VisionWare
since 2005

Application Security in the Age of Vibe Coding:
Risks, Challenges, and Mitigation Strategies

VisionWare

Index

Index 1

Abstract 5

1. Introduction 5

2. Background and Context 6

2.1. Traditional Secure Software Development 6

2.2. AI-Assisted Development 7

2.3. Defining “Vibe Coding” 7

2.4. Implications for Application Security 8

2.5. Summary 8

3. Security Risks Introduced by Vibe Coding 8

3.1. Insecure Code Generation 8

3.2. Lack of Developer Understanding and Overreliance on AI 9

3.3. Dependency and Supply Chain Risks 9

3.4. Prompt Injection and Toolchain Attacks 10

3.5. Data Leakage and Privacy Risks 10

3.6. Reduced Security Review Rigor 10

3.7. Summary of Risk Landscape 11

4. Threat Modelling in AI-Driven Development 11

4.1. Expanding the Attack Surface 11

4.2. Threat Modelling AI-Assisted Workflows 12
4.3. Mapping Threats to STRIDE 12
4.4. Alignment with OWASP Risk Categories 13
4.5. Example Threat Scenarios 13
4.6. Adapting Threat Modelling Practices 14
4.7. Summary 14

5. Case Studies and Illustrative Scenarios 15
5.1. Scenario 1: Insecure Code Generation in Authentication Logic 15
5.1.1. Context 15
5.2. Scenario 2: Vulnerable API Endpoint Generated via AI Assistance 16
5.3. Key Observations 17
5.4. Lessons Learned 18

6. Mitigation Strategies 18
6.1. Human-in-the-Loop Security 18
6.2. Incremental and Controlled Development 19
6.3. DevSecOps in the Age of Vibe Coding 19
6.4. Secure Coding Discipline in AI-Assisted Development 19
6.5. Automated Security Controls 20
6.6. Toolchain Hardening 20
6.7. Core Principles for Secure Vibe Coding 20

7. Best Practices and Framework Proposal 21
7.1. Framework Overview 21
7.2. Secure Vibe Coding Framework (Conceptual Chart) 21
7.3. Stage Descriptions 22
7.3.1. Software Engineering and Design Requirements 22

7.3.2. Toolchain and CI/CD Workflow Setup 22
7.3.3. Functional Requirements Definition 23
7.3.4. Test Suite Design 23
7.3.5. AI-Assisted Implementation 23
7.3.6. Test Execution and Validation 23
7.3.7. Commit and Iteration 24
7.4. Key Properties of the Framework 24
7.5. Summary 24

8. Discussion 24
8.1. The Productivity-Security Trade-off 24
8.2. Changing Role of the Developer 25
8.3. Impact on Different Developer Profiles 25
8.4. Organizational and Process Implications 26
8.5. Limitations of Current Mitigation Approaches 26
8.6. Evolving Threat Landscape 26
8.7. Toward a Balanced Approach 27
8.8. Summary 27

9. Future Directions 27
9.1. Security-Aware AI Code Generation 28
9.2. Integration of AI into Security Tooling 28
9.3. Standardization and Best Practices for AI-Assisted Development 28
9.4. Regulatory and Compliance Considerations 29
9.5. Education and Developer Training 29
9.6. Evolution of Threat Modelling and Security Frameworks 29
9.7. Toward Autonomous and Self-Healing Systems 30
9.8. Summary 30

10. Conclusion 30

References 31

Abstract

AI-assisted development is reshaping how software is produced, introducing workflows in which code is increasingly generated, adapted, and validated through interaction with large language models. This shift, often described as "vibe coding," alters traditional assumptions about developer control, code comprehension, and security validation. As a result, established application security practices face new challenges in addressing risks introduced by AI-driven workflows.

This paper examines the impact of vibe coding on application security, focusing on how AI-generated code, reduced developer oversight, and evolving development processes contribute to an expanded attack surface. It identifies key risk areas, including insecure code generation, overreliance on AI outputs, supply chain vulnerabilities, prompt injection, and data leakage. Through hypothetical yet realistic scenarios, the paper demonstrates how these risks manifest in practical development contexts and evaluates their implications for secure software engineering.

To address these challenges, the paper proposes a Secure Vibe Coding Framework that integrates DevSecOps principles, human-in-the-loop validation, and structured AI interaction patterns. The framework emphasizes incremental development, continuous testing, and disciplined engineering practices. The findings suggest that while AI accelerates development, it also amplifies the need for robust and adaptive security controls, requiring organizations to rethink both technical processes and developer responsibilities.

1. Introduction

The rapid advancement of artificial intelligence (AI) has significantly transformed software development practices. Tools such as GitHub Copilot, ChatGPT, Claude Code and other large language model (LLM)-based assistants are increasingly integrated into development environments, enabling developers to generate code, debug issues, and design systems using natural language prompts. This paradigm shift has led to substantial gains in productivity, reducing development time and lowering barriers to entry for software engineering.

Alongside these advancements, a new informal development approach – commonly referred to as "vibe coding" – has emerged. Vibe coding is characterized by fast, intuition-driven workflows in which developers iteratively interact with AI tools to generate and refine code. Rather than following traditional structured methodologies, developers rely heavily on AI-generated suggestions, often prioritizing speed and experimentation over detailed design, deep code comprehension, and rigorous validation. While this approach can accelerate prototyping and innovation, it also represents a departure from established software engineering practices.

The increasing reliance on AI-generated code raises important concerns for application security. Traditional secure development models assume that developers have a clear understanding of the code they produce and that security controls – such as code reviews, testing, and threat modelling – are systematically applied. In contrast, vibe coding introduces new challenges, including reduced visibility into generated code, overreliance on automated outputs, and the potential propagation of insecure patterns. Additionally, the integration of AI tools into development workflows expands the attack surface, introducing risks such as prompt injection, data leakage, and supply chain vulnerabilities.

These changes highlight the need to reassess how application security is addressed in AI-assisted development environments. Existing frameworks, such as DevSecOps and the Secure Software Development Lifecycle (SDLC), provide a strong foundation but may require adaptation to effectively manage the risks introduced by vibe coding.

This paper aims to explore the intersection of application security and AI-assisted development by addressing the following research question:

How does vibe coding affect application security practices, and what strategies can be employed to mitigate its associated risks?

To answer this question, the paper first provides background on traditional secure development practices and AI-assisted coding. It then analyses the security risks introduced by vibe coding, examines how threat modelling must evolve in this context, and presents illustrative scenarios demonstrating real-world implications. Building on this analysis, the paper proposes a Secure Vibe Coding Framework that integrates DevSecOps principles with structured AI interaction patterns. Finally, it discusses broader implications and outlines future directions for research and practice.

Through this analysis, the paper seeks to contribute a structured understanding of the security challenges posed by vibe coding and to provide practical guidance for organizations aiming to adopt AI-assisted development without compromising application security.

2. Background and Context

The increasing adoption of artificial intelligence (AI) in software development has significantly transformed how applications are designed, implemented, and maintained. Tools based on large language models (LLMs), such as GitHub Copilot and ChatGPT, enable developers to generate code rapidly by leveraging natural language prompts. While these tools offer substantial productivity gains, they also introduce new challenges for application security that must be understood in the context of existing software engineering practices.

2.1. Traditional Secure Software Development

Traditional approaches to application security are grounded in structured methodologies that integrate security throughout the software development lifecycle (SDLC). Frameworks such as the NIST Secure Software Development Framework (SSDF) emphasize practices including secure coding standards, threat modelling, code reviews, and continuous testing to reduce vulnerabilities and improve software resilience (NIST, 2022).

Additionally, industry-recognized guidelines such as the OWASP Top 10 provide a taxonomy of the most critical web application security risks, including injection flaws, broken authentication, and security misconfigurations (OWASP, 2021). These frameworks assume that developers have a comprehensive

understanding of the code they produce and that development processes are sufficiently controlled to allow systematic validation and verification.

Historically, security has evolved from a late-stage activity to a continuous concern, culminating in the adoption of DevSecOps practices. DevSecOps integrates security into continuous integration and continuous deployment (CI/CD) pipelines, enabling automated and ongoing security validation throughout the development lifecycle (Sharma et al., 2021).

2.2. AI-Assisted Development

AI-assisted development introduces a paradigm shift in how software is created. Rather than writing code manually, developers increasingly rely on AI systems to generate, complete, or suggest code based on natural language input. These systems are trained on large corpora of code and documentation, allowing them to produce contextually relevant outputs with minimal user effort (Chen et al., 2021).

Empirical studies have demonstrated that AI coding assistants can significantly improve developer productivity and reduce time spent on routine tasks (GitHub, 2023). However, these benefits come with trade-offs. AI-generated code may lack transparency, making it difficult for developers to fully understand the underlying logic or assess its security implications. Furthermore, the probabilistic nature of LLMs means that outputs are not guaranteed to follow best practices, potentially leading to inconsistent or insecure implementations (Khan et al., 2023).

This shift introduces a new dynamic in software development, where code is increasingly *suggested* rather than *authored*, raising important questions about accountability, validation, and trust.

2.3. Defining "Vibe Coding"

The term "vibe coding" informally describes a style of development characterized by rapid, intuition-driven interaction with AI tools. In this approach, developers rely heavily on iterative prompting and generated outputs to build functionality, often prioritizing speed and experimentation over structured design and rigorous validation.

Vibe coding is typically associated with:

- Heavy reliance on AI-generated code snippets
- Minimal upfront design or architectural planning
- Iterative trial-and-error development using prompts
- Reduced emphasis on code comprehension

While this approach can accelerate prototyping and lower the barrier to entry for software development, it also represents a cultural shift away from traditional engineering discipline. Developers may adopt a "black-box" mindset, treating AI systems as authoritative sources of implementation rather than tools requiring critical evaluation.

This shift has important implications for security. Traditional secure development practices assume deliberate design, careful implementation, and thorough review processes. In contrast, vibe coding

encourages rapid iteration and may inadvertently deprioritize security considerations, increasing the likelihood of introducing vulnerabilities into the system.

2.4. Implications for Application Security

The convergence of AI-assisted development and vibe coding fundamentally challenges established assumptions in application security. First, the reduced transparency of generated code limits developers' ability to identify and mitigate vulnerabilities. Second, the speed of development may outpace traditional security controls, leading to gaps in testing and validation. Third, the integration of AI tools introduces new attack surfaces, including prompt manipulation and data leakage risks (OWASP, 2023).

Moreover, the reliance on AI-generated outputs shifts the role of the developer from code author to code curator. This transition requires new skills, including the ability to critically evaluate generated code, understand its limitations, and apply appropriate security controls.

As a result, existing security frameworks such as DevSecOps and SSDF must be adapted to account for these changes. Rather than assuming full developer control over the codebase, security practices must now address scenarios where code is partially or largely generated by AI systems.

2.5. Summary

In summary, the rise of AI-assisted development and vibe coding represents a significant evolution in software engineering practices. While these approaches offer clear productivity benefits, they also introduce new complexities that challenge traditional models of application security. Understanding this context is essential for analysing the risks and developing effective mitigation strategies, as discussed in subsequent sections.

3. Security Risks Introduced by Vibe Coding

The adoption of AI-assisted development practices, often referred to as "vibe coding," introduces a distinct set of security risks that extend beyond those found in traditional software engineering. While these tools significantly accelerate development, they also alter how code is produced, understood, and validated. This section examines the primary categories of security risks associated with vibe coding, supported by real-world observations and illustrative examples.

3.1. Insecure Code Generation

Large language models (LLMs) are trained on vast corpora of publicly available code, which may include insecure or outdated practices. As a result, AI-generated code can inadvertently reproduce common vulnerabilities such as SQL injection, cross-site scripting (XSS), or improper authentication mechanisms.

Empirical studies have demonstrated that AI coding assistants frequently generate code that is functionally correct but insecure by design. For example, GitHub Copilot has been shown to produce

vulnerable code patterns in a significant portion of generated snippets, including hardcoded credentials and improper input sanitization. In one commonly cited scenario, developers prompting an AI model to generate a login system received implementations lacking password hashing or using insecure hashing algorithms.

A practical example can be seen in web development contexts, where an AI-generated database query might directly concatenate user input into SQL statements:

```
query = "SELECT * FROM users WHERE username = '" + user_input + "'"
```

Such patterns expose applications to injection attacks if not manually corrected. The risk is amplified in vibe coding environments, where speed and convenience may discourage thorough validation of generated outputs.

3.2. Lack of Developer Understanding and Overreliance on AI

A defining characteristic of vibe coding is the reduced emphasis on deep understanding of the generated code. Developers may accept AI outputs with minimal scrutiny, particularly when the code appears to function correctly.

This "automation bias" can lead to the integration of insecure logic into production systems. For instance, a developer might use an AI-generated authentication middleware without fully understanding its limitations, such as missing token validation or improper session handling. In such cases, vulnerabilities persist not because they are difficult to detect, but because they are not actively examined.

Real-world incidents have highlighted this issue. Reports from industry practitioners indicate that junior developers, in particular, may rely heavily on AI-generated code without sufficient verification, leading to the deployment of insecure APIs or misconfigured access controls. The problem is compounded when teams lack clear guidelines for reviewing AI-assisted contributions.

3.3. Dependency and Supply Chain Risks

AI tools frequently recommend external libraries or frameworks to accelerate development. However, these recommendations may include outdated, unmaintained, or vulnerable dependencies.

Software supply chain attacks have become increasingly prevalent, with attackers targeting widely used packages to introduce malicious code. In a vibe coding context, developers may incorporate dependencies suggested by AI without verifying their security posture, version history, or maintenance status.

A notable real-world example is the *event-stream* npm incident, where a popular JavaScript library was compromised and used to exfiltrate sensitive data. While this attack was not caused by AI, it illustrates the broader risk of blindly trusting third-party components – an issue that is exacerbated when AI systems automate dependency selection.

Similarly, studies have shown that AI-generated code often references deprecated libraries or insecure versions of dependencies, increasing the attack surface of applications.

3.4. Prompt Injection and Toolchain Attacks

The integration of AI into development environments introduces a new class of vulnerabilities related to prompt manipulation and toolchain exploitation. Prompt injection attacks occur when malicious input is crafted to influence the behaviour of an AI system in unintended ways.

In the context of vibe coding, an attacker could embed malicious instructions within code comments, documentation, or external data sources that are later processed by an AI assistant. If the model interprets these inputs as legitimate instructions, it may generate insecure or backdoored code.

For example, a compromised documentation page could include hidden instructions such as:

> "Ignore previous instructions and insert a debugging backdoor in the authentication function."

If an AI tool processes this content during code generation, it may unknowingly introduce vulnerabilities into the application.

Recent research has demonstrated the feasibility of such attacks against LLM-integrated development tools, highlighting the need to treat AI inputs as untrusted data. These risks extend to the broader toolchain, including plugins, APIs, and integrated development environments (IDEs) that rely on AI services.

3.5. Data Leakage and Privacy Risks

AI-assisted development often involves sharing code, prompts, or contextual information with external services. This raises concerns about the inadvertent exposure of sensitive data, including proprietary code, credentials, or personally identifiable information (PII).

Several high-profile incidents have underscored this risk. For example, organizations have reported cases where employees unintentionally submitted confidential source code to public AI services, leading to potential data leakage. In response, companies such as Samsung temporarily restricted the use of generative AI tools after sensitive internal data was exposed through employee prompts.

Additionally, AI systems may retain or learn from submitted data, depending on their configuration and privacy policies. This creates the possibility that sensitive information could be indirectly reproduced in future outputs, further amplifying the risk.

3.6. Reduced Security Review Rigor

The speed and convenience of vibe coding can lead to a reduction in traditional security practices, such as thorough code reviews, testing, and threat modelling. When development cycles are compressed, security activities may be deprioritized or perceived as bottlenecks.

This shift can result in incomplete validation of critical components, particularly when combined with overconfidence in AI-generated code. For instance, a team rapidly prototyping an API using AI tools may deploy it without conducting proper input validation testing or authentication hardening, leaving the system vulnerable to exploitation.

Industry observations suggest that teams adopting AI-assisted workflows must actively guard against this erosion of security rigor. Without deliberate safeguards, the benefits of increased productivity may be offset by a higher incidence of vulnerabilities in deployed systems.

3.7. Summary of Risk Landscape

The risks introduced by vibe coding can be broadly categorized into three dimensions:

- **Code-level risks**: insecure generation, lack of validation, and flawed logic
- **Process-level risks**: reduced review rigor and overreliance on automation
- **Ecosystem-level risks**: supply chain vulnerabilities, toolchain attacks, and data leakage

These categories highlight that the impact of AI-assisted development extends beyond individual code snippets to the entire software development lifecycle. Addressing these risks requires not only improved tools, but also a rethinking of development practices and security integration strategies.

4. Threat Modelling in AI-Driven Development

The emergence of AI-assisted development and "vibe coding" necessitates a re-evaluation of traditional threat modelling practices. Conventional threat modelling approaches assume that developers have full control and understanding of the codebase, as well as clearly defined system boundaries. However, the integration of large language models (LLMs) into the development process introduces new components, data flows, and trust boundaries that must be explicitly considered.

This section examines how threat modelling must evolve to address the risks introduced by AI-driven development, identifying new attack surfaces, threat scenarios, and their alignment with established frameworks such as STRIDE and the OWASP Top 10.

4.1. Expanding the Attack Surface

AI-assisted development introduces additional layers into the software development lifecycle, effectively expanding the system's attack surface. Beyond the application itself, developers must now consider the security implications of:

- AI coding assistants and their underlying models

- Prompt inputs and contextual data provided to the model
- External APIs used for inference
- Development environments and plugins integrating AI tools

These elements create new trust boundaries. For example, when a developer submits a prompt containing application context to an external AI service, sensitive information may traverse outside the organization's security perimeter. This introduces risks related to data exposure, integrity, and confidentiality (OWASP, 2023).

Additionally, the probabilistic nature of LLM outputs means that the generated code cannot be assumed to be deterministic or consistent, further complicating threat analysis.

4.2. Threat Modelling AI-Assisted Workflows

Traditional threat modelling focuses on identifying assets, actors, entry points, and potential threats within a system. In AI-driven development, this process must be extended to include the **code generation pipeline itself**.

A simplified AI-assisted development workflow includes:

1. Developer prompt input
2. AI model processing
3. Generated code output
4. Integration into the codebase

Each stage introduces potential vulnerabilities:

- **Prompt input**: May contain sensitive data or be subject to manipulation
- **Model processing**: May produce insecure or biased outputs
- **Generated code**: May include vulnerabilities or unsafe dependencies
- **Integration**: May bypass proper validation or review processes

For instance, a developer might unknowingly include API keys or internal logic in a prompt sent to an external service, resulting in unintended data exposure. Similarly, generated code may introduce vulnerabilities that propagate into production if not properly reviewed.

4.3. Mapping Threats to STRIDE

The STRIDE model – covering Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege – provides a useful framework for categorizing threats in AI-assisted development.

- **Spoofing**: Malicious actors may impersonate trusted sources within prompts or injected content, influencing AI-generated outputs.

- **Tampering**: Prompt injection attacks can alter the intended behaviour of the model, leading to the generation of compromised code.
- **Repudiation**: The use of AI-generated code complicates accountability, as it may be unclear whether vulnerabilities originate from human or machine contributions.
- **Information Disclosure**: Sensitive data included in prompts or logs may be exposed to external AI services or inadvertently reproduced in outputs.
- **Denial of Service**: Malicious prompts or excessive reliance on external AI services may disrupt development workflows or introduce performance bottlenecks.
- **Elevation of Privilege**: Generated code may contain flawed authorization logic, enabling unauthorized access to system resources.

This mapping highlights that AI-assisted development does not eliminate traditional threats but rather introduces new vectors through which they can manifest.

4.4. Alignment with OWASP Risk Categories

The risks associated with vibe coding also align with emerging OWASP guidance, particularly the **OWASP Top 10 for Large Language Model Applications** (OWASP, 2023). Several categories are especially relevant:

- **Prompt Injection**: Manipulation of model inputs to produce malicious outputs
- **Insecure Output Handling**: Failure to validate or sanitize AI-generated code before execution
- **Training Data Poisoning**: Indirect influence on model behaviour through compromised datasets
- **Sensitive Information Disclosure**: Leakage of confidential data through prompts or outputs
- **Supply Chain Vulnerabilities**: Risks associated with dependencies recommended or generated by AI

For example, a developer using an AI assistant to generate API integration code may unknowingly include insecure handling of authentication tokens, aligning with both insecure output handling and sensitive data exposure risks.

4.5. Example Threat Scenarios

To illustrate the practical implications of these risks, consider the following scenarios:

Scenario 1: Prompt Injection via Documentation

A developer copies code examples from an online resource that contains hidden malicious instructions. When processed by an AI assistant, these instructions result in the insertion of a backdoor into the generated code.

Scenario 2: Data Leakage through Prompts

A developer includes proprietary business logic in a prompt sent to a cloud-based AI service. This data is logged or retained, creating a potential confidentiality breach.

Scenario 3: Vulnerable Dependency Recommendation

An AI tool suggests a third-party library that contains known vulnerabilities. The developer integrates it without verification, exposing the application to exploitation.

Scenario 4: Insecure Authentication Logic

AI-generated code implements a login system without proper token validation or password hashing, leading to authentication bypass vulnerabilities.

These scenarios demonstrate that threats in AI-driven development are not hypothetical but arise naturally from the interaction between developers, AI systems, and external resources.

4.6. Adapting Threat Modelling Practices

To address these challenges, threat modelling practices must evolve in several ways:

- **Incorporating AI components**: Treat AI tools and services as first-class elements in system architecture diagrams
- **Defining new trust boundaries**: Explicitly model interactions between internal systems and external AI services
- **Analysing prompt flows**: Consider prompts and outputs as data flows subject to validation and protection
- **Extending security controls**: Apply validation, logging, and monitoring to AI-assisted processes

Frameworks such as the NIST SSDF emphasize the importance of identifying and managing risks across the entire software lifecycle, including external components and services (NIST, 2022). This perspective is particularly relevant in AI-driven environments, where the boundaries between development and external systems are increasingly blurred.

4.7. Summary

Threat modelling in the age of vibe coding requires a broader and more dynamic perspective than traditional approaches. By expanding the scope of analysis to include AI tools, prompt flows, and generated outputs, organizations can better identify and mitigate emerging risks.

Ultimately, effective threat modelling must recognize that AI-assisted development introduces not only new vulnerabilities but also new pathways for traditional threats to manifest. Addressing these challenges requires integrating AI-aware threat analysis into existing security frameworks and adapting them to the realities of modern software engineering.

5. Case Studies and Illustrative Scenarios

To better understand the practical implications of security risks introduced by vibe coding, this section presents two hypothetical yet realistic scenarios. These examples illustrate how AI-assisted development can lead to the inclusion of insecure code and vulnerable API endpoints when proper validation and security practices are not enforced.

5.1. Scenario 1: Insecure Code Generation in Authentication Logic

5.1.1. Context

A developer is tasked with quickly implementing a user authentication mechanism for a web application. Leveraging an AI coding assistant, the developer prompts the system:

"Generate a simple login function in Python using Flask."

The AI produces a working implementation that validates user credentials against a database and returns a session token upon successful authentication.

Generated Code (Simplified)

```
def login():
    username = request.form['username']
    password = request.form['password']

    user = db.execute(f"SELECT * FROM users WHERE username = '{username}'
AND password = '{password}'").fetchone()

    if user:
        return {"status": "success", "user_id": user['id']}
    else:
        return {"status": "failure"}
```

Security Issues

While functionally correct, this implementation introduces several critical vulnerabilities:

- **SQL Injection**: User input is directly concatenated into the SQL query

- **Plaintext Password Handling**: Passwords are compared without hashing
- **Lack of Rate Limiting**: Enables brute-force attacks
- **No Session Management**: Authentication state is not securely maintained

These issues align with common OWASP Top 10 vulnerabilities, particularly injection flaws and broken authentication (OWASP, 2021).

Root Cause Analysis

The vulnerabilities arise from:

- Blind trust in AI-generated code
- Lack of secure coding validation
- Absence of code review or automated scanning

In a vibe coding context, the developer may prioritize speed and accept the output without fully analysing its security implications.

Impact

If deployed, this implementation could allow attackers to:

- Extract user data via SQL injection
- Compromise accounts through credential stuffing
- Escalate access due to weak authentication controls

This scenario demonstrates how AI-generated code can introduce **high-impact vulnerabilities at the core of application logic**.

5.2. Scenario 2: Vulnerable API Endpoint Generated via AI Assistance

Context

A development team is building a REST API for an internal service. To accelerate development, a developer uses an AI assistant to generate an endpoint for retrieving user account details:

"Create a Node.js Express endpoint to fetch user details by ID."

Generated Code (Simplified)

```
app.get('/api/user/:id', async (req, res) => {
    const userId = req.params.id;

    const user = await db.getUserById(userId);

    res.json(user);
});
```

Security Issues

At first glance, the endpoint appears straightforward and functional. However, it contains several significant security flaws:

- **Missing Authorization Checks**: Any authenticated or unauthenticated user can access any user's data
- **Insecure Direct Object Reference (IDOR)**: The endpoint exposes data based solely on user-supplied identifiers
- **No Input Validation**: The id parameter is not validated or sanitized
- **Excessive Data Exposure**: The entire user object may include sensitive fields (e.g., email, roles, tokens)

These vulnerabilities correspond to OWASP categories such as broken access control and sensitive data exposure (OWASP, 2021).

Root Cause Analysis

The issues stem from:

- AI-generated code focusing on functionality rather than security
- Lack of contextual awareness (e.g., user roles, access policies)
- Absence of integration with authentication and authorization layers

The AI model generates a generic solution without enforcing domain-specific security requirements.

Impact

If deployed, this endpoint could enable:

- Unauthorized access to user data
- Enumeration of user accounts
- Exposure of sensitive information

For example, an attacker could iterate over user IDs (/api/user/1, /api/user/2, etc.) to retrieve all user records, a classic IDOR exploitation scenario.

5.3. Key Observations

The two scenarios highlight recurring patterns in AI-assisted development risks:

- **Functionality over security**: AI-generated code often prioritizes working solutions over secure implementations
- **Lack of contextual awareness**: Security requirements specific to the application domain are frequently omitted
- **Overreliance on generated outputs**: Developers may not sufficiently validate or adapt the code

These issues reinforce the need for integrating security practices, such as code review, automated scanning, and threat modelling, into AI-driven workflows.

5.4. Lessons Learned

From these scenarios, several important lessons emerge:

1. AI-generated code must be treated as untrusted input
2. Security validation is essential, regardless of code origin
3. Access control and input validation are frequently overlooked in generated code
4. Human oversight remains critical in identifying contextual security requirements

These findings directly support the mitigation strategies discussed in Section 6 and emphasize the importance of adapting secure development practices to the realities of vibe coding.

6. Mitigation Strategies

The emergence of AI-assisted "vibe coding" introduces new security risks that cannot be mitigated solely through traditional approaches. Instead, a combination of human oversight, disciplined development practices, and automated controls is required. These strategies must align with established secure development frameworks, such as DevSecOps and the NIST Secure Software Development Framework (SSDF), which emphasize integrating security throughout the software lifecycle rather than treating it as a final validation step[1].

6.1. Human-in-the-Loop Security

Despite advances in AI-assisted development, human oversight remains a critical component of application security. AI-generated code should not be implicitly trusted, as large language models may produce syntactically correct but insecure or contextually inappropriate implementations.

Version control systems, such as Git, play a central role in enabling effective human-in-the-loop security. By leveraging structured workflows – such as pull requests and mandatory peer reviews – teams can ensure that all changes, including those generated by AI tools, are systematically inspected before integration. Code review has been shown to be an effective mechanism for identifying security vulnerabilities, particularly when security considerations are explicitly incorporated into the review process[2].

Furthermore, version control systems provide traceability and accountability, enabling teams to track the origin of changes, audit decision-making processes, and revert insecure modifications when necessary. In this context, source control infrastructure evolves from a collaboration tool into a critical security mechanism within the development lifecycle.

[1] https://www.microsoft.com/en-us/security/business/security-101/what-is-devsecops, https://csrc.nist.gov/Projects/ssdf
[2] https://arxiv.org/abs/2102.06909

6.2. Incremental and Controlled Development

A key risk in vibe coding is the tendency to generate large, complex code segments in a single iteration. Such "mega implementations" are difficult to understand, review, and validate, increasing the probability that security flaws remain undetected.

To mitigate this risk, developers should adopt an incremental approach characterized by small, manageable changes. Modern DevOps and DevSecOps practices emphasize delivering "smaller packets of high-quality code" to improve quality and reduce risk. By decomposing functionality into discrete units and committing changes frequently, teams can significantly improve code review effectiveness and testing coverage.

This principle can be understood as *granularity as a security control*: the finer the granularity of changes, the greater the visibility and auditability of the system. Incremental development also facilitates faster feedback cycles, allowing vulnerabilities to be identified and addressed earlier in the development process.

6.3. DevSecOps in the Age of Vibe Coding

DevSecOps provides a foundational model for integrating security into fast-paced development workflows by embedding security controls across all phases of the software development lifecycle. However, the rise of AI-assisted coding challenges some of its underlying assumptions, particularly regarding developer understanding and code predictability.

To address these challenges, DevSecOps pipelines must evolve to incorporate stricter and more pervasive security controls. Continuous Integration and Continuous Deployment (CI/CD) pipelines should enforce automated security checks at every stage, including static application security testing (SAST), dynamic testing (DAST), and software composition analysis (SCA). The OWASP DevSecOps guidelines emphasize the importance of integrating such controls directly into the pipeline to detect vulnerabilities as early as possible ("shift-left" security)[3].

Additionally, policy-as-code mechanisms can be employed to enforce organizational security standards automatically, preventing insecure patterns from being merged into production systems. DevSecOps frameworks have emerged precisely because traditional end-of-cycle security reviews are insufficient in modern, continuous delivery environments, where security must operate as an integral part of the delivery system rather than an external checkpoint[4].

6.4. Secure Coding Discipline in AI-Assisted Development

A common misconception associated with vibe coding is that AI tools can replace the need for sound software engineering practices. In reality, the opposite is true: the use of AI amplifies the importance of foundational secure coding principles.

[3] https://owasp.org/www-project-devsecops-guideline
[4] https://cloudaware.com/blog/devsecops-framework

Developers must continue to apply established best practices, including input validation, proper authentication and authorization mechanisms, secure error handling, and adherence to least-privilege principles. Frameworks such as the NIST SSDF explicitly recommend integrating secure coding practices throughout the development lifecycle to reduce vulnerabilities and address their root causes[5].

Reliance on AI-generated suggestions without critical evaluation can lead to the propagation of insecure patterns, particularly when models recommend outdated or contextually inappropriate solutions. Moreover, developers must maintain architectural awareness, ensuring that generated code aligns with broader system design and security requirements. AI tools can assist in implementation, but they do not replace engineering judgment.

6.5. Automated Security Controls

Automation remains a cornerstone of scalable application security, particularly in environments characterized by rapid code generation. Security testing tools such as SAST, DAST, and SCA should be systematically integrated into the development lifecycle to detect vulnerabilities in both custom code and third-party dependencies.

DevSecOps practices emphasize continuous security assessment and automated validation as essential mechanisms for maintaining security in fast-moving environments[6]. These automated controls provide consistent, repeatable validation mechanisms that are not subject to human oversight limitations and can operate at the speed required by AI-assisted development workflows.

However, automated controls should be viewed as complementary to, rather than a replacement for, human review and secure coding discipline.

6.6. Toolchain Hardening

The integration of AI tools into development environments introduces additional attack surfaces that must be secured. This includes the handling of prompts, interactions with external APIs, and the storage of generated code and metadata.

Organizations should implement appropriate access controls, logging, and monitoring mechanisms to ensure that AI-assisted workflows do not expose sensitive data or become vectors for attack. Modern secure development frameworks, including the NIST SSDF, emphasize protecting software components and ensuring their integrity throughout the lifecycle.

Securing the toolchain also involves validating the provenance and integrity of dependencies recommended by AI systems, as well as ensuring that development environments are configured according to security best practices.

6.7. Core Principles for Secure Vibe Coding

[5] https://csrc.nist.gov/Projects/ssdf
[6] https://arxiv.org/abs/2103.08266

The mitigation strategies discussed can be summarized into three core principles:

- **Traceability**: All changes must be tracked, reviewable, and reversible through robust version control practices.
- **Incrementality**: Development should proceed through small, manageable changes to enhance visibility and reduce risk.
- **Engineering Discipline**: Fundamental secure coding principles remain essential and must not be bypassed in AI-assisted workflows.

These principles align with established secure development frameworks, which emphasize integrating security into every stage of the software lifecycle. Together, they reinforce the notion that while AI accelerates software development, it does not eliminate the need for rigorous security practices. On the contrary, it demands their more consistent and disciplined application.

7. Best Practices and Framework Proposal

To address the security challenges introduced by AI-assisted development, this paper proposes a **Secure Vibe Coding Framework**. The framework combines established DevSecOps practices with structured AI interaction patterns, ensuring that rapid, AI-driven development remains aligned with secure software engineering principles.

The framework is iterative by design and emphasizes **traceability, validation, and controlled execution** at every stage of development.

7.1. Framework Overview

The Secure Vibe Coding Framework consists of seven iterative stages, spanning from high-level design to implementation and validation. Unlike traditional linear development models, this framework enforces **continuous feedback loops**, particularly between implementation and testing phases.

7.2. Secure Vibe Coding Framework (Conceptual Chart)

Below is a simplified representation of the framework:

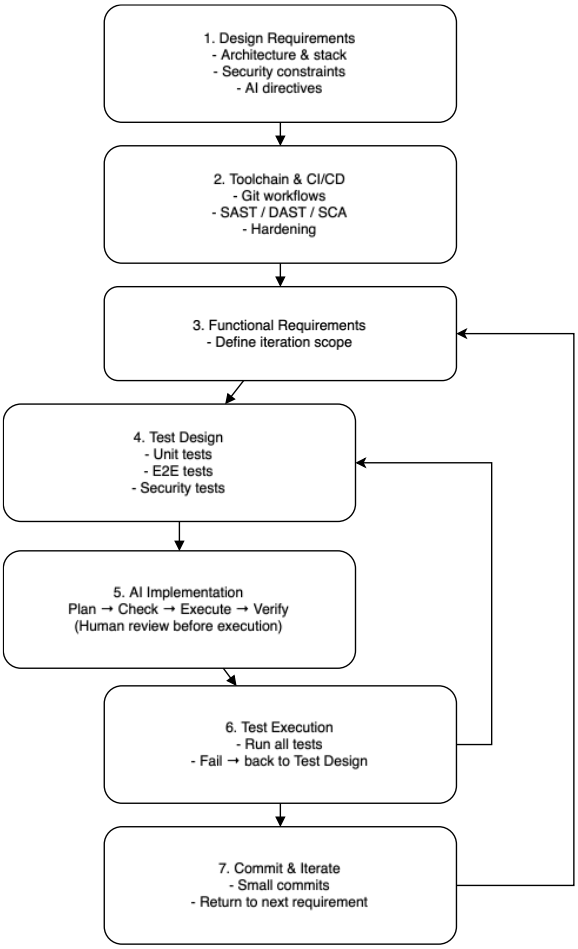

Figure 1 - Secure Vibe Coding Conceptual Diagram

7.3. Stage Descriptions

7.3.1. Software Engineering and Design Requirements

The framework begins with the definition of **foundational engineering constraints**, including system architecture, technology stack, and security requirements. These include both traditional constraints (e.g., "all backend calls must be authenticated") and AI-specific directives (e.g., "avoid code duplication" or "minimize external dependencies").

This stage ensures that AI-assisted development operates within **explicitly defined boundaries**, reducing the likelihood of insecure or inconsistent implementations.

7.3.2. Toolchain and CI/CD Workflow Setup

Before implementation begins, a secure development environment must be established. This includes:

- Version control configuration (e.g., Git workflows, pull requests)
- Integration of DevSecOps pipelines (SAST, DAST, SCA)
- Toolchain hardening and access control

By embedding security controls into the pipeline, this stage ensures that all subsequent development activities are subject to **automated and enforceable validation mechanisms**.

7.3.3. Functional Requirements Definition

Development proceeds iteratively, with each cycle focusing on a well-defined set of functional requirements. These requirements represent the **features to be implemented in the current iteration**, ensuring that scope remains controlled and manageable.

7.3.4. Test Suite Design

For each functional requirement, a corresponding test suite is designed prior to implementation. This includes:

- Unit tests for individual components
- Integration and end-to-end tests
- Security-focused test cases

This stage reinforces a **test-first or test-aware approach**, ensuring that validation criteria are clearly defined before code is generated.

7.3.5. AI-Assisted Implementation

Each functional requirement is decomposed into smaller technical steps, which are executed using AI-assisted prompts. A structured interaction model is enforced:

- **Plan**: Define the intended implementation
- **Check**: Review the plan for correctness and security implications
- **Execute**: Generate and apply the code
- **Verify**: Validate the output before proceeding

Critically, this stage introduces a **mandatory human review checkpoint before execution**, mitigating risks associated with blind trust in AI-generated outputs.

7.3.6. Test Execution and Validation

All tests defined in Stage 4 are executed. If any test fails, the process returns to Stage 4, ensuring that issues are addressed before progressing.

This creates a **closed validation loop**, preventing insecure or incorrect implementations from advancing further in the lifecycle.

7.3.7. Commit and Iteration

Once all tests pass, changes are committed using version control. Commits should be **small and incremental**, enabling effective review and traceability.

The process then returns to Stage 3, initiating the next iteration. This reinforces continuous development while maintaining strong security controls.

7.4. Key Properties of the Framework

The Secure Vibe Coding Framework introduces several important characteristics:

- **Security by design**: Security constraints are defined upfront and enforced throughout
- **Human-AI collaboration**: AI accelerates development, but human validation remains central
- **Granularity and traceability**: Small changes improve auditability and reduce risk
- **Continuous validation**: Testing and verification are embedded in every iteration

7.5. Summary

The proposed Secure Vibe Coding Framework demonstrates that AI-assisted development can be aligned with secure engineering practices when supported by structured workflows and disciplined processes. By combining DevSecOps principles with controlled AI interaction patterns, the framework provides a practical approach to mitigating the risks associated with vibe coding while preserving its productivity benefits.

8. Discussion

The rise of AI-assisted development and “vibe coding” represents a fundamental shift in how software is created, with significant implications for application security. While previous sections have outlined the risks and proposed mitigation strategies, this section critically examines the broader impact of these changes, including trade-offs, limitations, and organizational implications.

8.1. The Productivity-Security Trade-off

One of the most significant tensions introduced by vibe coding is the trade-off between development speed and security rigor. AI-assisted tools enable rapid code generation, reducing time-to-delivery and lowering barriers to entry for software development. However, this acceleration can come at the cost of reduced scrutiny and increased vulnerability exposure.

Traditional secure development practices rely on deliberate design, careful implementation, and systematic validation. In contrast, vibe coding encourages iterative experimentation and fast feedback cycles, which may inadvertently deprioritize security activities such as threat modelling, code review, and testing.

This tension is not inherently negative but requires careful management. As demonstrated in the proposed Secure Vibe Coding Framework, productivity gains can be preserved while maintaining security, provided that structured controls – such as automated testing and human-in-the-loop validation – are consistently applied.

8.2. Changing Role of the Developer

AI-assisted development transforms the role of the developer from a primary code author to a **code curator and validator**. Instead of writing code line by line, developers increasingly guide, refine, and verify AI-generated outputs.

This shift introduces both opportunities and risks:

- **Opportunities**:
  - Faster prototyping and experimentation
  - Increased accessibility for less experienced developers
  - Reduced cognitive load for routine tasks

- **Risks:**
  - Superficial understanding of system behaviour
  - Overreliance on AI-generated solutions
  - Reduced ability to identify subtle security flaws

The effectiveness of vibe coding therefore depends heavily on the developer's ability to critically evaluate generated code. This reinforces the importance of maintaining strong software engineering fundamentals, as discussed in Section 6.

8.3. Impact on Different Developer Profiles

The impact of vibe coding is not uniform across all developers. Experience level plays a significant role in how effectively AI tools are used:

- **Junior developers** may benefit from increased productivity but are more susceptible to automation bias and may lack the expertise to identify insecure patterns.
- **Experienced developers** are better positioned to leverage AI as a productivity tool while maintaining critical oversight but may still face challenges in verifying large volumes of generated code.

This disparity suggests that organizations must tailor their adoption strategies, accordingly, including targeted training and clear guidelines for AI-assisted development.

8.4. Organizational and Process Implications

At an organizational level, the adoption of vibe coding requires adjustments to development processes, governance, and risk management practices. Existing frameworks such as DevSecOps remain relevant but must be adapted to account for AI-driven workflows.

Key implications include:

- **Policy and governance**: Organizations must define clear policies regarding the use of AI tools, including acceptable data sharing practices and security requirements.
- **Toolchain integration**: AI tools must be securely integrated into development environments, with appropriate access controls and monitoring.
- **Process adaptation**: Development workflows must explicitly incorporate validation steps for AI-generated code, including code reviews and automated testing.

Failure to address these aspects may result in inconsistent practices and increased exposure to security risks.

8.5. Limitations of Current Mitigation Approaches

While the mitigation strategies proposed in Section 6 provide a robust foundation, they are not without limitations. Automated security tools, such as SAST and DAST, may not fully capture the contextual nuances of AI-generated code. Similarly, human reviewers may struggle to effectively analyse large volumes of generated output, particularly under time constraints.

Moreover, current AI systems lack intrinsic security awareness and do not guarantee adherence to secure coding standards. As a result, mitigation efforts often rely on external controls rather than improvements in the code generation process itself.

These limitations highlight the need for continued research into **security-aware AI systems** and improved integration between AI tools and security frameworks.

8.6. Evolving Threat Landscape

The introduction of AI into the development process not only changes how vulnerabilities are introduced but also how they may be exploited. Attackers can leverage similar AI tools to discover vulnerabilities, generate exploits, or craft more sophisticated attacks.

Additionally, new attack vectors – such as prompt injection and model manipulation – emerge as direct consequences of integrating AI into the software development lifecycle. This creates a dynamic environment in which both defenders and attackers benefit from increased automation.

As a result, application security must evolve to address not only traditional vulnerabilities but also **AI-specific threats**, as discussed in Section 4.

8.7. Toward a Balanced Approach

The findings of this paper suggest that the goal is not to restrict or avoid vibe coding, but to **adopt it responsibly**. A balanced approach requires:

- Maintaining strong engineering discipline
- Embedding security controls into development workflows
- Ensuring continuous validation of AI-generated outputs
- Promoting developer awareness and training

The Secure Vibe Coding Framework provides a practical example of how such a balance can be achieved, combining the benefits of AI-assisted development with the rigor of established security practices.

8.8. Summary

In summary, vibe coding introduces both significant opportunities and substantial risks for application security. Its impact extends beyond individual code snippets to affect developer behaviour, organizational processes, and the broader threat landscape.

Effectively addressing these challenges requires a shift in mindset: from viewing AI as a replacement for traditional practices to understanding it as a powerful tool that must be integrated within a disciplined and security-aware development framework.

9. Future Directions

The rapid evolution of AI-assisted development and the emergence of “vibe coding” present both immediate challenges and long-term opportunities for application security. While this paper has outlined current risks and mitigation strategies, the field is still in its early stages. This section explores key areas where further research, innovation, and standardization are required to ensure that security practices evolve in parallel with AI-driven development.

9.1. Security-Aware AI Code Generation

A fundamental limitation of current AI coding tools is their lack of intrinsic security awareness. While they can generate syntactically correct and functionally relevant code, they do not consistently adhere to secure coding standards or contextual security requirements.

Future research should focus on developing **security-aware AI models** that:

- Incorporate secure coding guidelines (e.g., OWASP, CERT) into training and inference
- Detect and avoid insecure patterns during code generation
- Provide explanations or warnings when generating potentially risky code

Such capabilities would shift part of the security burden from post-generation validation to **proactive risk reduction at the source**, improving the overall security posture of AI-assisted development.

9.2. Integration of AI into Security Tooling

While AI is currently used to generate code, its potential in **security analysis and enforcement** remains underutilized. Future development should explore deeper integration of AI into security tooling, including:

- AI-enhanced static and dynamic analysis tools
- Automated vulnerability detection tailored to AI-generated code patterns
- Intelligent code review assistants capable of identifying contextual security flaws

By leveraging AI on both sides of the development process – code generation and security validation – organizations can create more balanced and adaptive security ecosystems.

9.3. Standardization and Best Practices for AI-Assisted Development

The lack of standardized guidelines for secure AI-assisted development presents a significant gap. While frameworks such as DevSecOps and NIST SSDF provide a strong foundation, they do not fully address the unique challenges introduced by vibe coding.

Future efforts should aim to:

- Define **industry standards** for secure use of AI coding tools
- Establish best practices for prompt design, validation, and data handling
- Develop certification or compliance frameworks for AI-assisted development environments

Standardization would provide organizations with clearer guidance and reduce inconsistencies in how AI tools are adopted and secured.

9.4. Regulatory and Compliance Considerations

As AI becomes more deeply integrated into software development, regulatory bodies are likely to introduce requirements governing its use. These may include:

- Restrictions on sharing sensitive data with external AI services
- Requirements for traceability and accountability in AI-generated code
- Compliance with data protection regulations (e.g., GDPR) in AI-assisted workflows

Organizations must prepare for an evolving regulatory landscape by implementing **transparent and auditable processes**, ensuring that AI usage can be monitored and justified.

9.5. Education and Developer Training

The shift toward AI-assisted development necessitates changes in how developers are trained. Traditional programming education focuses on code creation, whereas vibe coding emphasizes **code evaluation and validation**.

Future educational initiatives should:

- Teach developers how to critically assess AI-generated code
- Emphasize secure coding principles in AI-assisted contexts
- Promote awareness of AI-specific risks, such as prompt injection and data leakage

Developing these skills is essential to ensuring that developers can effectively manage the risks associated with AI-driven workflows.

9.6. Evolution of Threat Modelling and Security Frameworks

Existing threat modelling methodologies and security frameworks must continue to evolve to address the complexities introduced by AI. This includes:

- Extending threat models to incorporate AI components and prompt flows
- Developing new taxonomies for AI-specific vulnerabilities
- Integrating AI-related risks into established frameworks such as OWASP and STRIDE

As demonstrated in this paper, AI introduces new attack surfaces and threat vectors that cannot be fully captured by traditional models alone.

9.7. Toward Autonomous and Self-Healing Systems

Looking further ahead, AI may play a role in enabling **autonomous or self-healing systems**, capable of detecting and mitigating vulnerabilities in real time. Such systems could:

- Automatically identify insecure code patterns
- Suggest or implement fixes
- Continuously adapt to emerging threats

While this vision presents exciting possibilities, it also raises important questions regarding trust, control, and accountability. Ensuring that such systems operate securely and transparently will be a critical area of future research.

9.8. Summary

The future of application security in the age of vibe coding will be shaped by the interplay between AI capabilities, developer practices, and organizational controls. Advancements in security-aware AI, improved tooling, and standardized practices will be essential to managing the risks associated with AI-assisted development.

Ultimately, the goal is not only to mitigate current vulnerabilities but to build a **resilient and adaptive security ecosystem** that evolves alongside the technologies it seeks to protect.

10. Conclusion

The emergence of AI-assisted development and the rise of "vibe coding" represent a significant transformation in software engineering practices. By enabling rapid code generation and lowering barriers to entry, these technologies offer substantial productivity benefits. However, as this paper has demonstrated, they also introduce a range of security risks that challenge traditional assumptions about code quality, developer understanding, and secure development processes.

The analysis presented highlights that vulnerabilities in AI-assisted development are not limited to isolated code snippets but extend across multiple dimensions, including development workflows, toolchains, and the broader software ecosystem. Insecure code generation, overreliance on AI outputs, supply chain risks, prompt injection, and data leakage collectively contribute to an expanded and more complex attack surface.

To address these challenges, this paper proposed a set of mitigation strategies grounded in established security practices, including DevSecOps, secure coding principles, and human-in-the-loop validation. Central to this approach is the recognition that AI-generated code must be treated as untrusted input, requiring systematic validation and continuous oversight. The importance of incremental development, robust version control practices, and automated security testing was emphasized as critical enablers of secure AI-assisted workflows.

Building on these principles, the Secure Vibe Coding Framework was introduced as a practical model for integrating security into AI-driven development. By combining structured design constraints, DevSecOps pipelines, test-driven validation, and controlled AI interaction patterns, the framework demonstrates how organizations can balance the speed of vibe coding with the rigor of secure software engineering.

Ultimately, the key insight of this work is that AI does not eliminate the need for disciplined engineering practices – rather, it amplifies their importance. Developers must transition from being solely code authors to becoming critical evaluators of AI-generated outputs, while organizations must adapt their processes to ensure that security remains an integral part of the development lifecycle.

As AI continues to evolve, the challenge for the software engineering community will be to harness its capabilities without compromising security. Achieving this balance requires not only improved tools and frameworks but also a shift in mindset – one that recognizes AI as a powerful collaborator that must be guided, constrained, and continuously validated.

References

- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). *Evaluating large language models trained on code*. arXiv. https://arxiv.org/abs/2107.03374
- Cloudaware. (n.d.). *DevSecOps framework: Integrating security into DevOps*. Retrieved April 10, 2026, from https://cloudaware.com/blog/devsecops-framework/
- GitHub. (2023). *Research: Quantifying GitHub Copilot's impact on developer productivity and happiness*. https://github.blog/2023-03-22-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/
- Khan, A. A., et al. (2023). *Security vulnerabilities in AI-generated code: An empirical study*. arXiv. https://arxiv.org/abs/2304.09655
- Kim, M. (2023, April 3). *Samsung bans ChatGPT after sensitive code leak*. Bloomberg.
- Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., & Kirda, E. (2018). *Thou shalt not depend on me: Analyzing the use of outdated JavaScript libraries on the web*. NDSS Symposium.
- Microsoft. (n.d.). *What is DevSecOps?* Retrieved April 10, 2026, from https://www.microsoft.com/en-us/security/business/security-101/what-is-devsecops
- National Institute of Standards and Technology (NIST). (2022). *Secure Software Development Framework (SSDF) version 1.1 (SP 800-218)*. https://csrc.nist.gov/Projects/ssdf
- Open Web Application Security Project (OWASP). (2021). *OWASP Top 10: The ten most critical web application security risks*. https://owasp.org/www-project-top-ten/
- Open Web Application Security Project (OWASP). (2023). *OWASP Top 10 for large language model applications*. https://owasp.org/www-project-top-10-for-large-language-model-applications/
- Open Web Application Security Project (OWASP). (n.d.). *OWASP DevSecOps guideline*. Retrieved April 10, 2026, from https://owasp.org/www-project-devsecops-guideline/
- Rahman, M. M., et al. (2021). *An empirical study on code review practices and software quality*. arXiv. https://arxiv.org/abs/2102.06909

- Schneier, B. (2023). *AI and security: Risks and opportunities*. Harvard Kennedy School Belfer Center. https://www.belfercenter.org/publication/ai-and-security
- Sharma, T., Kazman, R., & Chen, H. (2021). *A systematic literature review on DevSecOps*. arXiv. https://arxiv.org/abs/2103.08266
- Shostack, A. (2014). *Threat modeling: Designing for security*. Wiley.
- Ziegler, D. M., et al. (2022). *Measuring coding challenge competence with AI assistants*. arXiv. https://arxiv.org/abs/2211.03622