S. Department of Commerce, Washington, D.C., NIST Special Publication (NIST SP) 800-37r2. https://doi.org/10.6028/NIST.SP.800-37r2

[13] Fan, Angela, et al. "Large language models for software engineering: Survey and open problems." 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE). IEEE, 2023.

[14] Kotsiantis, Sotiris, Vassilios Verykios, and Manolis Tzagarakis. "Ai-assisted programming tasks using code embeddings and transformers." Electronics 13.4 (2024): 767.

[15] Liang, Jenny T., Chenyang Yang, and Brad A. Myers. "A large-scale survey on the usability of ai programming assistants: Successes and challenges." Proceedings of the 46th IEEE/ACM international conference on software engineering. 2024.

[16] Xu, HanXiang, et al. "Large language models for cyber security: A systematic literature review." arXiv preprint arXiv:2405.04760 (2024).

[17] Ramírez, Leonardo Criollo, et al. "State of the Art of the Security of Code Generated by LLMs: A Systematic Literature Review." 2024 12th International Conference in Software Engineering Research and Innovation (CONISOFT). IEEE, 2024.

[18] Dincă, Ana-Maria, et al. "AI Tools introduced in Software Development. Analysis of Code quality, Security and Productivity Implications." 2024 IEEE 30th International Symposium for Design and Technology in Electronic Packaging (SIITME). IEEE, 2024.

[19] Baralla, Gavina, Giacomo Ibba, and Roberto Tonelli. "Assessing GitHub Copilot in Solidity Development: Capabilities, Testing, and Bug Fixing." IEEE Access (2024).

[20] Feng, Yunhe, et al. "Investigating code generation performance of ChatGPT with crowdsourcing social data." 2023 IEEE 47th Annual Computers, Software, and Applications Conference (COMPSAC). IEEE, 2023.

[21] Grewal, Balreet, et al. “Analyzing Developer Use of ChatGPT Generated Code in Open Source GitHub Projects.” Proceedings of the 21st International Conference on Mining Software Repositories. 2024.

[22] Klemmer, Jan H., et al. “Using ai assistants in software development: A qualitative study on security practices and concerns.” Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. 2024.

[23] OpenAI, “GPT-4 Technical report,” 2023, arXiv:2303.08774.

[24] Yetiştiren, Burak, et al. “Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt.” arXiv preprint arXiv:2304.10778 (2023).

[25] Borji, Ali. “A categorical archive of chatgpt failures.” arXiv preprint arXiv:2302.03494 (2023).

[26] Torka, Simon, and Sahin Albayrak. “Optimizing AI-Assisted Code Generation.” arXiv preprint arXiv:2412.10953 (2024).

[27] Pearce, Hammond, et al. “Asleep at the keyboard? assessing the security of github copilot’s code contributions.” Communications of the ACM 68.2 (2025): 96-105.

[28] He, Jingxuan, and Martin Vechev. “Controlling large language models to generate secure and vulnerable code.” arXiv preprint arXiv:2302.05319 (2023).

[29] Taeb, Maryam, Hongmei Chi, and Shonda Bernadin. “Assessing the effectiveness and security implications of ai code generators.” Journal of The Colloquium for Information Systems Security Education. Vol. 11. No. 1. 2024.

[30] Berabi, Berkay, et al. “DeepCode AI Fix: Fixing Security Vulnerabilities with Large Language Models. CoRR abs/2402.13291 (2024), 1–12.” arXiv preprint arXiv:2402.13291 (2024).

[31] Improta, Cristina. “Poisoning programs by un-repairing code: Security concerns of ai-generated code.” 2023 IEEE 34th International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2023.

[32] Fredj, Ouissem Ben, et al. “An OWASP top ten driven survey on web application protection methods.” Risks and Security of Internet and Systems: 15th International Conference, CRiSIS 2020, Paris, France, November 4–6, 2020, Revised Selected Papers 15. Springer International Publishing, 2021.

[33] Higuera, Juan R. Bermejo, et al. “Benchmarking Approach to Compare Web Applications Static Analysis Tools Detecting OWASP Top Ten Security Vulnerabilities.” Computers, Materials & Continua 64.3 (2020).

[34] Mateo Tudela, Francesc, et al. “On combining static, dynamic and interactive analysis security testing tools to improve owasp top ten security vulnerability detection in web applications.” Applied Sciences 10.24 (2020): 9119.

[35] Spotbugs Documentation Release 4.0.0-RC2 by spotbugs community, https://spotbugs.readthedocs.io/downloads/en/issue-1075/pdf/ Accessed 04 June 2025

[36] OWASP Foundation. (2024). Dependency-Check. Retrieved from https://owasp.org/www-project-dependency-check/

[37] OWASP Foundation. (2022). OWASP Zed Attack Proxy (ZAP) (Version 2.11.1) [Computer software]. Available from https://www.zaproxy.org/.