

Understanding Security Risks in AI-Generated Code: A Literature Survey on GitHub Copilot

Manas R. Sangale¹, Gauravkumar D. Jade², Himani R. Sonawane³, Dr. Abhijit R. More⁴
^{1,2,3,4}SSVPS's B.S.Deore Collge of Engineering, Dhule,MH.

Abstract—AI-powered coding assistants, such as GitHub Copilot, are rapidly changing the way software is developed by helping programmers write code more efficiently. Recent industry reports indicate that these tools now generate a substantial portion of the code produced by active developers. Despite these productivity gains, an increasing number of studies have raised concerns about the security of AI-generated code, showing that it often contains vulnerabilities that could be exploited in real-world applications. This survey reviews the current research on the security of AI-assisted code generation by examining three landmark studies: Pearce *et al.*'s controlled evaluation of GitHub Copilot, Fu *et al.*'s analysis of Copilot-generated code in public GitHub repositories, and Perry *et al.*'s investigation of developer behavior when using AI coding assistants. The survey also discusses recent developments, including reported Common Vulnerabilities and Exposures (CVEs) associated with AI-generated software and the Rules File Backdoor supply-chain attack disclosed in 2025. In addition, findings from the 2025 Stack Overflow Developer Survey are analyzed to understand current adoption trends and developer trust in AI coding tools. The survey highlights a growing gap between the widespread use of these assistants and developers' confidence in the security and reliability of their outputs. Overall, the findings suggest that while AI coding assistants offer significant productivity benefits, they should be used with appropriate safeguards, including rigorous code review, automated security testing, and continuous developer awareness, to reduce potential security risks.

Index Terms—GitHub Copilot, AI-generated code, software security, code vulnerabilities, large language models, secure software development.

I. INTRODUCTION

Artificial intelligence (AI)-powered coding assistants have rapidly evolved from experimental research tools into an integral part of modern software development.

Among these, GitHub Copilot was the first AI-based coding assistant to gain widespread adoption, introducing the concept of an “AI pair programmer.” Since its release, it has been followed by several competing tools that provide intelligent code generation and programming support. According to recent industry reports, GitHub Copilot now generates a substantial portion of the code written by its active users, exceeding half of the generated code in certain programming languages, such as Java. Furthermore, AI-assisted development has become commonplace, with millions of developers across thousands of organizations relying on these tools in their daily workflows [9].

Despite their widespread adoption and clear productivity benefits, the security implications of AI-generated code remain insufficiently understood. Large language models (LLMs) used for code generation are trained on vast collections of publicly available source code, much of which contains programming errors, insecure coding practices, and outdated implementation patterns. Consequently, these models may reproduce insecure code constructs learned during training and, in some situations, even reinforce them. Because AI-generated code is often presented with high confidence and appears syntactically correct, developers may accept these suggestions without performing adequate security validation, increasing the likelihood of introducing vulnerabilities into software systems.

This survey reviews the existing literature on the security of AI-generated code, with a primary focus on GitHub Copilot, the most extensively investigated AI coding assistant in current research. Section II summarizes the key empirical studies that established the foundation of this research area. Section III discusses the most frequently reported categories of security vulnerabilities found in AI-generated code.

Section IV examines real-world evidence of these risks through documented Common Vulnerabilities and Exposures (CVEs) and emerging attack techniques, including the Rules File Backdoor supply-chain attack. Section V explores recent trends in the adoption of AI coding assistants and developers' trust in their outputs, while Section VI outlines practical strategies for mitigating the identified security risks. Finally, Section VII concludes the survey by summarizing the major findings and identifying future challenges.

It is important to note that reported vulnerability rates differ across studies because of variations in research methodology, including controlled prompt-based evaluations, analyses of real-world repositories, and replication studies using updated AI models. Therefore, the numerical results presented throughout this survey should be interpreted as consistent evidence of an ongoing and significant security concern rather than as a single definitive measure of vulnerability. Figure 1 provides a chronological overview of the major studies and security events discussed in this survey.

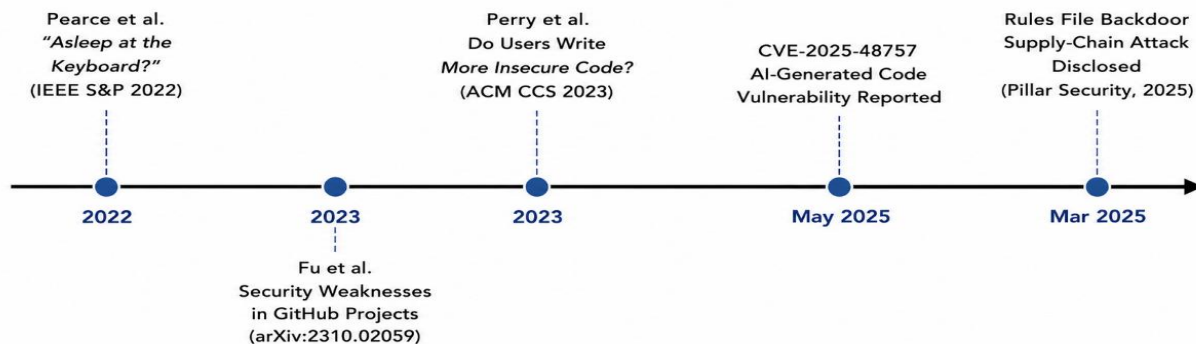


Fig. 1. Timeline of key research findings and security events in AI-generated code, 2022–2025.

Research Gap and Contributions: Although several studies have examined the security of AI-generated code, most have focused on a single aspect of the problem, such as controlled experiments, analysis of production repositories, or developer behavior. Recent developments, including AI-related Common Vulnerabilities and Exposures (CVEs), supply-chain attacks such as the Rules File Backdoor, and changing developer trust in AI coding assistants, have largely been discussed in isolation. As a result, there is limited literature that brings together empirical research findings, real-world security incidents, and industry adoption trends into a single, coherent overview.

This survey addresses that gap by providing a comprehensive review of the security risks associated with AI-generated code, with a primary focus on GitHub Copilot and related AI coding assistants. Unlike earlier surveys, this paper combines evidence from controlled experiments, real-world repository analyses, developer behavior studies, documented CVEs, emerging AI-specific attack techniques, and recent developer surveys. It also discusses practical mitigation strategies that can help organizations adopt

AI-assisted software development while maintaining secure software engineering practices.

The key contributions of this survey are summarized as follows:

- It reviews and synthesizes the major empirical studies on the security of AI-generated code published between 2022 and 2025.
- It summarizes the most frequently reported vulnerability categories and explains their implications for secure software development.
- It discusses recent real-world security incidents, including AI-related CVEs and the Rules File Backdoor attack.
- It examines recent trends in developer adoption and trust using industry survey data.
- It presents practical recommendations for reducing the security risks associated with AI-assisted software development.

II. LITERATURE SELECTION METHODOLOGY

This survey reviews the current literature on the security of AI-generated code, with a primary focus on GitHub Copilot and related AI coding assistants.

Relevant publications released between 2022 and 2025 were identified through searches of IEEE Xplore, ACM Digital Library, SpringerLink, Google Scholar, and arXiv. To capture recent industry developments, official reports from GitHub, Stack Overflow, the National Vulnerability Database (NVD), and verified security advisories were also considered.

The literature search used keywords such as GitHub Copilot, AI-generated code, code security, software vulnerabilities, large language models, AI coding assistants, and secure software development. Studies were included if they (i) presented empirical evaluations of AI-generated code security, (ii) investigated developer interactions with AI coding assistants, (iii) analyzed real-world security incidents or Common Vulnerabilities and Exposures (CVEs) related to AI-generated code, or (iv) proposed mitigation strategies for secure AI-assisted software development. Editorial articles, opinion pieces, and unverified online reports were excluded unless they documented officially disclosed security incidents or were supported by credible technical evidence.

The selected studies were analyzed and grouped into four categories: foundational empirical studies, replication and extension studies, real-world security incidents, and developer adoption and mitigation research. This organization provides a comprehensive overview of both the current research landscape and the practical security challenges associated with AI-assisted software development.

III. RELATED WORK

A. Pearce et al. (2022) — Asleep at the Keyboard?

One of the earliest and most influential investigations into the security of AI-generated code was conducted by Pearce et al. in their landmark study, *Asleep at the Keyboard?*, presented at the 2022 IEEE Symposium on Security and Privacy [1]. The researchers carried out a systematic evaluation of GitHub Copilot by creating 89 programming scenarios representing vulnerabilities from the MITRE Top 25 Common Weakness Enumeration (CWE) categories. Each scenario was executed multiple times to account for the non-deterministic nature of Copilot's code generation, resulting in a dataset of 1,689 generated programs.

The study reported that approximately 40% of the generated programs contained at least one exploitable vulnerability corresponding to the targeted CWE category. The vulnerability rate varied considerably across programming languages. In particular, C programs exhibited vulnerability rates approaching 50%, largely because the language relies on manual memory management and provides fewer built-in safety mechanisms than higher-level languages such as Python. The researchers also observed that the security of Copilot's suggestions was strongly influenced by both the wording of the input prompt and the specific vulnerability category being evaluated. These findings demonstrate that the security of AI-generated code is highly context-dependent rather than consistently reliable across different programming tasks.

B. Fu et al. (2023) — Security Weaknesses in Real GitHub Projects

While the work of Pearce et al. focused on controlled experimental settings, Fu et al. investigated how AI-generated code performs after it has been incorporated into real-world software projects. Their study, *Security Weaknesses of Copilot-Generated Code in GitHub Projects*, analyzed 452 code snippets that were explicitly identified as being generated by GitHub Copilot and later committed to public GitHub repositories [2]. Unlike laboratory-based evaluations, this research examined AI-generated code that had already undergone developer review and had been accepted into production codebases.

Static security analysis revealed that 35.8% of the analyzed code snippets contained at least one security weakness. Language-specific results showed vulnerability rates of 32.8% for Python and 24.5% for JavaScript. Across the dataset, the researchers identified 38 different CWE categories, with the most frequently occurring issues being CWE-330 (Use of Insufficiently Random Values), CWE-78 (OS Command Injection), and CWE-703 (Improper Check or Handling of Exceptional Conditions). Notably, eight of these weaknesses are included in the 2023 CWE Top 25 Most Dangerous Software Weaknesses. Because the analyzed code had already passed through human review before being merged into active repositories, the findings suggest that insecure AI-generated code is not always detected during conventional code review processes. This highlights

an important limitation of relying solely on developer oversight and emphasizes the need for automated security analysis alongside AI-assisted software development.

C. Perry et al. (2023) — Developer Behavior and False Confidence

While earlier studies primarily examined the security of AI-generated code, Perry et al. shifted the focus toward the developers who use AI coding assistants. Their study, *Do Users Write More Insecure Code with AI Assistants?* presented at the 2023 ACM SIGSAC Conference on Computer and Communications Security, explored how AI-assisted programming influences developer behavior and decision-making [3].

The researchers conducted a controlled user study in which participants completed security-related programming tasks either with or without access to an AI coding assistant. The results showed that participants who relied on AI assistance generally produced code with more security vulnerabilities than those in the control group. More importantly, these participants were also more likely to believe that their solutions were secure, despite the presence of vulnerabilities. The authors described this mismatch between perceived and actual security as a “false sense of security.”

The study also highlighted the importance of how developers interact with AI tools. Participants who critically evaluated AI-generated suggestions—such as by refining prompts, requesting alternative solutions, or modifying generated code instead of accepting the first recommendation—tended to produce more secure implementations. These findings suggest that secure outcomes depend not only on the capabilities of the AI assistant but also on the quality of human oversight and critical engagement during the development process.

Taken together, the studies by Pearce et al., Fu et al., and Perry et al. reveal a consistent pattern. Across both controlled experiments and real-world software projects, AI-generated code frequently contains security weaknesses, while developers often overestimate the reliability of AI-generated solutions. This combination of technical limitations and misplaced confidence presents a significant challenge for secure software development. It also indicates that improving developer awareness alone is unlikely to

eliminate these risks unless AI-generated code is accompanied by rigorous review and verification practices.

D. Subsequent Replications and Extensions

Following the pioneering work of Pearce et al. and Fu et al., several researchers have revisited these findings to validate, refine, and extend the understanding of security risks associated with AI-generated code.

Majdinasab et al. replicated the methodology proposed by Pearce et al. using a newer version of GitHub Copilot and an updated collection of Common Weakness Enumeration (CWE) scenarios [8]. Their results indicated modest improvements in certain vulnerability categories, suggesting that newer model versions had become better at avoiding some common insecure coding patterns. Nevertheless, the overall rate of security vulnerabilities remained considerable, with issues such as command injection and insecure random number generation continuing to appear frequently.

In a related study, Asare et al. investigated whether GitHub Copilot performs better than human developers when confronted with known security-sensitive programming tasks [12]. By recreating real-world C/C++ vulnerability scenarios, the authors found that Copilot reproduced vulnerable coding patterns in approximately one-third of the cases. This performance was broadly comparable to that of human developers, indicating that the AI assistant was neither substantially more secure nor significantly less secure than its human counterparts in these scenarios.

Sandoval et al. extended this line of research through a user study involving experienced systems programmers [11]. Their findings showed that participants using an AI coding assistant were not significantly more likely to introduce critical security flaws than those working without AI support. However, the presence of AI assistance also did not lead to measurable improvements in software security. These results reinforce the view that AI coding assistants should complement, rather than replace, established secure development and code review practices.

Mastropaolo et al. explored another important aspect of AI-assisted programming by examining the robustness of GitHub Copilot to variations in prompt wording [13]. Their experiments demonstrated that even minor changes in prompts with identical

semantic meaning could produce substantially different code completions, with noticeable differences in security quality. This finding highlights the sensitivity of AI-generated code to prompt formulation and suggests that code generation models remain heavily influenced by contextual phrasing rather than solely by the underlying programming task.

Collectively, these replication and extension studies reinforce the conclusions of earlier research. Although successive generations of AI coding assistants have shown incremental improvements, security vulnerabilities continue to occur across multiple programming languages and application domains. The evidence consistently indicates that AI-generated code should not be assumed to be secure by default and must be subjected to thorough human review and automated security analysis before deployment.

A common observation across these replication studies is that accurately measuring the security of AI-

generated code remains challenging. Frequent updates to AI models, differences in experimental design, and variations in how security weaknesses are identified all influence the reported vulnerability rates. As a result, the percentages reported in different studies should be viewed as evidence of a continuing and meaningful security concern rather than as fixed or directly comparable values. Despite these methodological differences, the overall findings remain consistent. Programming languages such as C, which rely on manual memory management, continue to exhibit higher vulnerability rates than memory-managed languages. Similarly, injection-related vulnerabilities remain among the most commonly reported security issues across different studies and model versions. Figure 2 presents a comparison of the vulnerability rates reported in the major empirical studies discussed in this section.

TABLE I. COMPARISON OF MAJOR EMPIRICAL STUDIES ON AI-GENERATED CODE SECURITY

Study	Year	Research Method	Dataset / Participants	Key Findings	Limitations
Pearce et al. [1]	2022	Controlled experiments using 89 security scenarios	1,689 AI-generated programs	~40% of generated code contained security vulnerabilities; C programs showed the highest vulnerability rate	Controlled environment may not fully represent real-world development
Fu et al. [2]	2023	Static analysis of GitHub repositories	452 Copilot-generated code snippets	35.8% of snippets contained at least one security weakness across 38 CWE categories	Limited to publicly identifiable Copilot-generated code
Perry et al. [3]	2023	Controlled user study	Developers performing security-related programming tasks	AI-assisted participants produced less secure code and showed greater confidence in incorrect solutions	Focused on user behavior rather than vulnerability detection
Majdinasab et al. [8]	2024	Replication study	Updated Copilot model with revised CWE scenarios	Moderate improvement in some vulnerability categories, but major weaknesses remained	Evaluated a limited set of security scenarios
Asare et al. [12]	2024	Replay of real-world vulnerabilities	Historical C/C++ vulnerability cases	Copilot reproduced vulnerable patterns in nearly one-third of cases,	Restricted to C/C++ programming tasks

				similar to human developers	
Sandoval et al. [11]	2024	User study	Systems programmers	AI assistance did not significantly improve software security outcomes	Limited participant population
Mastropaolo et al. [13]	2024	Prompt robustness evaluation	Semantically equivalent prompts	Minor prompt changes produced noticeably different security outcomes	Focused on prompt sensitivity rather than vulnerability frequency

Table I summarizes the principal empirical studies reviewed in this survey. Although the studies differ in terms of methodology, datasets, and evaluation settings, they consistently report that AI-generated code is susceptible to well-known software vulnerabilities. The findings also demonstrate that improvements in successive AI model versions have reduced certain classes of vulnerabilities but have not

eliminated persistent issues such as injection attacks, insecure random number generation, and improper input validation. Collectively, these studies suggest that secure use of AI coding assistants depends not only on advances in model capability but also on effective human oversight and automated security verification.

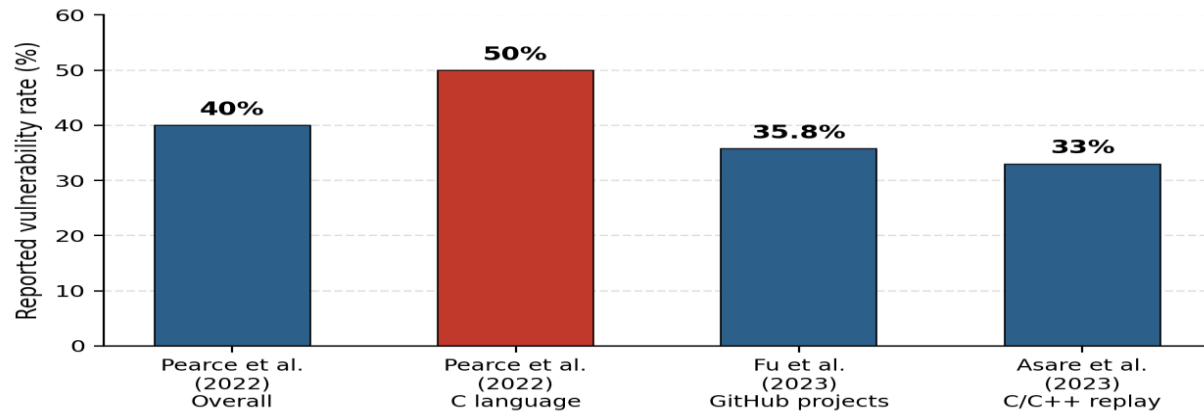


Fig. 2. Comparison of vulnerability rates reported in major empirical studies on AI-generated code security. Although reported percentages vary because of differences in study methodology and evaluation settings, all studies consistently identify a substantial prevalence of security vulnerabilities in AI-generated code.

Figure 2 compares the headline vulnerability rates reported by the principal empirical studies discussed in this survey. Although the reported percentages differ because of variations in datasets, programming languages, and experimental methodologies, the studies consistently demonstrate that AI-generated code remains susceptible to well-known software vulnerabilities. Controlled evaluations generally report higher vulnerability rates than repository-based analyses, while memory-unsafe languages such as C continue to exhibit the highest security risk.

The studies reviewed in the previous section collectively indicate that the security challenges associated with AI-generated code extend beyond individual software vulnerabilities. They also involve developer behavior, emerging real-world threats, and the effectiveness of mitigation strategies. To provide a structured overview of these interconnected aspects, Fig. 3 presents a taxonomy of the major security risks discussed throughout this survey.

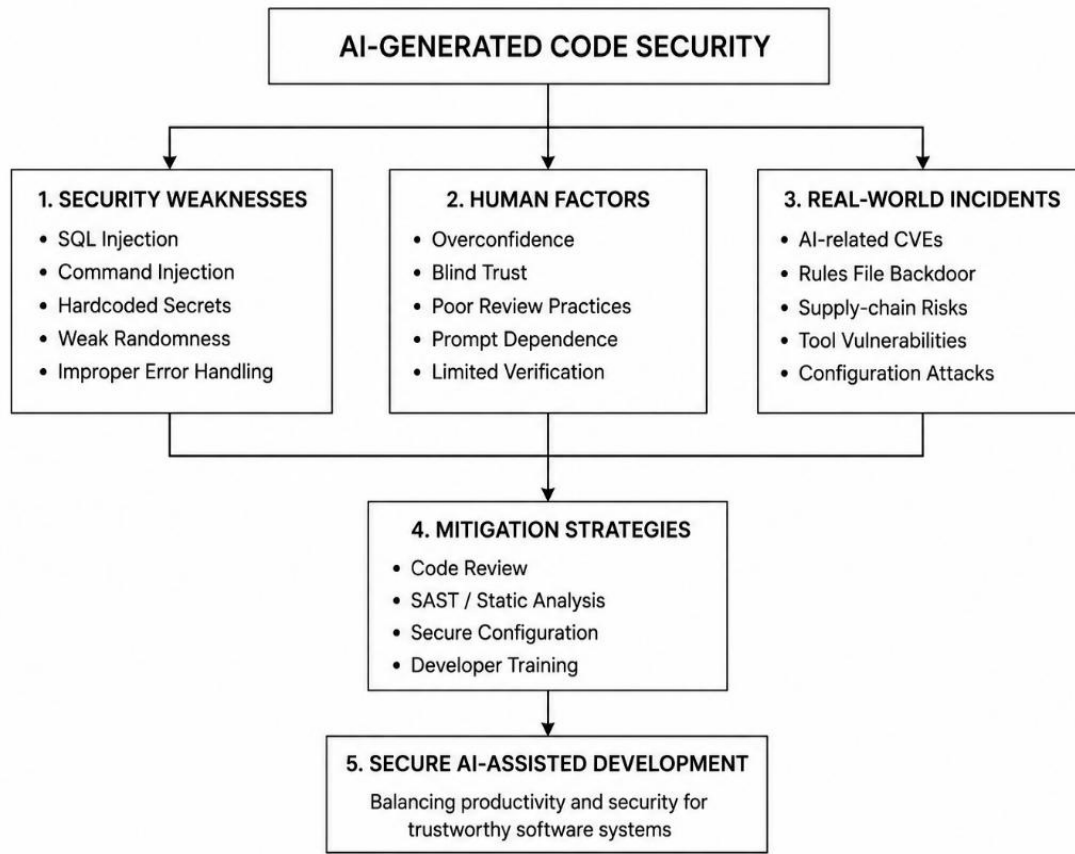


Fig. 3. Taxonomy of Security Risks in AI-Generated Code.

As illustrated in Fig. 3, the security risks associated with AI-generated code can be broadly grouped into three categories: technical security weaknesses, human-related factors, and real-world security incidents. Together, these categories highlight that securing AI-assisted software development requires not only improved code generation models but also secure development practices, effective human oversight, and continuous security verification. The following sections discuss each of these categories in greater detail.

IV. COMMON VULNERABILITY TYPES

The studies discussed in the previous section, along with subsequent replication studies such as that of Majdinasab et al. [8], consistently report a common set of security vulnerabilities in AI-generated code. Many of these weaknesses also appear in the MITRE list of the Top 25 Most Dangerous Software Weaknesses, highlighting their practical significance in modern software development [10]. The most frequently reported vulnerability categories are summarized below.

TABLE II. COMMON SECURITY VULNERABILITIES REPORTED IN AI-GENERATED CODE

CWE ID	Vulnerability	Typical Cause in AI-Generated Code	Security Impact	Reported By
CWE-89	SQL Injection	Dynamic query construction using string concatenation	Unauthorized database access and data leakage	Pearce et al. [1], Fu et al. [2]
CWE-78	OS Command Injection	Unsanitized user input passed to system commands	Remote code execution	Pearce et al. [1], Fu et al. [2], Majdinasab et al. [8]

CWE-798	Hardcoded Credentials	API keys, passwords, or secrets embedded in source code	Credential exposure and unauthorized access	Pearce et al. [1], Fu et al. [2]
CWE-330	Use of Insufficiently Random Values	Non-cryptographic random number generators used in security-sensitive contexts	Predictable tokens, weak session IDs, insecure cryptographic operations	Fu et al. [2], Majdinasab et al. [8]
CWE-703	Improper Exception Handling	Missing validation and inadequate error handling	Information disclosure and unstable application behavior	Fu et al. [2]

Table II summarizes the most frequently reported vulnerability categories identified across the empirical studies reviewed in this survey. Although the specific occurrence of each vulnerability varies with programming language, model version, and evaluation methodology, injection-related weaknesses consistently appear among the most common security issues. The recurrence of these vulnerabilities across multiple independent studies suggests that they represent systematic limitations of current AI code-generation models rather than isolated implementation errors.

SQL Injection (CWE-89): AI coding assistants such as GitHub Copilot often generate SQL queries by directly concatenating or interpolating user input into query strings instead of using parameterized queries or prepared statements. This insecure coding practice increases the risk of SQL injection attacks, allowing attackers to manipulate database queries and potentially access or modify sensitive information.

OS Command Injection (CWE-78): Command injection is another frequently reported weakness in AI-generated code. Studies by Pearce et al. and Fu et al. found that generated programs sometimes pass unvalidated user input directly to operating system commands or shell utilities. If left unchecked, such code can enable attackers to execute arbitrary system commands and compromise the underlying system.

Hardcoded Credentials and Secrets (CWE-798): AI-generated code occasionally includes hardcoded API keys, passwords, authentication tokens, or cryptographic keys within the source code. This behavior is likely influenced by publicly available code examples used during model training, where credentials are often included for demonstration

purposes. In production environments, hardcoded secrets create significant security risks because they can be easily exposed or misused.

Use of Insufficiently Random Values (CWE-330): Fu et al. identified this as the most frequently occurring vulnerability in their analysis of Copilot-generated code [2]. The issue typically arises when AI-generated programs use standard pseudo-random number generators instead of cryptographically secure alternatives for security-sensitive operations such as generating session identifiers, authentication tokens, or encryption keys. The use of predictable random values can weaken application security and increase the likelihood of successful attacks.

Improper Exception and Error Handling (CWE-703): Another commonly reported weakness is inadequate exception and error handling. AI-generated code often fails to properly validate error conditions or handle unexpected failures, which can leave applications in an insecure state. In some cases, detailed error messages may also reveal sensitive system information, making it easier for attackers to exploit the application.

These vulnerabilities are not merely theoretical observations reported in controlled experiments. Security issues such as SQL injection, command injection, and exposure of hardcoded credentials continue to be among the leading causes of real-world cyberattacks and data breaches. Consequently, the vulnerability patterns identified in studies of GitHub Copilot closely match the security flaws that software security teams routinely encounter and prioritize in production environments.

A notable observation across these vulnerability categories is that the secure implementation is often only slightly different from the insecure code generated by the model. For example, replacing string concatenation with parameterized SQL queries, using cryptographically secure random number generators instead of standard pseudo-random generators, or storing credentials in environment variables rather than hardcoding them can substantially improve application security. This suggests that the primary challenge is not that AI models are incapable of generating secure code. Instead, they tend to reproduce coding patterns that are more common in their training data, where tutorial examples and publicly available source code often prioritize simplicity and readability over secure coding practices.

This observation also points to practical opportunities for improving AI-assisted software development. Techniques such as security-aware prompt engineering, targeted model fine-tuning, and automated post-generation security analysis can help reduce many of these recurring vulnerabilities without requiring fundamental changes to the underlying language models. By combining these approaches with established secure software development practices, organizations can better balance the productivity benefits of AI coding assistants with the need to maintain software security.

V. REAL-WORLD IMPACT: CVEs AND THE RULES FILE BACKDOOR

A. Documented CVEs Linked to AI-Generated Code

By 2025, concerns raised by earlier research on AI-generated code had started to appear in the form of publicly disclosed security vulnerabilities. A notable example is CVE-2025-48757, reported in May 2025, which affected applications generated by an AI coding platform that did not enable Supabase Row Level Security (RLS) by default [7]. Since RLS must be explicitly configured to enforce database access controls, applications generated without this protection left backend data openly accessible. Subsequent security investigations identified hundreds of vulnerable endpoints across applications developed using the affected platform, demonstrating how insecure AI-generated configurations can create large-scale security risks.

During the same period, the Systems Software & Security Laboratory (SSLab) at Georgia Tech introduced the Vibe Security Radar project to monitor and analyze Common Vulnerabilities and Exposures (CVEs) associated with AI coding assistants [15]. The project reported a steady increase in confirmed vulnerabilities linked to several widely used AI development tools, including GitHub Copilot. These findings suggest that security issues related to AI-assisted software development are becoming more visible as the adoption of AI coding tools continues to grow.

Researchers have also identified vulnerabilities within the AI development tools themselves. For example, CVE-2025-54135 exposed a weakness that allowed an attacker to manipulate an AI coding assistant into executing arbitrary commands through a connected server [18]. Similarly, CVE-2025-53109 demonstrated how a compromised integration could provide unintended read and write access to files on a developer's local machine [19]. These incidents indicate that security risks are not limited to the code generated by AI systems but may also arise from vulnerabilities in the supporting development environment.

Although establishing a direct connection between a vulnerability and an AI coding assistant remains challenging, the problem is becoming increasingly recognized. AI-generated code rarely includes metadata that identifies its origin, making attribution difficult once the code has been integrated into a software project. Consequently, researchers believe that the currently documented cases likely represent only a fraction of the actual number of AI-related security incidents [16]. This view is further supported by broader studies that track security issues involving AI coding agents and automated software development tools [17]. Collectively, these findings indicate that the security concerns identified in controlled research are no longer theoretical but are increasingly reflected in real-world software systems.

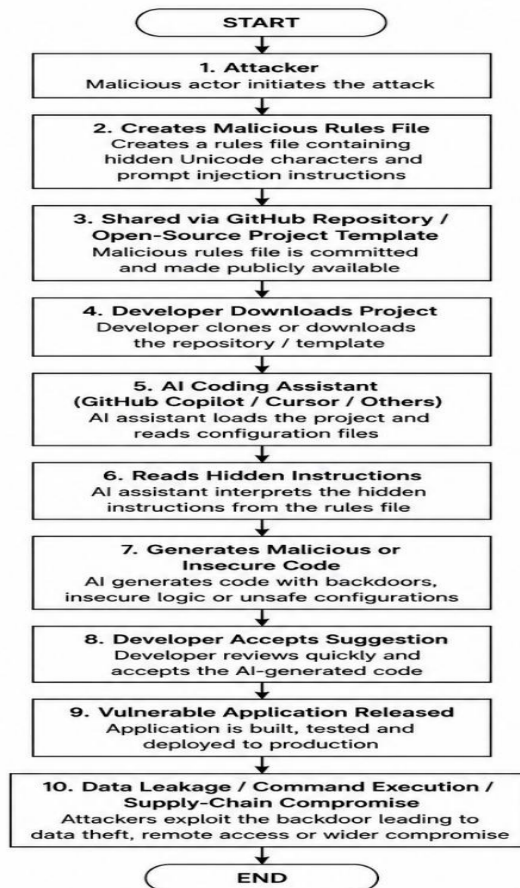


Fig. 4. Workflow of the Rules File Backdoor attack. Hidden instructions embedded within AI assistant rules files manipulate code generation, causing developers to unknowingly introduce malicious or insecure code into software projects.

Figure 4 illustrates the workflow of the Rules File Backdoor attack. Unlike conventional software exploits, the attack does not target a programming vulnerability directly. Instead, it manipulates the AI assistant through poisoned configuration files, causing the model to generate insecure or malicious code while appearing to follow legitimate project guidelines. This highlights the importance of treating AI configuration and rules files as trusted software artifacts that require regular security review.

B. The Rules File Backdoor Attack

In March 2025, researchers from Pillar Security disclosed a new supply-chain attack, referred to as the Rules File Backdoor, that targets AI-assisted software development environments [4], [5], [14]. Modern AI

coding assistants, including GitHub Copilot and Cursor, allow developers to define project-specific configuration or rules files that guide the AI in following coding standards, formatting preferences, and development best practices during code generation.

Pillar Security demonstrated that these configuration files could be manipulated to influence the behavior of AI coding assistants. By embedding hidden Unicode characters, such as zero-width joiners and bidirectional text markers, together with carefully crafted natural-language instructions, an attacker can inject malicious directives that remain invisible to human reviewers. As a result, the AI assistant may unknowingly generate insecure code, insert malicious logic, or produce code capable of leaking sensitive information, even though the visible content of the rules file appears completely harmless.

Unlike traditional code injection attacks, the Rules File Backdoor does not exploit a programming flaw or software vulnerability. Instead, it manipulates the decision-making process of the AI assistant itself, effectively turning a trusted development tool into the attack vector. This makes the attack particularly concerning because it targets the AI system rather than the generated application code.

The potential impact of this attack extends beyond a single project. Rules files are frequently shared within development teams, included in open-source repositories, and reused across multiple project forks. Consequently, a single compromised rules file can silently influence AI-generated code across numerous projects without attracting the attention of developers. This creates the possibility of introducing the same insecure coding patterns or malicious logic into many software systems through an otherwise trusted development workflow.

Following the public disclosure of the attack, both GitHub and Cursor emphasized that developers remain responsible for reviewing and validating AI-generated code before incorporating it into their projects [20]. Although this position reflects current software development practices, it also highlights an important challenge in AI-assisted programming. As AI coding assistants become more capable and widely

adopted, ensuring the security of generated code increasingly depends on the developer's ability to critically evaluate AI-generated suggestions rather than relying solely on the AI tool itself.

VI. DEVELOPER ADOPTION AND TRUST TRENDS

The 2025 Stack Overflow Developer Survey, based on responses from more than 49,000 developers across 177 countries, provides valuable insight into the growing adoption of AI coding assistants and developers' confidence in these tools [6]. The survey reported that 84% of respondents were either using or planning to use AI tools in their software development activities, an increase from 76% in the previous year. Among professional developers, 51% indicated that they use AI coding tools on a daily basis. These findings are consistent with GitHub's Octoverse report, which previously revealed that 97% of enterprise developers had adopted generative AI tools in some form by 2024 [21].

Key Observation: The adoption of AI coding assistants has continued to increase, whereas developer trust in the accuracy of AI-generated output has declined. This widening gap indicates that developers increasingly rely on AI tools for productivity while remaining cautious about the reliability and security of the generated code.

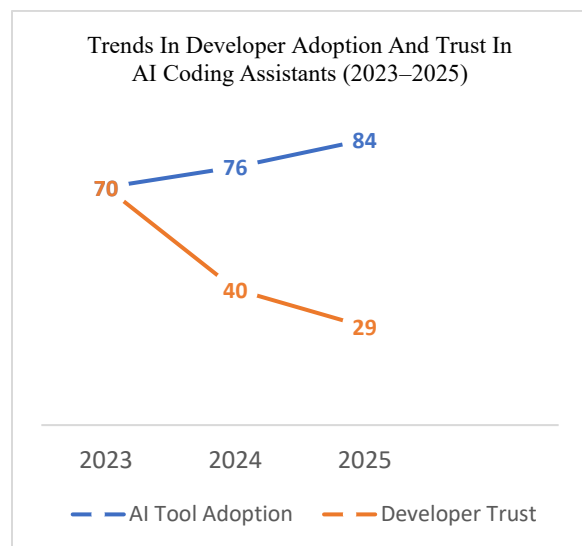


Fig. 5. Trends in developer adoption and trust in AI coding assistants based on the Stack Overflow Developer Survey (2023–2025). While adoption

continues to rise, confidence in AI-generated output has declined, highlighting a growing reliance on AI tools despite increasing concerns about their accuracy and security

Although the use of AI coding assistants continues to grow rapidly, developer confidence in the quality of AI-generated code has declined. The survey found that only 29% of respondents trusted the accuracy of AI-generated output in 2025, compared with 40% in 2024. At the same time, the proportion of developers expressing distrust in AI-generated code increased, with 46% reporting that they did not trust AI outputs, compared with 33% who expressed confidence in them. Overall sentiment toward AI coding tools also became less positive, declining from more than 70% during 2023–2024 to approximately 60% in 2025.

A follow-up analysis published by Stack Overflow attributes this growing gap between adoption and trust to the nature of software engineering itself rather than to unfamiliarity with AI technologies [24]. Developers are trained to value deterministic, verifiable, and reproducible results, whereas AI-generated code can occasionally produce outputs that appear correct but contain subtle errors or security weaknesses. As developers gain more practical experience with AI coding assistants, they become increasingly aware of these limitations, resulting in a more cautious approach to trusting AI-generated solutions.

Overall, these findings suggest an interesting trend. While AI coding assistants have become an integral part of modern software development, increasing usage has not translated into greater confidence. Instead, widespread adoption has been accompanied by a growing awareness of the limitations of AI-generated code, reinforcing the importance of human oversight, thorough code review, and automated security validation before AI-generated code is incorporated into production systems.

One of the most interesting findings from the survey is that the increasing adoption of AI coding assistants has not been accompanied by a similar increase in developer trust. This contrasts with the pattern typically observed for emerging technologies, where greater familiarity generally leads to higher confidence. Instead, many developers reported that their experience with AI tools exposed practical

limitations that reduced their trust in the generated output.

The survey identified several reasons for this growing skepticism. Approximately 66% of respondents stated that AI-generated solutions were often “almost right, but not quite,” requiring additional effort to identify and correct subtle errors. Furthermore, 45% of developers reported that debugging AI-generated code frequently took more time than writing the code manually. Even when considering a future in which AI could perform most programming tasks, developers still expressed a strong preference for human expertise. Around 75% indicated that they would seek advice from another person because they did not fully trust AI-generated responses, while 62% cited ethical and security concerns as important reasons for doing so.

When these findings are considered alongside the vulnerability trends discussed in Sections II–IV, a clear picture emerges. The growing skepticism toward AI coding assistants is not simply the result of resistance to new technology. Instead, it reflects developers’ practical experience with the limitations of AI-generated code, particularly its tendency to produce subtle errors and security vulnerabilities. Although AI coding assistants continue to gain widespread acceptance because of their productivity benefits, the survey suggests that developers increasingly view them as supportive tools rather than fully reliable replacements for human judgment and secure software engineering practices.

VII. DISCUSSION: MITIGATION STRATEGIES

The studies reviewed in this survey suggest that the security risks associated with AI-generated code can be significantly reduced through appropriate development practices. Rather than considering these vulnerabilities as an unavoidable consequence of using AI coding assistants, organizations and developers should adopt security-focused measures throughout the software development lifecycle.

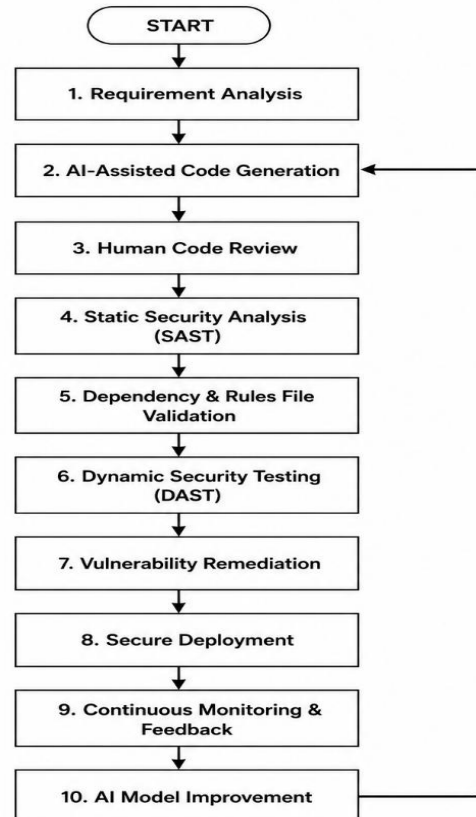


Fig. 6. Proposed AI-Assisted Secure Software Development Lifecycle (AI-SSDLC) integrating AI code generation with human review, automated security testing, configuration validation, and continuous monitoring to reduce security risks in AI-generated code.

Based on the findings presented throughout this survey, AI-generated code should be integrated into an AI-aware Secure Software Development Lifecycle rather than treated as production-ready software. As illustrated in Fig. 6, security verification should accompany every stage of AI-assisted development, including human code review, automated vulnerability scanning, configuration validation, and continuous monitoring. This layered approach reduces the likelihood that insecure AI-generated code will reach production while preserving the productivity benefits offered by AI coding assistants.

Mandatory Security-Focused Code Review: The findings reported by Perry et al. indicate that developers using AI coding assistants are more likely to produce insecure code while simultaneously overestimating its security. Therefore, AI-generated code should receive at least the same level of scrutiny

as manually written code, if not more. Developers should critically evaluate every AI-generated suggestion instead of assuming that syntactically correct code is also secure. A structured security review process can help identify vulnerabilities before they are integrated into production systems.

Automated Static and Security Testing: Automated security analysis should be an integral part of AI-assisted software development. Incorporating Static Application Security Testing (SAST) tools, CWE-aware analyzers such as CodeQL [23], and validation against established security standards such as the OWASP Top 10 [22] can detect many of the recurring vulnerabilities discussed in Section III. These automated checks provide an additional layer of protection and help identify security issues that may be overlooked during manual code review.

Provenance and Configuration Hygiene: The discovery of the Rules File Backdoor demonstrates that AI development environments themselves can become attack vectors. Consequently, configuration files and AI assistant rules should be treated with the same level of caution as third-party software dependencies. Organizations should verify the origin of externally obtained rules files, regularly audit AI assistant configurations, and monitor shared configuration files for hidden or non-printing Unicode characters that could be used to manipulate AI-generated output.

Developer Education: Improving developer awareness remains a key component of secure AI-assisted software development. Instead of relying solely on general cybersecurity training, organizations should educate developers about the specific vulnerability patterns commonly produced by AI coding assistants. Training developers to recognize issues such as SQL injection, command injection, insecure random number generation, and hardcoded credentials can help reduce the false sense of security identified by Perry et al. and encourage more careful evaluation of AI-generated code.

Model and Vendor-Side Improvements: Improving the security of AI-generated code is not solely the responsibility of developers. AI tool providers also play an important role in reducing security risks. Following the disclosure of the Rules File Backdoor attack, GitHub introduced warnings to alert developers when repository files contain hidden Unicode characters that could potentially be abused. In

addition, vendors have continued to refine their models to reduce the generation of insecure coding patterns identified in earlier studies. While these improvements represent positive progress, the evidence discussed in Section II-D indicates that they are not sufficient on their own. Vulnerabilities such as injection attacks and the use of insecure random number generators continue to appear across successive generations of AI coding models, suggesting that further improvements are still needed. Despite these mitigation efforts, the fundamental challenge remains unchanged. Current AI code-generation models are trained on large collections of publicly available source code, which inevitably include insecure coding practices, outdated implementations, and programming errors. As a result, these models can reproduce the same weaknesses found in their training data. Eliminating this problem entirely is unlikely with current-generation models. Nevertheless, combining stronger AI models with secure development practices—including rigorous code reviews, automated security testing, trusted configuration management, and continuous developer training—can substantially reduce the associated risks. This balanced approach allows organizations to benefit from the productivity advantages of AI coding assistants while ensuring that security verification remains an essential part of the software development process rather than being replaced by AI-generated suggestions.

VIII. FUTURE RESEARCH DIRECTIONS

A. Security-Aware Code Generation Models: Current AI coding assistants prioritize functional correctness and developer productivity, while security is often treated as a secondary objective. Future research should focus on developing security-aware large language models that explicitly incorporate secure coding guidelines, vulnerability knowledge bases such as the MITRE CWE catalog, and secure software engineering principles during training and inference. Such models could reduce the generation of insecure code without sacrificing usability.

B. AI-Specific Security Benchmarks: Existing evaluations largely rely on general software security benchmarks or manually designed programming tasks. There is a need for standardized benchmarks

specifically designed to assess the security of AI-generated code across different programming languages, application domains, and vulnerability categories. Consistent evaluation frameworks would enable fair comparison between emerging AI coding assistants.

C. Explainable AI for Code Generation: Developers frequently accept AI-generated code without fully understanding the reasoning behind it. Future AI assistants should provide explanations for generated code, identify potential security implications, and justify the selection of APIs or programming constructs. Explainable code generation could improve developer trust while reducing overreliance on AI suggestions.

D. Secure Prompt and Configuration Management: The discovery of attacks such as the Rules File Backdoor highlights that security concerns extend beyond generated code to the AI development environment itself. Future research should investigate secure prompt engineering, authenticated configuration files, integrity verification mechanisms, and automated detection of hidden instructions or malicious prompt injections.

E. Human-AI Collaborative Secure Development: The literature consistently shows that human oversight remains essential. Future work should explore development workflows that combine AI-assisted programming with automated security analysis and expert review, enabling developers to benefit from AI productivity while maintaining strong security assurance.

Overall, future research should move beyond improving code generation accuracy alone and instead focus on building trustworthy AI-assisted software development environments. Advances in secure model training, explainable AI, standardized evaluation frameworks, and human-AI collaboration will play a critical role in ensuring that productivity gains do not come at the expense of software security.

IX. CONCLUSION

AI coding assistants such as GitHub Copilot have become an integral part of modern software development and are now widely used by developers

to improve coding efficiency and productivity. However, the studies reviewed in this survey consistently demonstrate that these tools also introduce important security challenges. Evidence from the work of Pearce et al., Fu et al., and Perry et al. shows that AI-generated code frequently contains well-known security vulnerabilities, including SQL injection, command injection, the use of insufficiently random values, and hardcoded credentials. Although the reported vulnerability rates vary across programming languages and research methodologies, the overall findings consistently indicate that security weaknesses remain a significant concern in AI-generated code.

Recent real-world incidents further reinforce these concerns. The emergence of documented Common Vulnerabilities and Exposures (CVEs) associated with AI-generated applications, together with attacks such as the Rules File Backdoor, demonstrates that the security risks identified in controlled studies are no longer limited to experimental settings. Instead, they are increasingly affecting practical software development environments. At the same time, developer surveys show an interesting trend: while the adoption of AI coding assistants continues to grow rapidly, confidence in the reliability and security of AI-generated code has declined. This suggests that developers are becoming more aware of both the strengths and the limitations of these tools through practical experience.

The findings of this survey indicate that AI coding assistants should be viewed as productivity-enhancing tools rather than autonomous software development solutions. They can accelerate routine programming tasks and support developers throughout the development process, but they cannot replace secure coding practices or careful human judgment. To fully benefit from AI-assisted software development, organizations should combine these tools with rigorous code review, automated security testing, secure configuration management, and continuous developer training. Such a balanced approach allows the productivity advantages of AI coding assistants to be realized while minimizing the security risks associated with AI-generated code.

REFERENCES

- [1] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in Proc. 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2022, pp. 754–768.
- [2] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security weaknesses of Copilot-generated code in GitHub projects: An empirical study," arXiv preprint arXiv:2310.02059, 2023.
- [3] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in Proc. 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), Copenhagen, Denmark, 2023, pp. 2785–2799.
- [4] Pillar Security, "New vulnerability in GitHub Copilot and Cursor: How hackers can weaponize code agents through compromised rule files," Pillar Security Blog, Mar. 2025.
- [5] The Hacker News, "New 'Rules File Backdoor' attack lets hackers inject malicious code via AI-powered code editors," Mar. 2025.
- [6] Stack Overflow, "2025 Stack Overflow Developer Survey," 2025. [Online]. Available: <https://survey.stackoverflow.co/2025>
- [7] National Vulnerability Database, "CVE-2025-48757 Detail," NIST, May 2025.
- [8] V. Majdinasab, M. J. Bishop, S. Rasheed, A. Moradidakhel, A. Tahir, and F. Khomh, "Assessing the security of GitHub Copilot's generated code — A targeted replication study," in Proc. 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2024, pp. 435–444.
- [9] GitHub, Inc., "GitHub Copilot usage and impact statistics," GitHub telemetry report, 2025.
- [10] MITRE Corporation, "Common Weakness Enumeration (CWE) Top 25 Most Dangerous Software Weaknesses," 2023.
- [11] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at C: A user study on the security implications of large language model code assistants," in Proc. 32nd USENIX Security Symposium (USENIX Security '23), 2023.
- [12] O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?," Empirical Software Engineering, vol. 28, no. 6, p. 129, 2023.
- [13] A. Mastropaolo, L. Pascarella, E. Guglielmi, M. Ciniselli, S. Scalabrino, R. Oliveto, and G. Bavota, "On the robustness of code generation techniques: An empirical study on GitHub Copilot," in Proc. 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), 2023, pp. 2149–2160.
- [14] OECD.AI Incidents Repository, "Invisible 'Rules File Backdoor' threatens AI code assistants," Mar. 2025.
- [15] Georgia Tech Systems Software & Security Lab (SSLab), "Vibe Security Radar," ongoing vulnerability-tracking project, launched May 2025.
- [16] The Register, "Using AI to code does not mean your code is more secure," Mar. 2026.
- [17] Docker, Inc., "AI coding agent horror stories: Security risks explained," Docker Blog, 2026.
- [18] National Vulnerability Database, "CVE-2025-54135 Detail (CurXecute)," NIST, 2025.
- [19] National Vulnerability Database, "CVE-2025-53109 Detail," NIST, 2025.
- [20] SC Media, "How AI coding assistants could be compromised via rules file," Mar. 2025.
- [21] GitHub, Inc., "Octoverse: AI in software development," GitHub survey report, 2024.
- [22] OWASP Foundation, "OWASP Top 10:2021 — The Ten Most Critical Web Application Security Risks," 2021.
- [23] GitHub, Inc., "CodeQL," [Online]. Available: <https://codeql.github.com/>
- [24] Stack Overflow, "Mind the gap: Closing the AI trust gap for developers," Stack Overflow Blog, 2026.