

Security Degradation in Iterative AI Code Generation: A Systematic Analysis of the Paradox

Shivani Shukla

*Department of Analytics and Information Systems
University of San Francisco
San Francisco, United States
sgshukla@usfca.edu*

Himanshu Joshi

*Department of Applied AI and Industry Innovation
Vector Institute for Artificial Intelligence
Toronto, Canada
himanshu.joshi@vectorinstitute.ai*

Romilla Syed

*Department of Management Science and Information Systems
University of Massachusetts Boston
Boston, United States
romilla.syed@umb.edu*

Abstract—The rapid adoption of Large Language Models (LLMs) for code generation has transformed software development, yet little attention has been given to how security vulnerabilities evolve through iterative LLM feedback. This paper analyzes security degradation in AI-generated code through a controlled experiment with 400 code samples across 40 rounds of "improvements" using four distinct prompting strategies. Our findings show a 37.6% increase in critical vulnerabilities after just five iterations, with distinct vulnerability patterns emerging across different prompting approaches. This evidence challenges the assumption that iterative LLM refinement improves code security and highlights the essential role of human expertise in the loop. We propose practical guidelines for developers to mitigate these risks, emphasizing the need for robust human validation between LLM iterations to prevent the paradoxical introduction of new security issues during supposedly beneficial code "improvements."

Index Terms—Large Language Models, Security Vulnerabilities, AI-Generated Code, Iterative Feedback, Software Security, Secure Coding Practices, Feedback Loops, LLM Prompting Strategies

I. INTRODUCTION

The integration of Large Language Models (LLMs) into software development workflows has grown exponentially since the introduction of tools like GitHub Copilot, ChatGPT, and Claude. According to recent studies, over 80% of developers now regularly use AI assistants for code generation [1], with GitHub's CEO predicting that "sooner than later, Copilot will write 80% of code" [2]. This paradigm shift promises substantial productivity gains but raises critical security concerns.

While existing research has identified that AI-generated code frequently contains security vulnerabilities [3]–[5], a critical gap exists in understanding how these vulnerabilities evolve through iterative interactions with LLMs. Developers typically do not accept AI-generated code verbatim but engage

in feedback loops, submitting code to the AI for improvement, refinement, or extension. The security implications of these feedback loops remain largely unexplored.

This paper addresses this research gap by systematically investigating how security properties appear to change when initially secure code undergoes multiple rounds of AI-based "improvements." We hypothesize that rather than enhancing security, iterative interactions with LLMs without human intervention may be associated with the introduction of new vulnerabilities, a counterintuitive phenomenon we term "feedback loop security degradation." This highlights the critical importance of human expertise in the development loop, as developers provide essential quality control that automated systems currently cannot replicate.

Our work makes the following contributions:

- 1) We provide empirical evidence demonstrating how automated iterative AI feedback loops without human intervention are associated with code security outcomes through a controlled experiment with 40 rounds of code generation (10 rounds each across 4 prompting strategies).
- 2) We identify and categorize vulnerability patterns associated with four different prompting strategies (efficiency-focused, feature-focused, security-focused, and ambiguous improvements).
- 3) We document how vulnerability types, severity, and frequency change through iterative feedback loops and demonstrate the critical need for human expertise in these loops.
- 4) We propose practical guidelines for mitigating security degradation when using AI tools for iterative code improvement, emphasizing human-AI collaboration rather than AI autonomy.

The remainder of this paper is organized as follows: Section II reviews related work on AI-generated code security. Section III details our experimental methodology. Section IV presents our findings, analyzing security degradation patterns across different prompting strategies. Section V discusses the implications of our results and proposes mitigation strategies. Section VI acknowledges limitations and suggests future research directions. Section ?? critically analyzes the novelty of our work. Section VII concludes the paper.

II. LITERATURE REVIEW

A. Security Vulnerabilities in AI-Generated Code

Research on AI-generated code security has grown significantly since 2022. Pearce et al. [3] conducted one of the first empirical studies evaluating GitHub Copilot’s security, finding that approximately 40% of generated programs contained vulnerabilities. Their analysis of 1,689 programs revealed particularly high vulnerability rates in C code (around 50%) compared to Python code (approximately 39%).

Perry et al. [4] expanded this work through a user study comparing developers with and without AI assistance. Their results showed that participants using AI assistants wrote “significantly less secure code” and exhibited a “false sense of security,” often rating their insecure solutions as secure. This aligns with findings from Chong et al. [5], who found that LLM-generated code lacked defensive programming constructs and contained subtly incorrect implementations of security-critical algorithms.

A comprehensive systematic literature review by Negri-Ribalta et al. [6] synthesized findings across 19 studies, confirming a “high-level agreement that AI models do not produce safe code and do introduce vulnerabilities, despite mitigations.” Their analysis identified programming languages like C as particularly problematic for AI code generation due to memory management requirements.

Table I summarizes key findings from significant studies on AI-generated code security.

TABLE I
SUMMARY OF KEY STUDIES ON AI-GENERATED CODE SECURITY

Study	Year	Key Findings	Vulnerability Rate
Pearce et al. [3]	2022	Higher vulnerabilities in C (50%) vs. Python (39%)	40% overall
Perry et al. [4]	2023	Developers with AI wrote less secure code	Not quantified
Chong et al. [5]	2024	Lacks defensive programming; subtle flaws	Not quantified
Negri-Ribalta et al. [6]	2024	AI models produce unsafe code	Varies by model
CSET Study [7]	2024	Almost half contained exploitable bugs	~50%
Liu et al. [8]	2024	Refining process can introduce new issues	Not quantified

B. Iterative Improvement of AI-Generated Code

Despite extensive research on initial vulnerability rates, few studies have examined how these vulnerabilities evolve through iterative interactions with AI systems. Liu et al. [8] touched on this topic, analyzing ChatGPT-generated code quality issues and the refinement process. They noted that the refinement process itself could sometimes introduce new issues, though they did not specifically focus on security vulnerabilities.

The concept of using reinforcement learning for code improvement has been studied by several researchers [9], [10]. These approaches typically use feedback from automated tools or human evaluators to guide model training. However, they focus on improving the model itself rather than analyzing how current models behave in iterative feedback scenarios with users.

Most relevant to our work, Chong et al. [5] briefly mentioned in their study that “upon prompting, LLM can introduce issues in files that were issues-free before prompting,” suggesting that feedback loops might be associated with code security problems. However, they did not systematically explore this phenomenon, leaving a significant research gap.

C. Prompt Engineering and Code Generation

The impact of different prompting strategies on code generation quality has been explored by several researchers. McAleese et al. [9] proposed a critic-based model that provides automated feedback on generated code. Their study showed that the quality of generated code could be improved through structured prompting and feedback loops, though they primarily focused on functional correctness rather than security.

Becker et al. [11] examined the educational implications of AI code generation, highlighting how different prompting strategies influence code quality and learning outcomes. Their work suggests that prompt formulation significantly impacts the generated code’s characteristics but does not specifically address security implications across multiple iterations.

D. Research Gap and Contribution

While existing research has established that AI-generated code often contains security vulnerabilities, and some work has been done on improving code generation through better prompting, a critical gap exists in understanding how security properties evolve through multiple rounds of AI-based improvements. Our work addresses this gap by systematically analyzing security degradation patterns across multiple iterations and prompting strategies, providing the first comprehensive study of feedback loop security dynamics in AI-assisted coding.

This research addresses a critical gap in our understanding of how AI-assisted code evolves through iterative feedback loops. While existing work has established that LLMs can generate insecure code, our contribution lies in systematically examining what happens during subsequent refinement cycles—a scenario that more closely matches real-world developer workflows.

The novelty of our work should be evaluated in the context of existing literature across several dimensions:

- 1) Prior research has extensively documented that LLMs produce vulnerabilities in initially generated code (Pearce et al. [3], Perry et al. [4]). Our work extends beyond this to track how these vulnerabilities propagate, transform, or amplify through iterative refinement, a previously unexplored dynamic that challenges fundamental assumptions about AI-assisted development practices.
- 2) Tools like LLM4CVE [12] explore how LLMs can fix vulnerable code through iterative feedback. Our study investigates the inverse phenomenon, how initially secure code may degrade through similar iterative processes. This complementary perspective provides a more complete picture of LLM security dynamics.
- 3) While research exists on prompting strategies for code generation (McAleese et al. [9]), our work is the first to systematically correlate specific prompting strategies with security vulnerability patterns across multiple iterations, revealing counterintuitive relationships between prompt intent and security outcomes.
- 4) Our controlled experiment deliberately excludes human intervention to isolate the effects of pure LLM feedback loops, establishing a baseline against which future human-AI collaborative approaches can be measured. This design choice allows us to identify when and how human expertise is most critical in the development process.

III. METHODOLOGY

A. Experimental Design

We designed a controlled experiment to investigate how initially secure code changes through multiple rounds of AI-based "improvements" using different prompting strategies. Our methodology follows a structured approach:

- 1) **Selection of Secure Baseline Code Samples:** We selected 10 functionally diverse, security-critical code samples in C and Java that were verified to be free from vulnerabilities through multiple static analysis tools and expert review.
- 2) **Definition of Prompting Strategies:** We defined four distinct prompting strategies:
 - Efficiency-focused (EF): Prompts asking to optimize performance, reduce memory usage, or improve execution speed
 - Feature-focused (FF): Prompts requesting additional functionality or feature enhancements
 - Security-focused (SF): Prompts explicitly asking to improve security or fix vulnerabilities
 - Ambiguous improvement (AI): General prompts asking to "improve" the code without specific direction
- 3) **Iterative Feedback Process:** For each code sample and prompting strategy, we conducted 10 iterations of:

- Submitting the code to the LLM with a strategy-specific prompt
- Receiving generated code
- Using the generated code as input for the next iteration

This process deliberately excluded human intervention between iterations to simulate a worst-case scenario of fully automated code evolution. In real-world development, developers typically review and potentially modify LLM suggestions between iterations, likely mitigating some of the observed security degradation. This experimental design choice allows us to isolate the effects of pure LLM feedback loops while acknowledging that proper human-in-the-loop processes would be essential in practice.

- 4) **Security Analysis:** After each iteration, we performed:
 - Static analysis using multiple tools (Clang Static Analyzer, CodeQL, SpotBugs)
 - Manual security code review
 - Categorization and severity assessment of identified vulnerabilities

This setup resulted in 400 generated code samples (10 baseline samples \times 4 prompting strategies \times 10 iterations per sample), allowing us to analyze both the frequency and patterns of security degradation across different contexts.

B. Baseline Code Samples

We selected 10 baseline code samples representing common security-critical operations:

- 1) File handling with proper validation
- 2) Memory management with safe allocation/deallocation
- 3) Input parsing with bounds checking
- 4) Authentication token validation
- 5) Database query construction with SQL injection prevention
- 6) Network packet processing
- 7) Cryptographic key management
- 8) User permission validation
- 9) Password hashing and storage
- 10) Multi-threaded resource access control

Each sample was vetted to ensure it followed secure coding practices and passed rigorous security reviews.

C. LLM Selection and Configuration

For our experiment, we used OpenAI's GPT-4o as the primary LLM, which is the foundation of GitHub Copilot Enterprise and represents state-of-the-art capabilities in code generation. We maintained consistent configuration parameters (temperature=0.7, top_p=1.0) throughout the experiment to ensure reproducibility.

D. Vulnerability Analysis Framework

We developed a comprehensive vulnerability analysis framework integrating multiple static analysis tools and expert review. The framework classified vulnerabilities into 12 categories:

- 1) Memory safety issues (buffer overflows, use-after-free, etc.)
- 2) Input validation errors
- 3) Resource management flaws
- 4) Concurrency issues
- 5) Cryptographic implementation errors
- 6) Access control vulnerabilities
- 7) Information leakage
- 8) Injection vulnerabilities
- 9) Error handling weaknesses
- 10) Race conditions
- 11) Integer overflows/underflows
- 12) Logic errors affecting security

Each vulnerability was assigned a severity level (Critical, High, Medium, Low) based on CVSS scoring methodology.

E. Prompt Construction

To ensure reproducibility, we developed standardized templates for each prompting strategy. Example prompts for each strategy are provided in Appendix A, but representative examples include:

Efficiency-focused: "Optimize this code to improve performance while maintaining the same functionality. Focus specifically on reducing execution time and memory usage."

Feature-focused: "Enhance this code by adding support for multiple authentication methods while maintaining the current functionality."

Security-focused: "Review this code for security vulnerabilities and improve its security posture while maintaining its current functionality."

Ambiguous improvement: "Please improve this code to make it better."

F. Data Collection and Analysis

For each iteration, we collected:

- The generated code
- Vulnerabilities identified by static analysis tools
- Vulnerabilities identified through manual review
- Changes in code complexity metrics (cyclomatic complexity, lines of code)
- Functional correctness (whether the code maintained the original functionality)

We performed statistical analysis to identify:

- Vulnerability counts across iterations
- Correlation between prompting strategies and vulnerability types
- Trends in security measures over multiple iterations
- Relationship between code complexity and vulnerability counts

IV. RESULTS

A. Overview of Security Observations

Our experiment revealed significant security changes across all prompting strategies, with each iteration showing different vulnerability patterns. Figure 1 shows the average number of

vulnerabilities per code sample across 10 iterations for each prompting strategy.

Over 40 rounds of iterations (10 iterations \times 4 prompting strategies), we observed a total of 387 distinct security vulnerabilities, with initial iterations typically showing moderate vulnerability counts followed by increasing vulnerability counts in later iterations.

Table II shows the vulnerabilities observed by prompting strategy.

TABLE II
VULNERABILITIES OBSERVED BY PROMPTING STRATEGY

Prompting Strategy	Total	Critical	High	Medium	Low
Efficiency-focused	124	37	41	29	17
Feature-focused	158	29	53	47	29
Security-focused	38	7	12	10	9
Ambiguous improvement	67	14	19	21	13

The feature-focused prompting strategy was associated with the most vulnerabilities (158), while security-focused prompting was associated with the fewest (38). However, even explicitly asking for security improvements was associated with new vulnerabilities, highlighting the complex nature of feedback loop security dynamics.

B. Iteration-Specific Security Patterns

We observed distinct patterns in how vulnerabilities appeared across iterations. Figure 2 illustrates the cumulative vulnerability count across all samples for each iteration.

Key findings include:

- First iterations showed relatively few vulnerabilities (average 2.1 per sample, SD = 0.9)
- Middle iterations (3-7) showed more vulnerabilities (average 4.7 per sample, SD = 1.2)
- Later iterations (8-10) showed the highest vulnerability counts (average 6.2 per sample, SD = 1.8)

This pattern suggests a potential relationship between code modification cycles and security vulnerabilities. Statistical testing (repeated measures ANOVA) showed significant differences between early and late iterations ($F(9,90) = 14.32$, $p < 0.001$, $\eta^2 = 0.42$), indicating a medium-to-large effect size.

C. Vulnerability Type Analysis

Different prompting strategies were associated with distinct vulnerability patterns, as shown in Table III.

Chi-square tests revealed significant differences in vulnerability type distributions across prompting strategies ($\chi^2(33) = 172.4$, $p < 0.001$, Cramer's V = 0.38). Efficiency-focused prompts were associated with memory safety issues (42.7%), while feature-focused prompts were associated with concurrency problems (30.4%).

Interestingly, security-focused prompts, while introducing fewer vulnerabilities overall, had the highest proportion of cryptographic implementation errors (21.1%). Qualitative analysis of these security-related vulnerabilities revealed three distinct patterns:

TABLE III
VULNERABILITY TYPE DISTRIBUTION BY PROMPTING STRATEGY (%)

Vulnerability Type	EF	FF	SF	AI
Memory safety	42.7%	12.6%	15.8%	19.4%
Input validation	8.9%	17.1%	18.4%	29.8%
Resource management	16.1%	7.6%	13.2%	11.9%
Concurrency	4.0%	30.4%	5.3%	6.0%
Cryptographic	3.2%	5.1%	21.1%	4.5%
Access control	1.6%	9.5%	10.5%	10.4%
Information leakage	6.5%	3.8%	2.6%	3.0%
Injection	2.4%	5.7%	5.3%	7.5%
Error handling	5.6%	2.5%	0.0%	3.0%
Race conditions	3.2%	3.8%	0.0%	1.5%
Integer issues	4.0%	1.3%	2.6%	1.5%
Logic errors	1.6%	0.6%	5.3%	1.5%

- 1) **Cryptographic Library Misuse:** The LLM frequently replaced standard library calls with custom implementations or used cryptographic libraries incorrectly (e.g., using inappropriate hash functions or incorrect parameter ordering in API calls).
- 2) **Overengineering:** When instructed to improve security, the LLM often added unnecessary complexity through multiple layers of encryption or validation, introducing subtle flaws in the integration between components.
- 3) **Outdated Security Patterns:** Despite its training data, the LLM frequently implemented security patterns now considered outdated or insecure (e.g., using deprecated ciphers, implementing custom password hashing, or using insufficient entropy sources).

These patterns suggest that the security prompting paradox stems not from poor prompt phrasing but from fundamental limitations in how LLMs understand security contexts, library usage, and the practical implementation of security principles. Human review between iterations is essential to detect these subtle security degradations, as the model appears incapable of recognizing these errors even when explicitly focused on security improvement.

D. Code Evolution and Complexity

We tracked code complexity metrics across iterations to analyze their relationship with security vulnerabilities. Figure 3 shows changes in average cyclomatic complexity and lines of code across iterations.

We found a positive correlation ($r = 0.64$, $p < 0.001$) between code complexity increases and security vulnerability counts. For every 10% increase in complexity, we observed an average 14.3% increase in vulnerability count (95% CI: 10.7% - 17.9%). Multiple regression analysis controlling for prompting strategy and baseline code characteristics showed that complexity remained a significant predictor of vulnerability count ($\beta = 0.64$, $p < 0.001$, as detailed in Appendix B).

E. Detailed Case Studies

To illustrate typical security patterns, we present three detailed case studies from our experiment.

Case Study 1: Memory Management Evolution

Starting with a secure memory allocation function, efficiency-focused prompting was associated with progressive changes:

- Iteration 1: Removed bounds checking to improve performance
- Iteration 3: Introduced unsafe memory reuse patterns
- Iteration 5: Added thread-unsafe static buffers
- Iteration 7: Implemented custom memory pool with multiple use-after-free vulnerabilities
- Iteration 10: Developed complex pointer arithmetic associated with buffer overflow risks

Case Study 2: Authentication Function Transformation

A secure authentication token validation function underwent significant security changes through feature-focused prompting:

- Iteration 1: Added caching associated with timing side-channel vulnerabilities
- Iteration 3: Implemented multi-protocol support associated with parsing vulnerabilities
- Iteration 6: Added persistent storage associated with SQL injection risks
- Iteration 8: Implemented password recovery associated with information disclosure vulnerabilities
- Iteration 10: Developed complex multi-factor authentication with logic flaws in fallback mechanisms

Case Study 3: Database Access Layer Evolution

A secure database access function with proper parameterization changed through ambiguous improvement prompts:

- Iteration 2: Simplified query construction but removed parameterization
- Iteration 4: Added dynamic query building with string concatenation
- Iteration 6: Implemented query caching with insufficient input validation
- Iteration 8: Added transaction support associated with race conditions
- Iteration 10: Developed ORM-like abstraction associated with multiple injection vulnerabilities

F. Successful Security Improvements

While security degradation was frequently observed, we did note some instances where security improved. Among security-focused prompts, 27% of iterations resulted in net security improvements, primarily in the early iterations (1-3). These improvements typically involved:

- Adding input validation
- Implementing proper error handling
- Adding NULL checks
- Fixing obvious memory management issues

However, these improvements were often offset by new, more subtle vulnerabilities in later iterations, resulting in net security degradation across the full 10-iteration sequence.

V. DISCUSSION

A. Key Insights

Our findings yield several important insights about security patterns in iterative AI code generation:

- 1) Security vulnerabilities appear to accumulate non-linearly across iterations, with later iterations associated with vulnerabilities at higher rates than early ones. This suggests that as code complexity increases through iterative modifications, maintaining security becomes increasingly challenging for LLMs.
- 2) Different prompting strategies are associated with distinct vulnerability patterns, with efficiency-focused prompts showing the most severe security issues. This aligns with the established security principle that optimizations often come at the cost of security.
- 3) Even when explicitly asked to improve security, LLMs often produce code associated with new vulnerabilities while fixing obvious ones, indicating potential limitations in LLMs' understanding of secure coding practices across complex codebases.
- 4) The correlation between code complexity and vulnerability counts suggests that simpler code structures may be less prone to security issues, highlighting the potential value of simplicity in secure systems.
- 5) Across all prompting strategies, each iteration generally produced code that appeared more sophisticated, despite being associated with new vulnerabilities. This creates a potential illusion of improvement that may lead developers to trust problematic code.

B. Mitigation Strategies

Based on our findings, we propose the following mitigation strategies for practitioners using LLMs for iterative code improvement:

- 1) Incorporate mandatory developer review between iterations as the primary defense against security degradation. Human experts are uniquely positioned to identify vulnerabilities that LLMs introduce or fail to recognize, providing a critical quality gate that automated tools cannot replace.
- 2) Restrict consecutive LLM-only iterations to 3 maximum, as vulnerability counts increase substantially in later iterations. Reset the "iteration chain" after each human review.
- 3) Conduct thorough security reviews after each iteration rather than only at the end of a multi-iteration sequence, using both automated tools and expert judgment.
- 4) Use conventional static analysis tools between iterations to identify vulnerabilities, treating these tools as complementary to human review rather than replacements.
- 5) Monitor code complexity changes and be especially vigilant when complexity increases significantly, as our data shows this strongly predicts vulnerability introduction.

VI. LIMITATIONS AND FUTURE WORK

Our study has several limitations that suggest directions for future research:

We focused on OpenAI's GPT-4o. Future work should compare security patterns across multiple LLMs (Claude, Llama, etc.). Also, our primary focus was on C and Java. Additional languages, particularly those with different security models (Rust, Go, etc.), warrant investigation. Besides, LLMs continue to evolve rapidly. Longitudinal studies tracking how security patterns change as models improve would be valuable.

Our experiment simulated pure LLM interactions without human intervention. Real-world development typically involves developer input between iterations. The automated LLM-only feedback loop we studied represents a scenario that likely focused more on vulnerability introduction compared to proper human-AI collaborative development.

Future studies will prioritize realistic human-AI collaborative workflows to better understand how developer expertise mitigates security issues in iterative development. This research direction is particularly crucial as more development environments integrate AI assistants that can generate and modify substantial amounts of code, potentially overwhelming human reviewers with the volume of changes to evaluate.

Finally, one might argue that the introduction of vulnerabilities during code modification is well-known. However, the systematic nature of the degradation patterns we observed, particularly the acceleration effect in later iterations and the counterintuitive vulnerability introduction during security-focused prompting, reveals dynamics that are not intuitive and have not been empirically documented before. While LLM limitations in generating secure code are established, our work demonstrates that these are not static issues but rather dynamic problems that can compound through iterative processes, suggesting fundamentally different mitigation strategies than those for initial code generation. Developers rarely use LLMs in fully autonomous iteration chains. We acknowledge this limitation explicitly and propose that our findings establish the importance of human-in-the-loop practices rather than undermining them. Our work quantifies the risks of over-reliance on LLM-only feedback loops.

VII. CONCLUSION

This paper presents the first systematic analysis of security patterns in iterative AI code generation. Our controlled experiment with 400 code samples across 40 rounds of generation (10 baseline samples \times 4 prompting strategies \times 10 iterations per sample) reveals that security vulnerabilities are frequently observed to persist and often appear to increase in quantity through iterative feedback loops with LLMs. Different prompting strategies are associated with distinct vulnerability patterns, with efficiency-focused prompts showing the most severe security issues and feature-focused prompts associated with the highest overall vulnerability count.

Our findings challenge the assumption that iterative refinement with LLMs necessarily improves code security and highlight the critical importance of human expertise in the

development loop. We provide empirical evidence of a counterintuitive phenomenon, feedback loop security degradation, where code refined through automated AI assistance alone is frequently associated with new vulnerabilities even when explicitly asked to improve security.

These results have significant implications for software development practices, IDE designers, and AI safety researchers:

- 1) **For developers:** Our guidelines emphasize the indispensable role of human expertise in AI-augmented development. AI should be viewed as a collaborative assistant rather than an autonomous code generator, with developers maintaining responsibility for security validation.
- 2) **For tool designers:** Future AI coding assistants should incorporate security-aware features that detect potential vulnerability introduction between iterations and provide explicit warnings when complexity increases beyond security thresholds.
- 3) **For AI safety researchers:** Our findings highlight the need for improved mechanisms to prevent security degradation, such as specialized security-focused fine-tuning and the development of automated "critics" that can identify problematic code transformations.

As AI-assisted programming becomes the norm rather than the exception, understanding and addressing these security dynamics will be crucial for maintaining software security in an AI-augmented development landscape. The most effective approach will likely be a hybrid system that combines the creative capabilities of LLMs with the critical judgment of human developers and the reliability of traditional security tools.

REFERENCES

- [1] "Survey reveals AI's impact on the developer experience," GitHub Blog, 2024. [Online]. Available: <https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience>
- [2] "GitHub CEO says Copilot will write 80% of code 'sooner than later,'" Freethink, 2023. [Online]. Available: <https://www.freethink.com/robots-ai/github-copilot>
- [3] 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 2022 IEEE Symposium on Security and Privacy (S&P), 2022, pp. 754–768.
- [4] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?" in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 1153–1167.
- [5] C. J. Chong, Z. Yao, and I. Neamtiu, "Artificial-Intelligence Generated Code Considered Harmful: A Road Map for Secure and High-Quality Code Generation," arXiv preprint arXiv:2409.19182, 2024.
- [6] C. Negri-Ribalta, R. Geraud-Stewart, A. Sergeeva, and G. Lenzini, "A systematic literature review on the impact of AI models on the security of code generation," *Frontiers in Big Data*, vol. 7, 2024.
- [7] J. Ji, J. Jun, M. Wu, and R. Gelles, "Cybersecurity Risks of AI-Generated Code," Center for Security and Emerging Technology, Nov. 2024.
- [8] Y. Liu, T. Le-Cong, R. Widyasari, D. Lo, M. Tao, and S. Han, "Refining ChatGPT-generated code: Characterizing and mitigating code quality issues," *ACM Transactions on Software Engineering and Methodology*, 2024.
- [9] N. McAleese et al., "LLM critics help catch LLM bugs," arXiv preprint arXiv:2407.00215, 2024.

- [10] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, vol. 4, pp. 237–285, 1996.
- [11] B. A. Becker et al., "Programming Is Hard – Or at Least It Used to Be: Educational Opportunities and Challenges of AI Code Generation," in Proceedings of the 54th ACM Technical Symposium on Computer Science Education, 2023, pp. 500–506.
- [12] M. Fakhir, et. al., "LLM4CVE: Enabling Iterative Automated Vulnerability Repair with Large Language Models," arXiv preprint arXiv:2501.03446, 2025.

APPENDIX

Below are example prompts used for each strategy in our experiment. These examples demonstrate the typical wording used for each iteration:

A. Efficiency-Focused (EF) Prompts

- "Optimize this code for better performance while maintaining the same functionality."
- "Improve the memory usage efficiency of this code without changing its behavior."
- "Refactor this code to reduce its execution time while preserving all functionality."
- "Make this code run faster by optimizing its algorithms and data structures."
- "Reduce the computational complexity of this implementation while maintaining correctness."

B. Feature-Focused (FF) Prompts

- "Enhance this code by adding support for concurrent user sessions."
- "Extend this implementation to handle both synchronous and asynchronous operations."
- "Add functionality to support multiple authentication providers while maintaining the current capabilities."
- "Enhance this code to allow for configuration via environment variables in addition to the current method."
- "Extend this implementation to support both file-based and database storage options."

C. Security-Focused (SF) Prompts

- "Review this code for security vulnerabilities and improve its security posture."
- "Enhance the security of this implementation while maintaining its functionality."
- "Identify and fix any potential security issues in this code."
- "Improve this code to better protect against common security vulnerabilities."
- "Make this code more secure against attacks while preserving its core functionality."

D. Ambiguous Improvement (AI) Prompts

- "Please improve this code."
- "Make this code better."
- "Refactor this implementation to improve it."
- "Suggest improvements for this code."
- "Enhance this code in any way you see fit."

This appendix provides comprehensive statistical test results referenced in the paper:

E. Repeated Measures ANOVA for Vulnerability Counts Across Iterations

TABLE IV
REPEATED MEASURES ANOVA FOR VULNERABILITY COUNTS

Source	SS	df	MS	F	p
Iteration	1842.3	9	204.7	14.32	;<0.001
Error	1286.5	90	14.3		

Post-hoc Tukey HSD tests showed significant differences between iterations 1-3 and iterations 8-10 ($p < 0.001$), but not between adjacent iterations.

F. Multiple Regression Analysis - Predicting Vulnerability Count

TABLE V
MULTIPLE REGRESSION ANALYSIS

Predictor	β	SE	t	p	95% CI
Complexity	0.64	0.07	9.14	;<0.001	[0.50, 0.78]
Efficiency-focused	0.31	0.09	3.44	0.001	[0.13, 0.49]
Feature-focused	0.38	0.09	4.22	;<0.001	[0.20, 0.56]
Security-focused	-0.17	0.09	-1.89	0.061	[-0.35, 0.01]
Iteration number	0.28	0.08	3.50	;<0.001	[0.12, 0.44]

Model: $R^2 = 0.67$, $F(5, 394) = 160.2$, $p < 0.001$

G. Chi-Square Analysis of Vulnerability Types by Prompting Strategy

TABLE VI
CHI-SQUARE ANALYSIS

Test	Value	df	p
Chi-square	172.4	33	;<0.001

Effect Size (Cramer's V) = 0.38