International Journal of Research in Engineering and Science (IJRES)
ISSN (Online): 2320-9364, ISSN (Print): 2320-9356
www.ijres.org Volume 14 Issue 8 ǁ August 2026 ǁ PP. 102-121

Comparative Analysis of Security Vulnerabilities in AI-Assisted and Traditional Software Applications Using Ethical Hacking Techniques

Charles Denver Ean Torres, Arabela A. Barbon, Jose Genelido G. Cabalquinto Jr., Jegie B. Mahusay, Francis D. Francisco, Serafin C. Palmares, Kristine T. Soberano
State University of Northern Negros, Philippines

Abstract
This study compared the security posture of AI-assisted and traditionally developed web applications using internationally recognized software security assessment frameworks. Two web applications implementing identical functional requirements were developed using different software development approaches and evaluated under identical testing conditions. Dynamic security evaluation was performed using OWASP Zed Attack Proxy (OWASP ZAP), whereas static source code analysis was conducted with SonarQube Community Edition. All detected weaknesses were categorized based on the OWASP Top 10 (2021), mapped to the corresponding Common Weakness Enumeration (CWE), and evaluated using the Common Vulnerability Scoring System (CVSS) Version 3.1. The assessment recorded 33 OWASP ZAP alerts for the AI-assisted application and 26 alerts for the traditionally developed application. Both applications contained one critical vulnerability associated with an outdated third-party JavaScript library. However, the AI-assisted application exhibited a greater number of medium-severity findings, particularly those involving security misconfiguration, Cross-Origin Resource Sharing (CORS), Content Security Policy (CSP), and the absence of anti-CSRF protection. Static analysis produced comparable results, identifying four security issues in the AI-assisted application, which received a Security Rating of E, while the traditionally developed application contained two security issues and obtained a Security Rating of B. Despite the differences in security ratings, both applications demonstrated similar levels of reliability and maintainability. The findings indicate that AI-assisted software development can produce functional web applications with performance characteristics comparable to those developed through conventional programming practices. Nevertheless, the results also show that AI-generated code may introduce additional security weaknesses when it is incorporated without comprehensive review and verification. These observations emphasize the importance of integrating both static and dynamic security assessments throughout the software development lifecycle, irrespective of the development approach. By combining internationally recognized cybersecurity assessment frameworks in a single comparative evaluation, this study provides quantitative evidence of differences in the security posture of AI-assisted and traditionally developed web applications and reinforces the need for systematic security validation before AI-generated code is deployed in production environments.

Keywords*: Artificial Intelligence, Ethical Hacking, OWASP ZAP, Software Security, Vulnerability Assessment.*

Date of Submission: 05-08-2026 Date of acceptance: 17-08-2026

I. INTRODUCTION

Recent advances in artificial intelligence have transformed software development by enabling tools that assist programmers in writing source code, automating repetitive tasks, and accelerating application development. Large language models and AI-powered code completion systems enable developers to produce functional software more efficiently while reducing development time (Noy & Zhang, 2023). As these tools become more widely adopted, AI-assisted programming is increasingly being incorporated into software engineering workflows. Although these technologies offer significant productivity benefits, they also introduce security concerns. AI-generated code may contain insecure programming practices, inadequate input validation, security misconfigurations, or vulnerable third-party dependencies when generated outputs are adopted without careful review. Such weaknesses may increase the likelihood of security vulnerabilities that compromise application integrity and user data. These concerns highlight the need to evaluate whether AI-assisted software provides a level of security comparable to software developed using traditional programming approaches.

Since AI-generated outputs may contain hidden weaknesses, conducting a proper security assessment is necessary to determine whether AI-assisted applications provide comparable protection to software developed using traditional programming approaches. Therefore, security testing and validation should be performed before

deploying AI-generated software to ensure that possible vulnerabilities are identified and addressed (Open Worldwide Application Security Project [OWASP], 2024).

The security of web applications remains a critical issue because these systems are commonly used to handle valuable personal and organizational data through online services. OWASP (2021) identified several recurring security risks that continue to affect web applications, including access control problems, injection vulnerabilities, insecure configurations, authentication weaknesses, and outdated software dependencies. These issues may create opportunities for attackers to compromise application systems, access restricted information, interrupt service availability, or affect the overall performance and functionality of the software.

Application security is most effective when security practices are incorporated throughout the software development lifecycle instead of being performed only after software development is complete. According to the National Institute of Standards and Technology (NIST, 2022), integrating security evaluation into multiple development phases enables developers to identify and address vulnerabilities before they become more difficult and costly to resolve. This approach applies to both traditionally developed software and applications created with AI-assisted programming tools. To achieve a comprehensive assessment, software security is commonly evaluated using Dynamic Application Security Testing (DAST) and Static Application Security Testing (SAST). Although both methods aim to identify security weaknesses, they examine different aspects of an application. DAST assesses a running application by observing its behavior during execution and identifying vulnerabilities that may be exploited in a live environment. In contrast, SAST analyzes the application's source code to detect insecure programming practices and implementation flaws before deployment. This study employed OWASP Zed Attack Proxy (OWASP ZAP) to perform dynamic security testing and SonarQube to conduct static source code analysis. OWASP ZAP identifies vulnerabilities that become evident while an application is operating, whereas SonarQube evaluates the source code for security weaknesses, code quality issues, reliability concerns, and maintainability problems (OWASP, 2024; SonarSource, 2024). Using these complementary testing approaches provides a more complete evaluation of application security by examining both runtime behavior and the underlying implementation of the software.

To standardize vulnerability analysis and facilitate consistent interpretation of security findings, cybersecurity researchers and practitioners use internationally recognized classification and assessment frameworks. Several established cybersecurity frameworks are used to organize and evaluate software security issues. The application of these security frameworks enables researchers and practitioners to organize findings consistently, evaluate vulnerability risks, and compare security issues using recognized industry standards. While previous research has explored topics related to AI-generated code, AI-supported software development, and secure programming practices, limited studies have focused on directly examining the differences in security characteristics between AI-assisted and traditionally developed web applications using similar testing environments.

Furthermore, there is still a need for experimental studies that assess applications built with identical functional requirements while utilizing various security testing approaches and established vulnerability classification models. Investigating this research gap is significant because the method used to develop software may affect the security weaknesses that appear within an application. A comparative analysis between AI-assisted and conventional development approaches can provide measurable evidence regarding the security implications of using AI-generated code rather than relying only on assumptions about its potential risks.

To address the identified research gap, this study performs a comparative evaluation of the security characteristics of two web applications developed through different approaches: AI-assisted development and conventional software development. Both applications were designed with the same functional requirements and examined using the same testing environment to ensure that the comparison was conducted fairly. This study evaluated application security through two complementary testing techniques. Dynamic Application Security Testing (DAST) was conducted using OWASP Zed Attack Proxy (OWASP ZAP), while Static Application Security Testing (SAST) was carried out with SonarQube Community Edition. Security findings from both tools were interpreted using the OWASP Top 10 (2021), mapped to the appropriate Common Weakness Enumeration (CWE) categories, and assessed according to the Common Vulnerability Scoring System (CVSS) Version 3.1. Integrating these assessment methods enabled the identification of security weaknesses from both the application's runtime behavior and its underlying source code.

The findings of this study provide practical insights for software developers, researchers, educators, and organizations adopting AI-assisted development practices. In particular, the results demonstrate that AI-generated code should undergo thorough review, validation, and security testing before deployment rather than being accepted solely because it produces the expected functionality. The study also reinforces the importance of incorporating secure coding practices and comprehensive security assessment throughout the software development lifecycle to reduce potential risks associated with AI-assisted software development. In academic

settings, the findings may help educators emphasize the role of secure programming practices, code evaluation, and security testing when teaching software development. Furthermore, researchers may use the results as a foundation for future studies exploring the security effects of AI-supported programming practices.

The main contribution of this research is the systematic comparison between AI-assisted and traditionally developed applications using a controlled experimental setup. Unlike studies that only examine AI-generated code independently, this research uses a traditionally developed application with the same requirements as a baseline for comparison. Both applications undergo similar security assessments using recognized cybersecurity frameworks and testing methodologies, allowing differences in their security conditions to be measured objectively. The study aims to provide a clearer understanding of how AI-assisted development may influence software security and highlights the importance of continuous security evaluation throughout the software development process, regardless of the development method used. The following section discusses the methodology applied in creating, testing, classifying, and comparing the security findings of the two applications.

Statement of the Problem

This study aimed to compare the security posture of AI-assisted and traditionally developed web applications using standardized cybersecurity assessment frameworks.

Specifically, it sought to answer the following questions:

1. What security vulnerabilities are identified in the traditionally developed and AI-assisted web applications using OWASP ZAP?
2. What software security issues, reliability issues, and maintainability issues are identified using SonarQube static code analysis?
3. How are the identified vulnerabilities classified according to:
    - OWASP Top 10 (2021);
    - Common Weakness Enumeration (CWE); and
    - Common Vulnerability Scoring System (CVSS) Version 3.1?
4. Is there a difference in the overall security posture of the traditionally developed and AI-assisted web applications based on the results of OWASP ZAP, SonarQube, OWASP Top 10, CWE, and CVSS Version 3.1?

Objectives of the Study

General Objective

To compare the security posture of AI-assisted and traditionally developed web applications using recognized software security assessment frameworks and standards.

Specific Objectives

Specifically, this study aims to:

1. To examine the security weaknesses identified during the Dynamic Application Security Testing (DAST) process using OWASP ZAP;
2. To evaluate source code quality and identify potential security weaknesses through Static Application Security Testing (SAST) using SonarQube.
3. To classify and categorize identified vulnerabilities using recognized security frameworks, specifically the OWASP Top 10 (2021) and Common Weakness Enumeration (CWE).
4. To assess the risk level and potential impact of the discovered vulnerabilities through the Common Vulnerability Scoring System (CVSS) Version 3.1; and
5. To analyze the differences in security characteristics between AI-assisted and traditionally developed web applications based on the assessment results generated from the selected security tools and evaluation frameworks.

Significance of the Study

The findings of this study may benefit the following:

Software Developers. The findings may help developers understand the security differences between AI-assisted and traditional software development practices.

Researchers. This research may serve as a source of empirical information for future studies related to AI-assisted software development, application security, and cybersecurity assessment methodologies. The findings may contribute to further investigation of the security challenges and opportunities associated with AI-supported programming.

Higher Education Institutions. The outcomes of this study may assist academic institutions in strengthening software engineering and cybersecurity courses by incorporating secure development practices, vulnerability assessment techniques, and security testing concepts into their curriculum.

Students. This study may provide students with practical exposure to the application of recognized cybersecurity tools and frameworks, including OWASP ZAP, SonarQube, OWASP Top 10, CWE, and CVSS, for assessing web application security.

Scope and Delimitation of the Study

This research investigates the security characteristics of two web applications developed through different approaches: traditional software development and AI-assisted software development. Both applications were created based on the same functional requirements and assessed using the same testing conditions to establish a consistent basis for comparison.The evaluation involved automated security assessment techniques consisting of Dynamic Application Security Testing (DAST) through OWASP ZAP Version 2.17.0 and Static Application Security Testing (SAST) using SonarQube Community Edition. Security findings obtained from these tools were analyzed and organized using recognized cybersecurity frameworks, including the OWASP Top 10 (2021), Common Weakness Enumeration (CWE), and Common Vulnerability Scoring System (CVSS) Version 3.1.

The study only considered automated security testing results generated through DAST and SAST approaches. Activities such as manual penetration testing, extended source code inspection beyond SonarQube analysis, exploit creation, fuzzing activities, reverse engineering, performance testing, usability assessment, and continuous monitoring after deployment were not included within the research scope. Furthermore, CVE references were considered only for vulnerabilities with officially assigned identifiers related to third-party software components. Security issues involving configuration problems or implementation weaknesses without available CVE records were assessed using researcher-calculated CVSS Base Scores following the CVSS Version 3.1 methodology.

The conclusions drawn from this research are based only on the two applications evaluated within the defined testing environment. Therefore, the findings should be interpreted specifically within the context of the applications examined and should not be generalized to all AI-assisted development platforms or software development approaches.

II. REVIEW OF RELATED LITERATURE

The integration of artificial intelligence (AI) into software development has introduced new opportunities for improving programming efficiency; however, it has also raised concerns regarding the security and reliability of AI-generated source code. Existing research has investigated whether AI-assisted programming tools can produce secure software outputs and how these tools influence the overall software development process.

Beyond examining generated code itself, researchers have also studied how AI-assisted programming affects developer practices. Perry et al. (2023) conducted a large-scale study to determine the influence of AI coding assistants on developers' security behavior. Their findings indicated that developers using AI assistance produced code with weaker security characteristics compared with those who relied solely on their own programming process. Nevertheless, the researchers observed that developers who carefully reviewed and modified AI-generated suggestions were able to reduce security issues, emphasizing that human judgment remains an important factor in AI-supported development.

The security performance of AI coding assistants was further evaluated by Majdinasab et al. (2023) through an updated analysis of GitHub Copilot using CodeQL. Their study identified improvements in the quality of generated code, as insecure suggestions decreased from 36.54% to 27.25%. Despite this improvement, security vulnerabilities continued to appear, indicating that enhancements in AI programming tools do not eliminate the risks associated with AI-generated code.

Cotroneo et al. (2025) extended the investigation of AI-generated software by comparing large-scale code samples written by AI systems and human developers. Their analysis covered more than 500,000 Python and Java code samples and examined differences in security risks and software characteristics. The study reported that AI-generated code was more frequently associated with high-risk vulnerabilities, while human-developed code showed different patterns related to code structure and maintainability. These findings suggest that the development approach used in creating software may influence both security outcomes and overall software quality.

Apart from studies focused on AI-generated code, previous research has also explored the effectiveness of automated tools for detecting software vulnerabilities. Jarupunphol et al. (2023) compared Burp Suite and OWASP ZAP while assessing a university web application and found that the two tools produced different results in terms of vulnerability detection, severity classification, confidence ratings, and OWASP Top 10 categorization.

The study shows that automated security testing solutions can provide valuable support in discovering and evaluating security weaknesses in web applications.

Overall, previous studies demonstrate that AI-assisted programming can enhance software development efficiency while also introducing security concerns that require further investigation. Existing research has mainly focused on specific areas, including individual AI-generated code outputs, developers' interaction with AI programming tools, and separate vulnerability assessment methods. These limitations indicate the need for more comprehensive comparative research that evaluates the security performance of complete applications developed through both AI-assisted and traditional approaches using multiple security assessment techniques. However, limited research has focused on comparing complete web applications developed through AI-assisted and traditional approaches while maintaining equivalent requirements and controlled testing conditions. This limitation highlights the need for further investigation, particularly through the combined use of dynamic and static security assessment methods supported by recognized cybersecurity frameworks.

III. MATERIALS AND METHODS

This research used a quantitative comparative experimental approach to examine the security differences between web applications developed through AI-assisted and conventional programming methods. Two applications were created based on the same set of functional requirements, with the primary difference being the development approach used in producing each system. To ensure a reliable comparison, both applications were deployed within the same environment and evaluated using equivalent security testing procedures and configurations.

The security evaluation data for both applications were obtained using automated testing techniques, specifically Dynamic Application Security Testing (DAST) through OWASP ZAP and Static Application Security Testing (SAST) through SonarQube. Findings from these assessments were further examined using recognized cybersecurity frameworks, where vulnerabilities were classified using the OWASP Top 10 (2021), mapped to Common Weakness Enumeration (CWE), and assigned severity levels based on the Common Vulnerability Scoring System (CVSS) Version 3.1. The comparison between the two applications considered differences in vulnerability occurrence, security classifications, and software quality indicators. Metrics such as the number of detected issues, security ratings, vulnerability categories, and CVSS scores were used to measure and compare their overall security performance.

Research Workflow of the Study

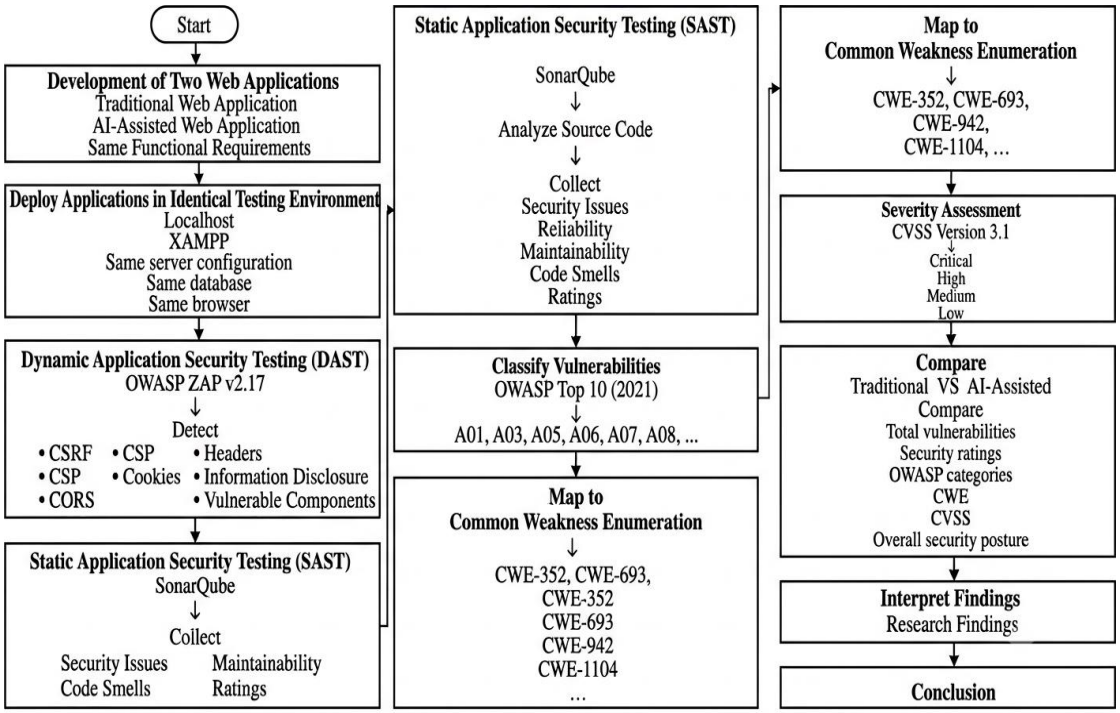


Figure 3.1. Research Workflow of the Comparative Security Assessment

OWASP ZAP Dynamic Application Security Testing

The web applications were tested using Dynamic Application Security Testing (DAST) with OWASP Zed Attack Proxy (OWASP ZAP) Version 2.17.0. Developed by the Open Worldwide Application Security Project (OWASP), OWASP ZAP is an open-source tool used to identify security vulnerabilities in web applications while they are running. In this study, the tool was used to examine application traffic, including HTTP requests and responses, cookies, headers, forms, and client-side resources, to detect possible security issues (OWASP Foundation, 2024).

For the assessment process, both the traditionally developed and AI-assisted applications were configured and executed within the same local deployment environment. Each application underwent a separate automated security scan using OWASP ZAP while maintaining consistent server settings, scanning configurations, and evaluation procedures. This approach ensured that the vulnerability results from both applications could be compared using the same testing conditions.

The automated scan examined the applications for common web security vulnerabilities, including but not limited to:

- Cross-Site Request Forgery (CSRF)
- Content Security Policy (CSP) Misconfigurations
- Cross-Origin Resource Sharing (CORS) Misconfigurations
- Missing Security Headers
- Cookie Security Weaknesses
- Information Disclosure
- Outdated Third-Party Components
- HTTPS Configuration Issues

SonarQube Static Code Analysis

Static Application Security Testing (SAST) through SonarQube Community Edition was used to assess the source code of both web applications. The same SonarQube Quality Profile and scanning configurations were applied to the AI-assisted and traditionally developed applications to maintain a fair and consistent comparison. Using identical analysis settings enabled the security findings from both applications to be evaluated under the same conditions.

The following software quality indicators were collected:

- Security Issues
- Reliability Issues
- Maintainability Issues
- Security Rating
- Reliability Rating
- Maintainability Rating
- Code Duplication
- Test Coverage.

The resulting metrics were compared to determine whether AI-assisted development introduced a greater or lesser number of static security weaknesses than traditional software development.

Vulnerability Classification using OWASP Top 10 (2021)

Each vulnerability identified by OWASP ZAP and SonarQube was classified according to the OWASP Top 10 (2021), which categorizes the most critical web application security risks based on industry consensus and real-world vulnerability data (OWASP Foundation, 2021).

The identified vulnerabilities were mapped to their corresponding OWASP categories whenever applicable, including:

OWASP Top 10 Category	**Description**
A01:2021 – Broken Access Control	Weaknesses that allow unauthorized access to resources or functionality.
A03:2021 – Injection	Injection flaws such as SQL Injection and Cross-Site Scripting.
A05:2021 – Security Misconfiguration	Missing or improperly configured security settings, headers, and services.

A06:2021 – Vulnerable and Outdated Components	Use of software components containing known vulnerabilities.
A07:2021 – Identification and Authentication Failures	Weaknesses involving authentication mechanisms and session management.
A08:2021 – Software and Data Integrity Failures	Failures in software integrity verification and trusted updates.

The OWASP Top 10 classification enabled the researchers to compare the overall security posture of both applications based on internationally recognized security risk categories.

Vulnerability Classification using Common Weakness Enumeration (CWE)

To identify the underlying software weaknesses associated with each detected vulnerability, all findings were mapped to the Common Weakness Enumeration (CWE) maintained by the MITRE Corporation (MITRE, 2024).

Unlike CVEs, which identify specific publicly disclosed software vulnerabilities, CWEs describe the underlying classes of programming and configuration weaknesses responsible for security issues.

Examples of the mappings used in this study include:

Vulnerability	**CWE**
Missing Anti-CSRF Token	CWE-352
Missing HttpOnly Cookie	CWE-1004
Missing Secure Cookie	CWE-614
Missing SameSite Cookie	CWE-1275
Missing HSTS Header	CWE-319
Missing Content Security Policy	CWE-693
CORS Misconfiguration	CWE-942
Server Information Disclosure	CWE-200
Missing X-Content-Type-Options Header	CWE-693
Vulnerable Third-Party Library	CWE-1104

The use of CWE allowed the researchers to determine whether vulnerabilities identified in both applications originated from similar classes of software weaknesses.

Vulnerability Severity Assessment using CVSS v3.1

The framework was applied to assign numerical severity values to identified vulnerabilities and provide a consistent basis for evaluating their potential impact.

This study utilized the CVSS Base Score to assess the inherent risk characteristics of each vulnerability. The Base Score was selected because it measures the fundamental severity of a vulnerability based on factors related to its exploitation potential and impact, without considering changes that may result from specific deployment environments or time-dependent conditions.

Each vulnerability was evaluated using the following Base Metrics:

- Attack Vector (AV)
- Attack Complexity (AC)
- Privileges Required (PR)
- User Interaction (UI)
- Scope (S)
- Confidentiality Impact (C)
- Integrity Impact (I)
- Availability Impact (A)

Official CVSS scores were adopted whenever a vulnerability corresponded to an officially published Common Vulnerabilities and Exposures (CVE) entry. For security misconfigurations and implementation

weaknesses without associated CVEs, researcher-derived CVSS Base Scores were assigned using the CVSS v3.1 Base Metric specification.

The resulting Base Scores were interpreted according to the standard CVSS severity ratings:

CVSS Score	**Severity**
0.0	None
0.1–3.9	Low
4.0–6.9	Medium
7.0–8.9	High
9.0–10.0	Critical

The CVSS assessment enabled objective comparison of the security impact of vulnerabilities detected in both applications.

Comparative Analysis

Following the completion of the dynamic and static security assessments, the results obtained from OWASP ZAP and SonarQube were consolidated for analysis.

The comparison between the traditional and AI-assisted applications considered the following evaluation criteria:

- Number of detected vulnerabilities
- Distribution of vulnerabilities according to OWASP Top 10 (2021)
- Classification of weaknesses using CWE
- Severity of vulnerabilities using CVSS v3.1
- SonarQube Security Rating
- SonarQube Reliability Rating
- SonarQube Maintainability Rating
- Overall security posture based on combined static and dynamic analysis

Descriptive statistics, including frequency counts and comparative tables, were used to summarize the findings. The results were analyzed to determine whether the AI-assisted application demonstrated a security posture comparable to or different from that of the traditionally developed application.

The use of standardized evaluation frameworks ensured that both applications were assessed consistently, objectively, and in accordance with internationally recognized cybersecurity standards.

OWASP ZAP Dynamic Security Assessment

The two web applications were analyzed through OWASP ZAP Version 2.17.0 using the Dynamic Application Security Testing (DAST) approach. To ensure a reliable comparison, both applications underwent the same deployment setup and scanning configuration. The results presented in Table 4.1 show that the traditionally developed application recorded 26 detected security alerts, while the AI-assisted application produced a total of 33 alerts.

Although both applications were identified with one high-risk vulnerability, differences were observed in the number of medium-level and informational findings. The AI-assisted application generated a higher volume of these additional findings compared with the traditional application. Based on the automated DAST evaluation, the AI-assisted application demonstrated a higher number of security issues detected by OWASP ZAP under the defined testing conditions.

Figure 4.1
Overview of OWASP ZAP Scanning Report for the Traditional Application

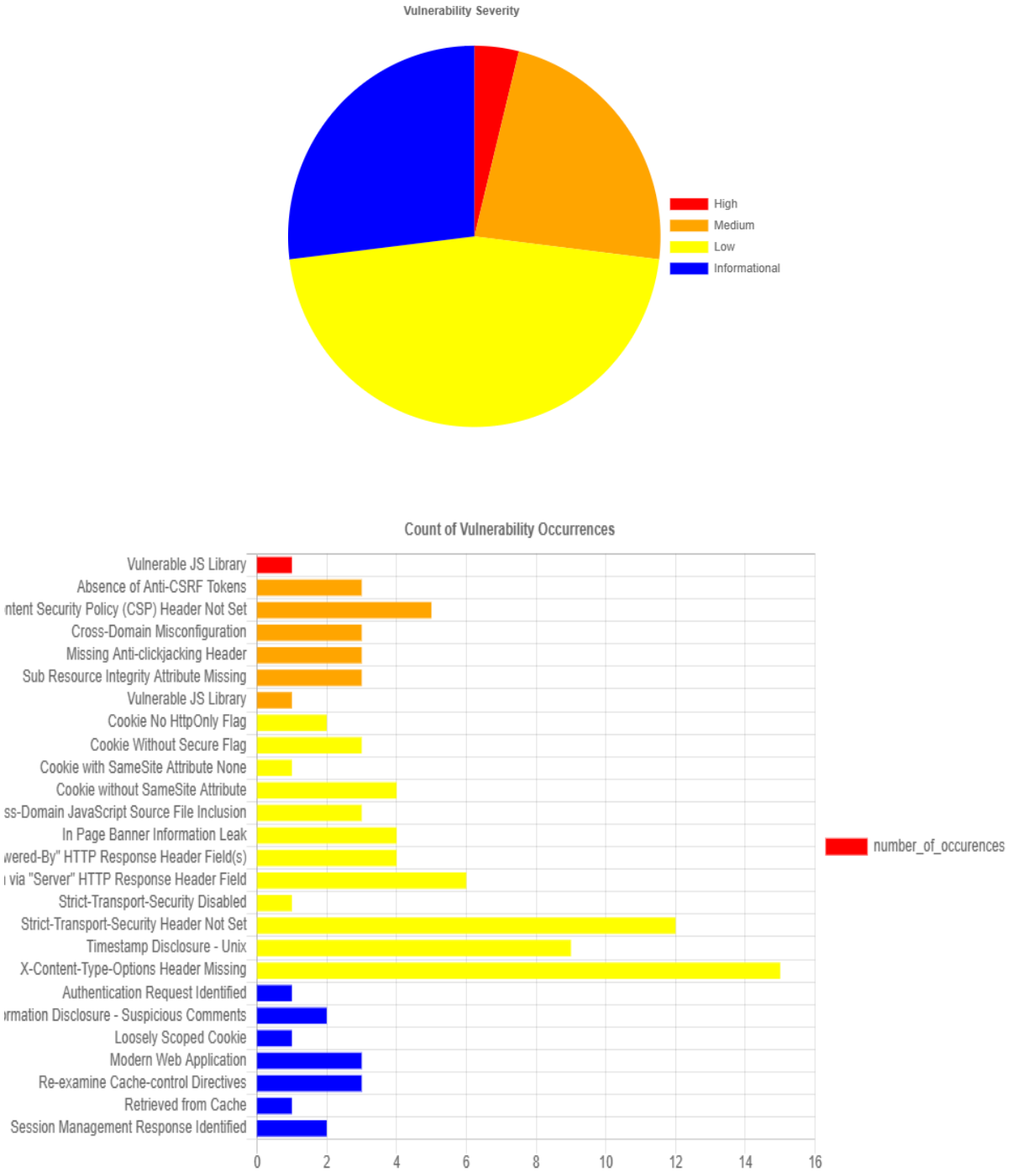


Figure 4.2
Overview of OWASP ZAP Scanning Report for the AI-Assisted Application

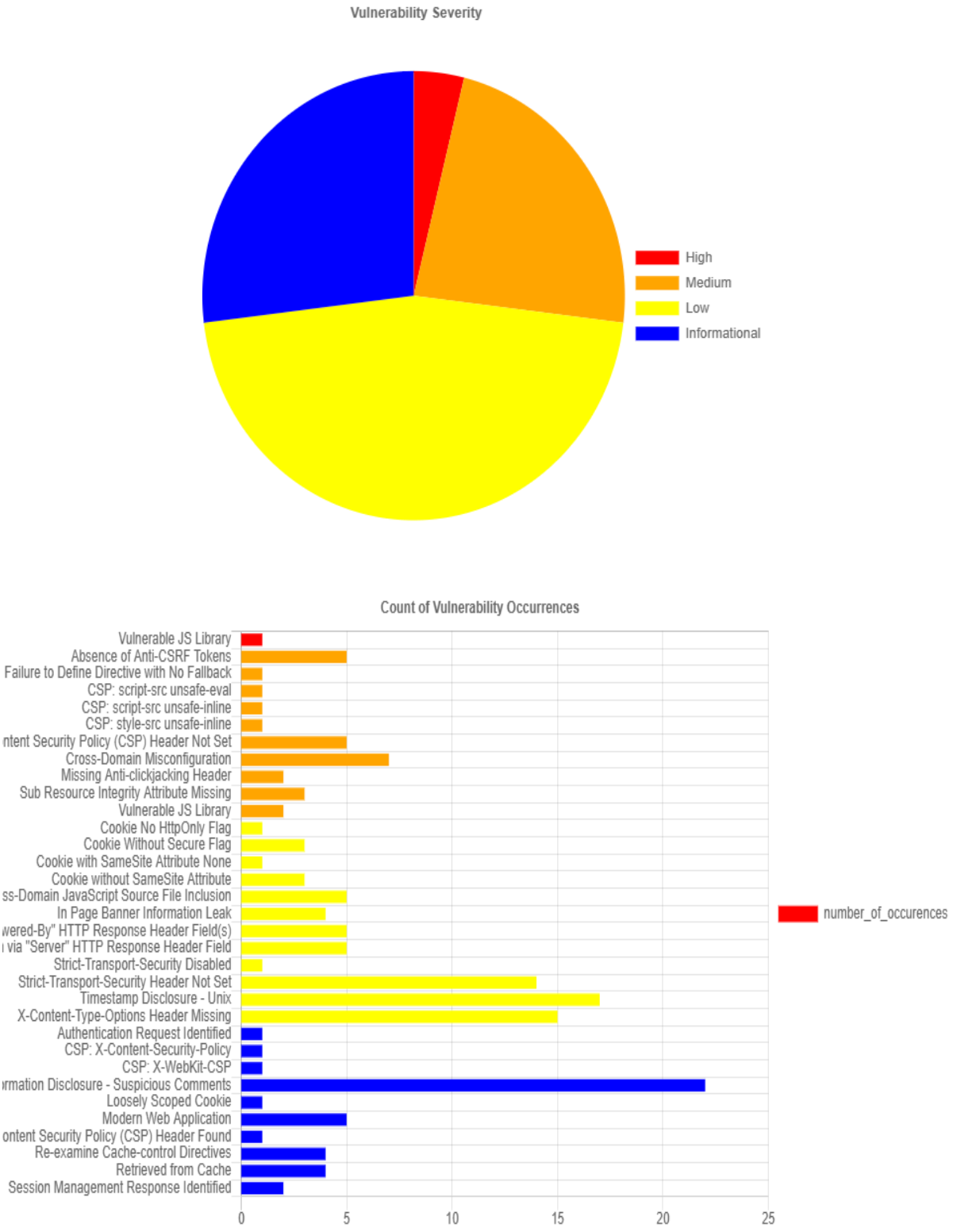


Table 4.1
Summary of OWASP ZAP Security Alerts

Risk Level	**Traditional Application**	**AI-Assisted Application**
High	1	1
Medium	6	10
Low	12	12
Informational	7	10
Total Alerts	**26**	**33**

Table 4.1 summarizes the overall security findings identified by OWASP ZAP. The traditional application generated 26 security alerts, while the AI-assisted application generated 33 security alerts. Although both applications contained one high-risk vulnerability, the AI-assisted application produced four additional medium-risk findings and three additional informational findings.

Medium-risk findings generally represent weaknesses that require remediation because they may contribute to successful attacks under favorable conditions, whereas informational findings usually indicate configuration issues or security observations that require further verification (OWASP Foundation, 2024).

These results suggest that the AI-assisted application contained a greater number of detectable security weaknesses than the traditionally developed application under identical scanning conditions. However, the difference primarily resulted from additional configuration-related findings rather than an increase in critical exploitable vulnerabilities.

Figure 4.3

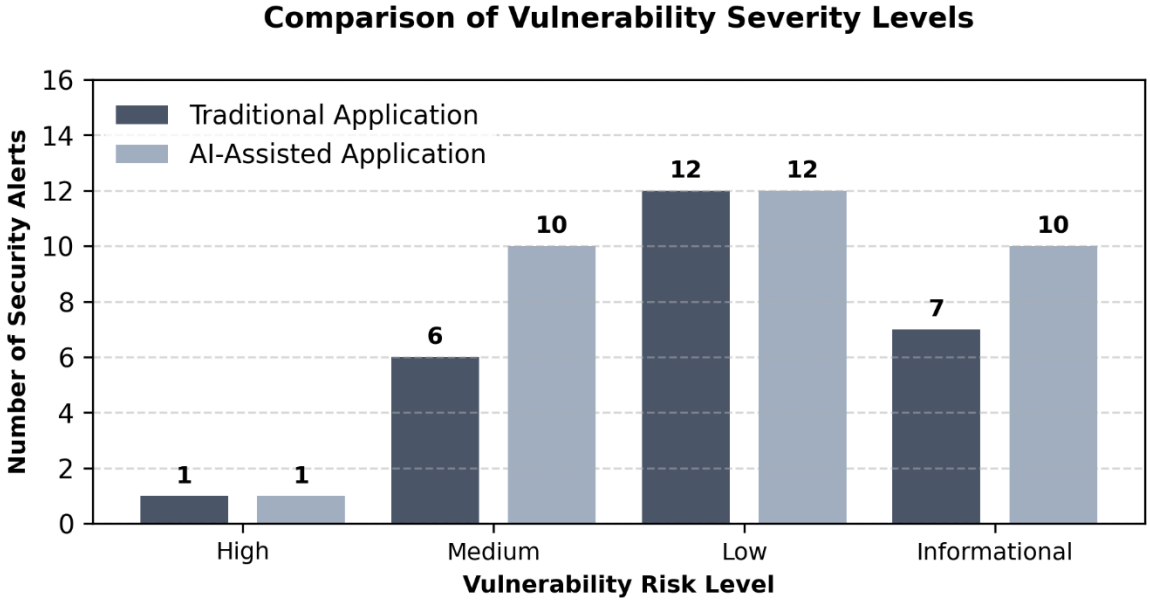


Figure 4.3 provides a visual comparison of the vulnerability severity distributions between the traditional and AI-assisted web applications.

As illustrated in the chart, there is a clear parity at the extremes of the severity spectrum. Both applications recorded exactly one high-risk vulnerability and twelve low-risk vulnerabilities. This visual alignment suggests that relying on AI assistance did not introduce a higher frequency of critical, exploitable security flaws, nor did it successfully alleviate standard, low-level coding issues commonly found in web development. The baseline for severe logical flaws and minor warnings remained consistent across both development methodologies.

The most notable divergence between the two applications is visually evident in the medium-risk and informational categories. The AI-assisted application demonstrates a marked increase in these areas, reflecting a 66% increase in medium-risk alerts (from 6 to 10) and a 43% increase in informational alerts (from 7 to 10) compared to the traditionally developed application.

As visually demonstrated by the chart's distribution, the higher overall vulnerability count for the AI-assisted application is driven entirely by these mid-to-lower-tier anomalies. This pattern indicates that

while AI code generators may match human performance in avoiding critical vulnerabilities, they appear more prone to generating generic, unoptimized code or broad configurations. Consequently, this triggers a higher volume of medium-severity warnings and informational observations during dynamic application security testing (DAST).

Table 4.2
Comparison of High-Risk Vulnerabilities

Vulnerability	**Traditional**	**AI-Assisted**
Vulnerable JavaScript Library	1	1

Both applications contained one high-risk finding related to the use of an outdated third-party JavaScript library. Since the same external component was used in both implementations, the detected vulnerability was common to both applications and did not differentiate their overall security posture.

Table 4.3
Comparison of Medium-Risk Vulnerabilities

Vulnerability	**Traditional**	**AI-Assisted**	**OWASP Top 10**	**CWE**	**CVSS**
Absence of Anti-CSRF Tokens	3	5	A01 Broken Access Control	CWE-352	5.4
Content Security Policy Header Not Set	1	1	A05 Security Misconfiguration	CWE-693	5.4
Cross-Domain Misconfiguration (CORS)	3	7	A05 Security Misconfiguration	CWE-942	6.5
Missing Anti-clickjacking Header	3	2	A05 Security Misconfiguration	CWE-1021	5.4
Subresource Integrity Missing	3	3	A08 Software and Data Integrity Failures	CWE-353	6.5
CSP Missing Directives	–	1	A05 Security Misconfiguration	CWE-693	5.4
CSP script-src unsafe-inline	–	1	A05 Security Misconfiguration	CWE-693	5.4
CSP script-src unsafe-eval	–	1	A05 Security Misconfiguration	CWE-693	5.4
CSP style-src unsafe-inline	–	1	A05 Security Misconfiguration	CWE-693	5.4
Vulnerable JavaScript Library (Additional Detection)	1	2	A06 Vulnerable Components	CWE-1104	Official CVE

The medium-risk findings were primarily associated with security configuration weaknesses, including missing anti-CSRF protection, Content Security Policy (CSP) misconfigurations, Cross-Origin Resource Sharing (CORS) misconfigurations, and missing Subresource Integrity (SRI) attributes. Compared with the traditional application, the AI-assisted application generated additional CSP-related findings and a higher frequency of CORS and anti-CSRF issues. These findings indicate that configuration-related security controls were implemented less consistently in the AI-assisted application.

Table 4.4
Comparison of Low-Risk Vulnerabilities

Vulnerability	Traditional	AI-Assisted	CWE
Cookie No HttpOnly Flag	2	1	CWE-1004
Cookie Without Secure Flag	3	3	CWE-614
Cookie SameSite=None	1	1	CWE-1275
Cookie without SameSite	4	3	CWE-1275
Cross-Domain JavaScript Source Inclusion	3	5	CWE-829
In Page Banner Information Leak	4	4	CWE-200
X-Powered-By Header Leak	4	Present in multiple responses	CWE-200
Server Version Disclosure	Present in multiple responses	Present in multiple responses	CWE-200
HSTS Disabled	1	1	CWE-319
HSTS Header Not Set	12	Present in multiple responses	CWE-319
Timestamp Disclosure	Present in multiple responses	Present in multiple responses	CWE-200
X-Content-Type-Options Header Missing	Present in multiple responses	Present in multiple responses	CWE-16

The low-risk findings largely consisted of missing cookie security attributes, missing HTTP security headers, and information disclosure through server response headers. Both applications exhibited similar numbers of these findings, indicating that they shared comparable baseline security configurations. Although individually considered low risk, these weaknesses may collectively increase the application's attack surface if left unaddressed.

Table 4.5
Informational Findings

Finding	Traditional	AI-Assisted
Authentication Request Identified	1	1
Information Disclosure – Suspicious Comments	2	OWASP ZAP detected this alert on 22 separate responses.
Loosely Scoped Cookie	1	1
Modern Web Application	3	Present in multiple responses
Cache-Control Directives	3	4
Retrieved from Cache	1	4
Session Management Response	2	2
Obsolete CSP Header	–	1

CSP Legacy Headers	–	2

Informational findings represent observations identified during automated scanning that may require further verification but do not necessarily constitute confirmed vulnerabilities. The AI-assisted application generated more informational findings, particularly those related to suspicious comments and cache behavior. These observations provide additional insight into potential areas for review but were not included in the vulnerability severity assessment.

OWASP Top 10 Classification

Table 4.6
Comparison of Vulnerabilities Based on OWASP Top 10 (2021)

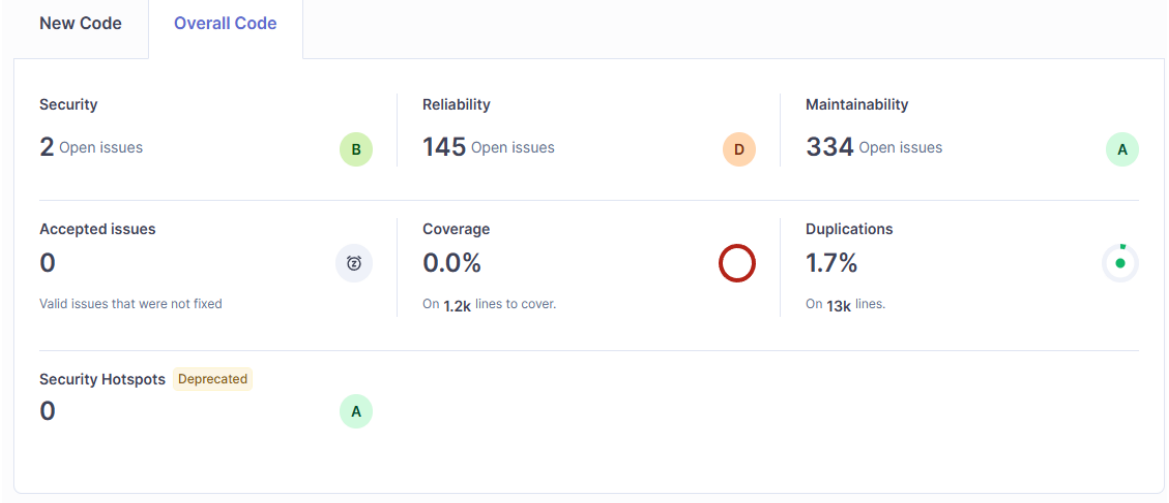
OWASP Top 10 Category	Traditional	AI-Assisted
A01 Broken Access Control	✓	✓
A02 Cryptographic Failures	✓	✓
A03 Injection	✓	✓
A05 Security Misconfiguration	✓	✓
A06 Vulnerable and Outdated Components	✓	✓
A07 Identification and Authentication Failures	✓	✓
A08 Software and Data Integrity Failures	✓	✓
A09 Security Logging and Monitoring Failures	✓	✓

The vulnerabilities detected by OWASP ZAP were classified according to the OWASP Top 10 (2021) categories to determine the types of security risks affecting each application. Both applications were affected by the same major categories, including Broken Access Control, Injection, Security Misconfiguration, Vulnerable and Outdated Components, Identification and Authentication Failures, Software and Data Integrity Failures, and Security Logging and Monitoring Failures. Although the categories were similar, the AI-assisted application contained a greater number of findings within.

SonarQube Static Code Analysis

Figure 4.3

Overview of SonarQube Static Code Analysis Results of the Traditional Application



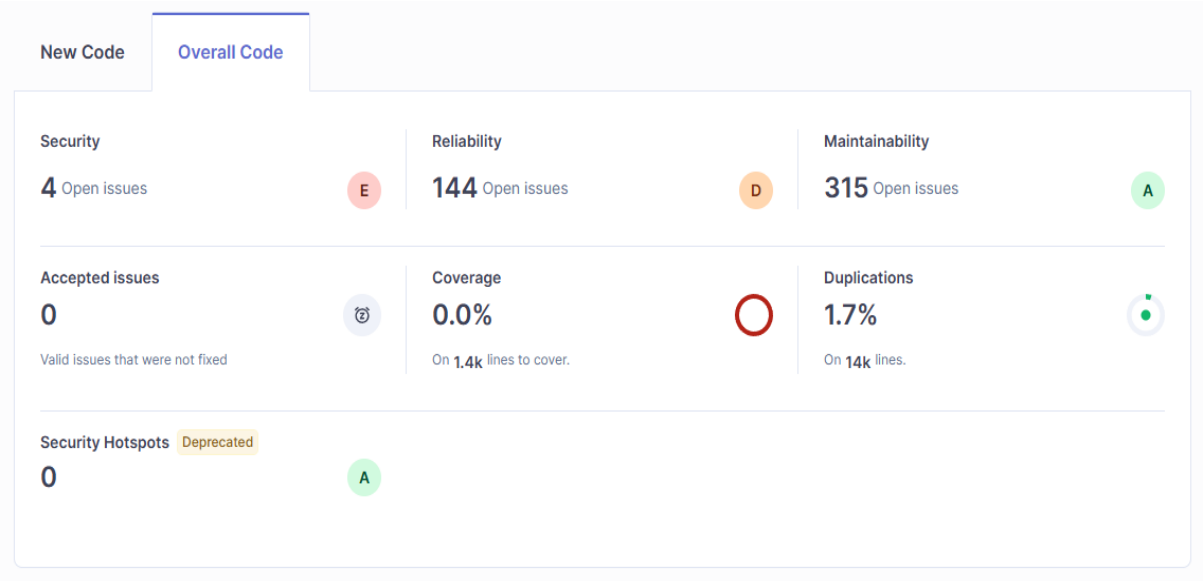


Table 4.7
Comparison of SonarQube Static Code Analysis Results

Evaluation Metric	Traditional	AI-Assisted
Security Issues	2	4
Security Rating	B	E
Reliability Issues	145	144
Reliability Rating	D	D
Maintainability Issues	334	315
Maintainability Rating	A	A
Test Coverage	0.0%	0.0%
Code Duplication	1.7%	1.7%
Security Hotspots	0	0

SonarQube Community Edition was used to perform Static Application Security Testing (SAST) on both source codes. As shown in Table 4.7, the traditional application contained two security issues and received a Security Rating of B, while the AI-assisted application contained four security issues and received a Security Rating of E. According to SonarQube, a B rating indicates minor security concerns, whereas an E rating indicates at least one severe security issue requiring remediation. Both applications obtained identical Reliability (D) and Maintainability (A) ratings, as well as identical test coverage and code duplication metrics, indicating comparable software quality outside the identified security issues.

Figure 4.4
Comparison of SonarQube Static Code Analysis Results

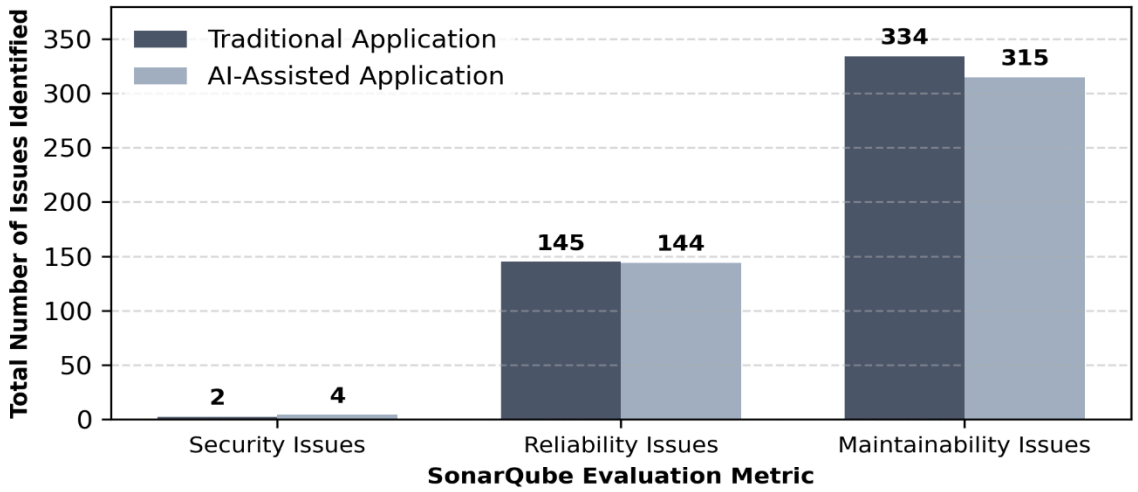


Figure 4.4 illustrates the comparative results of the SonarQube static code analysis for both the traditional and AI-assisted applications. The evaluation encompasses three primary metrics: Security Issues, Reliability Issues, and Maintainability Issues. The data indicates that while the AI-assisted application exhibited a marginal improvement in code maintainability (315 issues compared to 334) and demonstrated near parity in reliability (144 issues compared to 145), it yielded a 100% increase in security issues (4) relative to the traditionally developed application (2).

CVSS Severity Assessment

Table 4.8
Summary of CVSS Severity Classification

Vulnerability	**CWE**	**CVSS v3.1**	**Severity**
Vulnerable JavaScript Library	CWE-1104	Official CVE Score	Critical
Anti-CSRF Missing	CWE-352	5.4	Medium
CSP Misconfiguration	CWE-693	5.4	Medium
CORS Misconfiguration	CWE-942	6.5	Medium
Missing Clickjacking Protection	CWE-1021	5.4	Medium
Missing SRI	CWE-353	6.5	Medium
Cookie Missing HttpOnly	CWE-1004	5.4	Medium
Cookie Missing Secure	CWE-614	5.3	Medium
Cookie Missing SameSite	CWE-1275	5.4	Medium
Information Disclosure	CWE-200	5.3	Medium
HSTS Missing	CWE-319	5.3	Medium
X-Content-Type-Options Missing	CWE-16	5.4	Medium

Table 4.9 CVSS Metric

Metric	Traditional	AI-Assisted
Critical	1	1
High	0	0
Medium	10	14
Low	0	0

The identified vulnerabilities were mapped to their corresponding Common Weakness Enumeration (CWE) entries and assessed using the Common Vulnerability Scoring System (CVSS) Version 3.1. Official CVSS scores were used for vulnerabilities with available CVE references, while researcher-derived scores were assigned to security misconfigurations without published CVEs. The assessment revealed that both applications contained one Critical-severity vulnerability caused by a vulnerable JavaScript library, indicating the presence of a component-level security risk requiring immediate attention.

For Medium-severity vulnerabilities, the traditional application recorded ten findings, while the AI-assisted application recorded fourteen findings. These vulnerabilities were primarily related to security configuration weaknesses, including missing anti-CSRF protection, improper CORS settings, missing security headers, and insecure cookie configurations. The higher number of medium-severity findings in the AI-assisted application indicates that additional security weaknesses were introduced during development, particularly in application configuration and security controls.

Although both applications exhibited the same critical vulnerability level, the AI-assisted application demonstrated a higher overall vulnerability count due to the increased number of medium-severity issues. These results suggest that AI-assisted code generation may require additional security validation and review to ensure that generated implementations follow secure development practices.

Overall Comparative Analysis

Table 4.10
Overall Comparison of Security Assessment Results

Metric	Traditional	AI-Assisted
OWASP ZAP High Alerts	1	1
OWASP ZAP Medium Alerts	6	10
OWASP ZAP Low Alerts	12	12
OWASP ZAP Informational Alerts	7	10
Total ZAP Alerts	26	33
SonarQube Security Issues	2	4
Security Rating	B	E
Reliability Rating	D	D
Maintainability Rating	A	A
OWASP Top 10 Categories Affected	8	8
Distinct CWE Weaknesses	12	12

Highest CVSS Severity	Critical	Critical

The results obtained from OWASP ZAP, SonarQube, OWASP Top 10 classification, CWE mapping, and CVSS scoring were consolidated to compare the security posture of both applications. In SonarQube, higher ratings indicate better quality, where A represents the best rating and lower ratings indicate increasing issues in security, reliability, or maintainability.

The traditional application produced fewer security findings and achieved a higher Security Rating of B, indicating fewer security issues compared to the AI-assisted application, which received a Security Rating of E, indicating significant security weaknesses. Both applications obtained a Reliability Rating of D, while maintaining an A Maintainability Rating, showing similar code maintainability despite identified issues.

Both applications were affected by eight OWASP Top 10 categories, contained twelve distinct CWE weaknesses, and reached a highest CVSS severity of Critical. Overall, the traditional application demonstrated a stronger security posture based on the evaluation results, although both applications require vulnerability remediation before production deployment.

Table 4.11
Quantitative Comparison of Security Assessment Results

Metric	Traditional	AI-Assisted	Difference	% Difference	Risk Ratio
Total ZAP Alerts	**26**	**33**	**+7**	**26.9%**	**1.27**
Medium Alerts	**6**	**10**	**+4**	**66.7%**	**1.67**
Sonar Security Issues	**2**	**4**	**+2**	**100%**	**2.00**

Table 4.11 presents additional quantitative summaries of the comparative assessment. The AI-assisted application generated 26.9% more total OWASP ZAP alerts than the traditional application. Medium-risk vulnerabilities increased by 66.7%, while SonarQube identified 100% more security issues. The calculated risk ratios indicate that the AI-assisted application exhibited 1.27 times more overall vulnerabilities, 1.67 times more medium-risk findings, and 2.0 times more static security issues. These quantitative measures reinforce the descriptive findings and provide stronger evidence that the AI-assisted application demonstrated a weaker security posture under the testing conditions.

Based on OWASP ZAP, SonarQube, OWASP Top 10, CWE mapping, CVSS scoring, and the quantitative comparison presented in Table 4.11, the traditional application demonstrated fewer detected vulnerabilities and achieved a better overall security rating. Although both applications were affected by similar categories of weaknesses, the AI-assisted application consistently exhibited higher counts of medium-risk vulnerabilities and static security issues. These results suggest that AI-assisted software development may require additional security review to ensure compliance with secure coding practices.

IV. RESULT AND DISCUSSION

The security assessment conducted using OWASP ZAP Version 2.17.0 identified differences between the traditionally developed and AI-assisted web applications. Both applications were evaluated using the same deployment environment and scanning configuration to establish a fair comparison of their security characteristics. The dynamic analysis produced 26 alerts for the traditional application and 33 alerts for the AI-assisted application.

Both applications were affected by one high-risk vulnerability associated with a vulnerable third-party JavaScript library. However, differences were observed in the distribution of other detected issues. The AI-assisted application produced 10 medium-risk findings, while the traditional application recorded 6 medium-risk findings. In addition, the AI-assisted application generated 10 informational alerts compared with 7 informational alerts from the traditional application. The higher number of detected alerts suggests that the AI-assisted application contained more security weaknesses observable through automated dynamic testing. However, this result should not be interpreted as confirmation that AI-assisted development directly introduced every vulnerability identified during the assessment. Instead, the findings suggest that AI-generated implementations may require additional security verification to identify weaknesses that are not immediately apparent from functional evaluation alone.

The medium-risk vulnerabilities identified during the assessment were mainly associated with missing security mechanisms and improper configuration practices. These included the absence of anti-Cross-Site Request Forgery (CSRF) protection, Cross-Origin Resource Sharing (CORS) configuration weaknesses, Content Security Policy (CSP) issues, missing clickjacking protection, and the absence of Subresource Integrity (SRI) attributes. Several of these configuration-related weaknesses appeared more frequently in the AI-assisted application. The presence of these findings indicates that AI-assisted programming can generate applications capable of meeting functional requirements while still requiring additional security improvement. Security features such as browser protection policies, request validation controls, and secure configuration settings may not always be automatically included in generated code. Therefore, developers must review and strengthen security-related components after incorporating AI-generated outputs into software projects. The results further emphasize that functional performance alone cannot be used as the only measurement of software quality. An application may successfully perform its intended operations while still containing weaknesses that could affect confidentiality, integrity, and availability. Regardless of the development approach used, security assessment remains necessary to identify and address possible vulnerabilities before deployment. The vulnerable third-party JavaScript library detected in both applications requires separate interpretation. Since the same vulnerable component was present in both systems, the finding cannot be directly associated with either AI-assisted or traditional development. Instead, it represents a dependency management issue that may affect software regardless of the development method. The findings show the importance of checking third-party libraries and keeping software components updated throughout the development process.

The SonarQube Community Edition assessment revealed variations in the source code security results of the two applications. The traditionally developed application recorded two security issues and achieved a Security Rating of B, whereas the AI-assisted application recorded four security issues and received a Security Rating of E. These results indicate that more source-code-related security concerns were identified in the AI-assisted application during static analysis. The findings are consistent with the observations of Perry et al. (2023), who investigated the effect of AI coding assistants on developers' security practices. Their study reported that developers who used AI assistance generated less secure solutions in security-oriented programming tasks compared with those who developed solutions without AI support. The researchers further noted that overreliance on AI-generated recommendations may influence developers' security judgment and decision-making. In relation to the present study, the higher number of SonarQube security findings in the AI-assisted application provides additional evidence of possible source-code security challenges when using AI-assisted development. However, these results do not imply that AI-assisted applications are generally inferior to traditionally developed software. Both applications obtained the same Reliability Rating of D and Maintainability Rating of A. Additionally, both systems recorded 0.0% test coverage and 1.7% code duplication, showing comparable results in these software quality measurements.

These similarities indicate that the primary difference between the applications was related to security findings rather than general software quality characteristics. This observation supports the argument presented by Cotroneo et al. (2025), who reported that AI-generated and human-developed code may differ across multiple quality dimensions. Their research found that AI-generated code may demonstrate higher security risks, while human-written code may exhibit differences in complexity and maintainability characteristics.

V. CONCLUSION AND RECOMMENDATION

This study investigated the security performance of AI-assisted and traditionally developed web applications using several assessment methods, including OWASP ZAP, SonarQube, OWASP Top 10 (2021), Common Weakness Enumeration (CWE), and Common Vulnerability Scoring System (CVSS) Version 3.1. The assessment results showed that the AI-assisted application generated more security findings, with most differences observed in configuration-related weaknesses and medium-severity vulnerabilities. Both applications contained the same critical vulnerability linked to a vulnerable third-party JavaScript library, however, the overall evaluation from dynamic and static testing indicated that the traditionally developed application achieved a stronger security outcome.

The findings demonstrate that AI-assisted programming can improve development efficiency and support faster software creation. However, the ability of AI tools to generate functional code does not guarantee that the resulting application will be secure. Generated code may still require improvements in areas such as security configurations, protection mechanisms, and dependency management. Therefore, AI-produced code should undergo careful review, testing, and security validation before being incorporated into operational systems. AI coding assistants should be viewed as tools that support developers rather than replace established secure development practices. Human involvement remains necessary for reviewing generated code, verifying security controls, and performing appropriate security assessments. Applying both Static Application Security Testing (SAST) and Dynamic Application Security Testing (DAST) provides a broader evaluation by identifying

weaknesses in both source code and application behavior. The results also provide implications for computing education and organizational software practices. Academic programs should introduce responsible AI usage alongside secure coding principles to help future developers evaluate AI-generated outputs effectively. Organizations using AI-assisted development tools should establish guidelines for their use, integrate security testing into development workflows, and maintain processes for vulnerability monitoring.

Future research may explore other AI programming tools, larger software systems, and additional evaluation techniques such as manual penetration testing. Overall, the study indicates that AI-assisted development can produce applications with comparable functionality to traditionally developed systems, but continuous security assessment remains necessary to ensure software reliability and protection.

REFERENCES

[1]. Cotroneo, D., Improta, C., & Liguori, P. (2025). Human-written vs. AI-generated code: A large-scale study of defects, vulnerabilities, and complexity. *Proceedings of the 2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*, 252–263. https://doi.org/10.1109/ISSRE66568.2025.00035

[2]. FIRST. (2019). *Common Vulnerability Scoring System version 3.1: Specification document*. https://www.first.org/cvss/v3.1/specification-document

[3]. Jarupunphol, P., Seatun, S., & Buathong, W. (2023). Measuring vulnerability assessment tools' performance on the university web application. *Pertanika Journal of Science and Technology, 31*(6), 2973–2993. https://doi.org/10.47836/pjst.31.6.19

[4]. Lenarduzzi, V., Saarimäki, N., & Taibi, D. (2020). Some SonarQube issues have a significant but small effect on faults and changes: A large-scale empirical study. *Journal of Systems and Software, 170*, Article 110750. https://doi.org/10.1016/j.jss.2020.110750

[5]. Majdinasab, V., Bishop, M. J., Rasheed, S., Moradidakhel, A., Tahir, A., & Khomh, F. (2023). *Assessing the security of GitHub Copilot generated code—A targeted replication study*. arXiv. https://arxiv.org/abs/2311.11177

[6]. MITRE. (2024). *Common Weakness Enumeration (CWE™).* https://cwe.mitre.org

[7]. National Institute of Standards and Technology. (2022). *Secure Software Development Framework (SSDF) Version 1.1: Recommendations for mitigating the risk of software vulnerabilities (NIST Special Publication 800-218).* https://csrc.nist.gov/pubs/sp/800/218/final

[8]. Noy, S., & Zhang, W. (2023). Experimental evidence on the productivity effects of generative artificial intelligence. *Science, 381*(6654), 187–192. https://doi.org/10.1126/science.adh2586

[9]. Open Worldwide Application Security Project. (2021). *OWASP Top 10: The ten most critical web application security risks*. https://owasp.org/www-project-top-ten/

[10]. Open Worldwide Application Security Project. (2024). *OWASP Zed Attack Proxy (ZAP) documentation*. https://www.zaproxy.org/docs/

[11]. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. *Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP)*, 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[12]. Perry, N., Srivastava, M., Kumar, D., & Boneh, D. (2023). Do users write more insecure code with AI assistants? *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2745–2759. https://doi.org/10.1145/3576915.3623157

[13]. SonarSource. (2024). *SonarQube documentation*. https://docs.sonarsource.com/sonarqube/latest/