Proceedings of the 7th International Conference on Computing and Data Science
DOI: 10.54254/2755-2721/2025.AST27451

An Empirical Study of Security Risks for the Web Code Generation by ChatGPT

Mingrui Hu

School of Computer Engineering, Zhanjiang University of Science and Technology, Zhanjiang, China
3011547095@qq.com

Abstract. Large language models (LLMs) have demonstrated remarkable capabilities in code generation and semantic understanding, enabling ordinary users to generate their own software systems using natural language instructions. This study takes website systems as a case to investigate a user-centered paradigm for code generation and its evaluation. First, users submit their requirements to the LLM via a web interface, prompting the model to automatically generate website project code. Then, through a set of prompt engineering methods and quantitative evaluation techniques developed for this study, we conduct a multi-dimensional assessment of the quality and security of the generated website systems using different types of LLMs and varying system function weights. A hybrid evaluation strategy is proposed to integrate and optimize assessment results across different LLMs. Evaluation dimensions include the degree to which user requirements are satisfied, completeness of website functionality, potential security risks, and code reliability. This research introduces evaluation criteria such as automated review models, functional coverage, and static vulnerability analysis to explore the feasibility, advantages, and limitations of using LLMs as both code generators and reviewers. The findings contribute to our understanding of the practical value of multi-agent LLM collaboration in software development and reveal major current challenges such as functional hallucination, incomplete implementation, and overly optimistic evaluation mechanisms.

Keywords: LLM, Code Generation, PHP, Web Vulnerability, Software Development

1. Introduction

In recent years, large language models (LLMs) have demonstrated impressive capabilities in the fields of code generation and code comprehension. Trained on vast corpora of programming languages and software documentation, these models have developed a deep understanding of programming semantics, syntax, and functionality. LLMs have already been applied in real-world software engineering scenarios, including automated bug fixing, test generation, documentation writing, and even end-to-end web application development. They support a wide range of code generation tasks—from simple algorithmic snippets to complex system prototypes—allowing users to develop programs through natural language descriptions. This not only significantly shortens the

DOI: 10.54254/2755-2721/2025.AST27451

software development cycle but also enhances code compatibility across platforms and lowers the barrier for non-professionals to build application systems.

Despite the many advantages of LLM-based code generation, the quality and security of the generated code remain highly uncertain, particularly in production-grade applications intended for real users. Issues such as incomplete functionality, logical errors, hallucinated APIs, and security vulnerabilities occur frequently. When it comes to code directly involved in front-end and back-end website interactions, such risks can lead to serious consequences, including missing authentication mechanisms, insufficient input validation, and exposed interfaces. Therefore, it is imperative to investigate whether AI-generated code truly meets user requirements, whether it implements functionality completely, and whether it ensures fundamental security guarantees.

In recent years, research on dedicated evaluation of code generated by large language models has been gradually emerging. Jiang et al. [1] provided a systematic review, highlighting key advances in pre-trained model construction, prompt engineering, and evaluation benchmarks, while also pointing out their limitations. Building on this line of work, several specific evaluation frameworks have been proposed. For example, Liu et al. [2] introduced the EvalPlus framework, which extends the HumanEval dataset by generating large-scale test cases, revealing the tendency of existing evaluation metrics to overestimate code correctness. Abbassi et al. [3] further developed a taxonomy targeting performance inefficiencies and maintenance difficulties in LLM-generated code. However, most of these studies focus on unit-level outputs or benchmark dataset performance, while paying relatively little attention to whole-site development, user-led interaction processes, or key aspects of end-to-end development such as functional implementation rate, module completeness, and security protections.

This study uses website systems as a case to investigate code generation by LLMs, aiming to explore how different types of models perform in web project development. It proposes a user-centered paradigm for code generation and evaluation. From the perspective of non-expert users, the study guides them to design and develop a functional e-commerce website through natural language interaction. First, users specify their requirements via a web interface, prompting the LLM to automatically generate the corresponding website project code. Then, based on the prompt engineering and quantitative evaluation methods developed in this study, we conduct a multi-dimensional assessment of the quality and security of the generated website systems using different types of LLMs and varying functional weights. A hybrid evaluation strategy is proposed to integrate and optimize the evaluation results across different LLMs. Key evaluation dimensions include the extent to which user requirements are satisfied, the completeness of website functionalities, the presence of security risks, and code reliability. This study introduces automated review models, functional coverage analysis, and static vulnerability analysis as evaluation standards to explore the feasibility, advantages, and limitations of using LLMs as both code generators and reviewers. The results contribute to our understanding of the practical value of collaborative multi-agent LLM systems in software development, while also revealing major current challenges such as functional hallucinations, incomplete implementations, and overly optimistic evaluation mechanisms.

2. Background

2.1. LLM and ChatGPT

Large language models (LLMs) are trained on massive datasets composed of text, code, and other data sources, and exhibit powerful capabilities in natural language understanding and generation. By learning the patterns and knowledge embedded in human language, LLMs can perform complex

DOI: 10.54254/2755-2721/2025.AST27451

tasks such as context-aware understanding and logical reasoning. ChatGPT, developed by OpenAI, is a representative example of such models and has attracted significant attention for its fluent conversational ability and broad adaptability. LLMs and ChatGPT are applied across a wide range of domains. In natural language interaction, they support intelligent question answering and multilingual translation. In content creation, they are capable of copywriting, code generation and repair, and document summarization. In productivity tasks, they assist with email processing, report generation, education and learning, and intelligent customer service. Their core competencies include understanding complex instructions, generating coherent text, performing logical inference, and adapting to diverse application scenarios, thereby significantly enhancing efficiency in information processing and content production.

2.2. Research on intelligent code generation

At present, research on code generation by LLMs remains largely limited to simple and well-defined generation tasks. For example, Gu et al. [4] employed fuzz testing to detect issues in LLM-generated test cases, proposing specific filtering strategies to clean up the generated code and improved planning strategies to enhance generation accuracy. However, such studies have not explored the use of large models for directly generating framework-level code, such as Qt, nor have they accounted for variations in performance across different granularities and perspectives of code generation. As a result, the evaluations tend to be fragmented and ambiguous.

In recent years, LLM-based code generation technology has experienced rapid development. A systematic review published by Jiang et al. [1] pointed out that from 2018 to 2024, the number of related publications has grown exponentially, with significant progress in areas such as pre-trained model construction, prompt engineering, and evaluation benchmarks—e.g., HumanEval, MBPP, and BigCodeBench. The EvalPlus framework, which automatically generates additional test cases to expand HumanEval's test coverage, revealed that code passing under original benchmark conditions saw a 19–29% drop in pass@k when evaluated more rigorously, indicating that existing evaluation tools significantly overestimate the true correctness of LLM-generated code [2]. Abbassi et al. further proposed a classification system for code efficiency and quality, analyzing 492 code samples generated by models such as CodeLlama and DeepSeek-Coder on the HumanEval++ dataset. They found widespread issues in logic, performance, readability, and maintainability [3]. In 2024, Anthropic released the Claude Sonnet 3.5 model, which achieved industry-leading performance in code generation. Trained with techniques such as Reinforcement Learning from AI Feedback (RLAIF) and Constitutional AI, the model excelled in handling long contexts and terminal tool usage scenarios [7]. Moreover, empirical studies conducted among real-world developers using tools like ChatGPT and Claude have revealed a prominent sentiment of fear—that is, concern over the potential risks of incorrect code produced by the models [2].

Several major issues and limitations remain in this field. First, the current evaluation mechanisms are insufficient. Benchmarks like HumanEval have strong limitations, as they fail to cover real-world complex scenarios and tend to significantly overestimate the quality of model-generated code [1]. Second, these models show poor adaptability to complex development environments. For instance, Yin et al. [5] attempted to enhance LLMs' ability to generate repository-aware unit tests by injecting precise contextual information. Their results suggest some improvements in test coverage, but also reveal the difficulty of scaling such methods to broader, end-to-end development tasks. They struggle with repository-level tasks, cross-file dependencies, and large-scale challenges common in actual development, such as handling GitHub issues or complex logic [1]. Third, there are persistent issues with code quality and efficiency. The generated code often suffers from

DOI: 10.54254/2755-2721/2025.AST27451

inefficiency, poor readability, redundancy, and subpar performance, all of which severely affect maintainability and runtime efficiency [3]. In addition, uneven support for domain-specific languages remains a key concern. For instance, LLMs provide limited support for low-resource languages such as Rust and DSLs, where the scarcity of datasets adversely impacts model performance [6]. Security vulnerabilities also persist. Models may produce unsafe code or fabricate dependencies, leading to phenomena such as “slopsquatting” [9]. Some studies suggest the emergence of a productivity paradox in practical development settings—experienced developers may actually see a productivity decline of about 19% when using AI tools due to the time spent reviewing and correcting generated code [8]. Finally, a critical issue that undermines code reliability and trust in AI-generated outputs is the hallucination risk. LLMs may fabricate entirely invalid code structures, methods, or package names, making their outputs difficult to trust in real-world applications [9,10].

3. Approach

3.1. Workflow

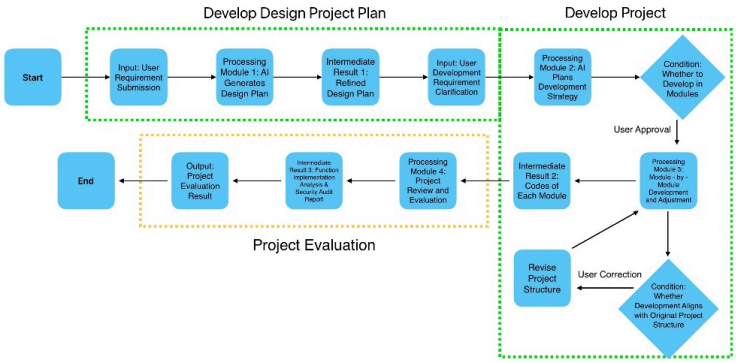

Figure 1. Generation process of an LLM-based "targeted group shopping website"

The user submits development requirements to the LLM, specifying the goal as a “targeted group shopping website,” and requests a detailed design proposal including technology stack selection, database planning, functional modules, and UI design. As shown in Figure 1, Processing Module 1 involves the AI generating the design proposal. Based on the user’s requirements, the LLM produces a comprehensive shopping website design plan, covering project positioning, technology stack, core functional modules, database schema, and UI design style. Intermediate Result 1 is the formation of the Shopping Website Design Proposal, which serves as the basis for subsequent development. Next, the user, referencing the design proposal, provides further development instructions to the LLM, including: “provide code by module,” “explain functionality in detail,” “ensure completeness and usability,” “consider basic security,” and “clearly define the project structure.” Processing Module 2 involves the AI assessing development complexity and proposing a “step-by-step modular development” strategy, where functions and pages are implemented sequentially, subject to user approval. Upon user agreement, the process proceeds to the next phase. Processing Module 3 sees the LLM developing and outputting code by module along with explanations of functionalities. During this process, the user reviews the outputs and identifies a deviation from the original project structure, providing feedback and requesting correction. Conditional Check: Does the generated content align with the original project structure? If not, the code is adjusted to conform to the

DOI: 10.54254/2755-2721/2025.AST27451

structure before development continues. If it does align, the LLM continues outputting code for each module. Intermediate Result 2 is a collection of code files for each functional module, providing foundational implementations of core features. Processing Module 4 initiates project review and evaluation. Doubao AI assesses the implementation from the perspectives of functionality and security; simultaneously, DeepSeek AI reviews the same project, calculating achievement rates by module (e.g., 60% for the user module, 45% for the product module), and generates a security audit report. Intermediate Result 3 consists of a Function Implementation Analysis Table and a List of Security Issues generated by Doubao AI and DeepSeek AI. The final output includes: Function implementation rate (e.g., overall completion rate: 37.5% / 41%). Security analysis results. Unimplemented functionalities. End.

3.2. Prompt engineering techniques for website development-oriented evaluation

3.2.1. Explicit requirement prompts

Natural language is used to clearly articulate the project objectives and task requirements. The instructions are unambiguous and semantically complete, with explicitly defined deliverables and specified output structures. Such clarity in user expression helps the LLM gain a comprehensive understanding of the project framework, user needs, and goals from the initial stage.

3.2.2. Modular and stepwise prompts

Once the project enters the implementation phase, a modular development approach is adopted. The LLM is instructed to generate code for each module incrementally, with each module undergoing validation for correctness and completeness. This decomposition of complex tasks helps prevent the model from failing to provide complete code due to its limitations. The prompts combine functional modules with project structure, requiring the LLM to output content according to defined standards.

3.2.3. Role-specific prompts

By assigning a role and working rules to the LLM, it is instructed to respond as a developer. The LLM is expected not only to generate code but also to explain its functions, ensuring the output is suitable for non-expert users and enhancing the readability of the code.

3.2.4. Constraint-based prompts

The user imposes clear structural constraints on the output. The LLM is explicitly prohibited from generating “pseudocode” or “simple examples,” and instead is required to ensure code executability and include annotations. It is also clearly instructed on the scope and content of audit tasks to be performed.

3.2.5. Review-oriented prompts

The LLM is given clearly defined code review responsibilities and execution details, including structured evaluation criteria, to conduct a formal code review process. This enhances the reliability and systematization of the review. Multiple models are employed in the evaluation to enable cross-validation.

DOI: 10.54254/2755-2721/2025.AST27451

3.3. LLM-based integrated method for evaluating generation quality and security

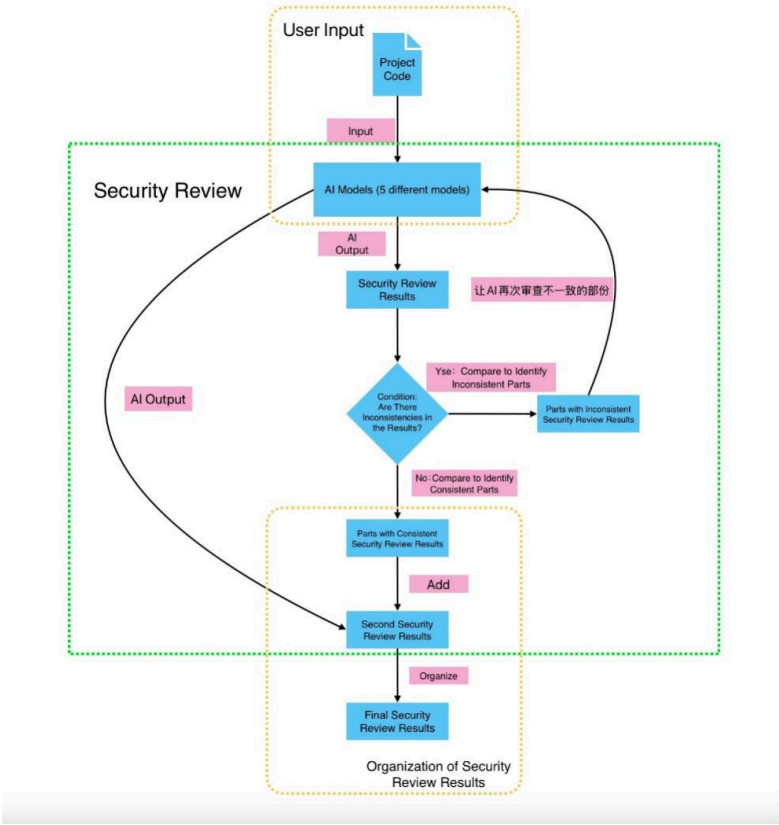

Figure 2. Workflow of the security review method

Since security review is a key focus of this study, the security auditing task employs five LLM models. All the LLMs currently used possess the capability to read and understand code files, ensuring that the results of the security review task are reliable and authentic. Due to certain constraints, the AI models participating in the security review task are: Doubao AI, DeepSeek AI, Qwen-3-Coder, ChatGPT, and Kimi-K2. The code files are uploaded to these large models along with explicit security review requirements, clearly defining the scope and limitations of the security assessment. The models then output their respective audit results. Upon organizing and comparing the review results from the five models, it is evident that the currently implemented security measures reported by the models are largely consistent. However, discrepancies arise regarding identified security vulnerabilities—the five models provide differing outputs on these issues. Subsequently, the inconsistent security issues identified by each model are extracted and posed back to the models as queries, asking each model to respond with a “yes” or “no” to whether it agrees with the inconsistencies relative to its own assessment. Finally, the outputs from all five models are consolidated to derive the final security audit results.

DOI: 10.54254/2755-2721/2025.AST27451

Table 1. Security review of LLM-based web generation

Issue Type	Common Issues	Specific Issues
JWT-related Issues	1. Improper key management (not managed via environment variables, default key fallback exists) 2. Missing token refresh mechanism 3. Excessively long token validity period Reported by Qwen, DeepSeek (DS), Kimi	1. Authentication failure depends on environment variable presence (Qwen) 2. Missing config module referenced by the project (Qwen)
Input Validation & Injection	1. Insufficient input validation, leading to XSS vulnerability 2. Presence of NoSQL injection risk Reported by DS, Doubao, Kimi, GPT	1. Product price not validated as numeric; order quantity not checked as positive integer (DS) 2. User comments lack content escaping/cleaning (GPT)
API Protection	1. Lack of rate limiting (vulnerable to brute-force attacks, DDoS) 2. Missing CSRF protection mechanisms Reported by DS, GPT	1. No secondary verification for sensitive operations (DS) 2. Registration, login, and order APIs lack request frequency limits (GPT)
Authorization & Data Security	1. Weak password strength/complexity 2. Risk of sensitive information leakage Reported by DS, Kimi, GPT	1. User permissions not isolated (normal users/admins, DS) 2. API responses contain MongoDB ObjectIDs (DS) 3. Coarse user permission control (lack of unified middleware, GPT) 4. Password hashes may be leaked due to improper error handling (Doubao)
Configuration Issues	1. CORS misconfiguration 2. Lack of HTTPS setup Reported by Qwen, Kimi, GPT	1. Incomplete server configuration (missing server.js, incorrect config module reference, Qwen)
Others	Environment variable management issues (missing .env file for sensitive data storage) Reported by Qwen, Kimi	1. Unlimited login failure attempts (DS) 2. Lack of security mechanisms for third-party login (to be addressed in future, Doubao) 3. Payment security concerns (to be addressed in future, Doubao) 4. Long token validity after JWT theft (Kimi) 5. Error responses enable user enumeration attacks (Kimi)

In Table 1, abbreviations are used for brevity in presentation: DS stands for DeepSeek AI, Qwen for Qwen-3-Coder, Kimi for Kimi-K2, GPT for ChatGPT, and Doubao for Doubao AI. From Table 1, it is evident that the five LLMs identified different security issues across various categories. They also commonly detected certain security problems during the initial review. Moreover, in the subsequent second round of security audits, each of the five models acknowledged the distinct security issues raised by the other four models.

DOI: 10.54254/2755-2721/2025.AST27451

4. Evaluation

4.1. Design requirement

Table 2. Website function description table

Function Category	Implemented Features	Typical Function?	Description
User Module	Email registration, user login	Yes	User authentication is a fundamental function of all web applications, especially e-commerce platforms where registration and login ensure user data and transaction security. It is a typical core feature.
Product Module	Product list query, single product detail query	Yes	Product browsing is the core entry point of e-commerce platforms. Users obtain product information through listings and detail pages. It is an essential function of all shopping platforms and considered typical.
Shopping Cart Module	Add product to cart, delete cart items, clear cart	Yes	The shopping cart serves as a temporary storage tool, allowing users to manage multiple products before purchase. It is a key step in the e-commerce transaction flow and considered typical.
Order Module	Create orders, query order list and details	Yes	The order system is central to the transaction lifecycle, recording purchase information and status. It underpins transactions on e-commerce platforms and is a typical function.

The website design requirements used in the experiment are summarized in Table 2. The implemented features represent the fundamental and core typical functions of e-commerce platforms, covering the basic transaction process from user login and product browsing to order creation. However, some typical and enhanced features (such as payment and filtering) have not been implemented, resulting in an overall "streamlined typical e-commerce framework." Therefore, this project serves as an objective, fair, and reasonable testbed for evaluating AI code generation capabilities.

4.2. Experimental setup

4.2.1. Test subjects

The free version of ChatGPT was used, with the model version identified as ChatGPT-4o in the output results. The auditing AIs included Doubao AI (client version), DeepSeek AI running the R1 inference model, Kimi, Qwen-3-Coder, all of which have code file reading and analysis capabilities implemented via an IDE platform, as well as ChatGPT-4o. In total, there were five auditing AIs and one test subject AI.

4.2.2. Quantitative evaluation metrics

The quality of the generated website was quantitatively assessed using the function implementation rate, calculated as follows:

$$Overall\ Achievement\ Rate = \sum(Module\ Achievement\ Rate \times Module\ Weight) \quad (1)$$

DOI: 10.54254/2755-2721/2025.AST27451

Module weights were assigned based on practical importance typical for e-commerce platform functionalities. The user module, representing personal user information, is a highly important and core part of the application. Product display and order management are indispensable functions for e-commerce platforms; thus, the product module and order module also occupy central roles in the web application. Conversely, social features—such as product sharing—are clearly less critical relative to other functions. Based on this rationale, module weights were assigned as follows: User Module (core account system): 30%. Product Module (platform foundation): 30%. Order Module (transaction closure key): 30%. Social Features (value-added services): 10%.

4.3. Results

Table 3. Comparison of audit results

Comparison Dimension	Existing Work (Based on Literature)	Results from This Project Audit
Function Implementation Rate	Most literature finds that LLM-generated code has low functional coverage [1,2,5].	Based on the aggregated audits from various LLMs, the function implementation rate is 41%, indicating many designed features remain unimplemented or only partially implemented.
Logical Completeness	Abbassi [3] found common logical incompleteness in generated code, with missing functions or module gaps.	Some functions failed to be implemented, and parts of the system were missing.
Code Executability	Liu [2] suggested that some LLM-generated code passes completion checks but is hard to run directly or integrate.	The project's multiple modules are runnable, but users had to correct deviations in structure, file paths, and module naming during development, as initial LLM outputs did not strictly follow the user-specified structure.
Security Awareness	Existing literature points out common issues like XSS, IDOR, weak authentication; security tools expose blind spots in model evaluation [2].	All five auditing LLMs identified potential vulnerabilities such as insufficient input validation, missing CSRF protection, and unsafe JWT usage.
Content Consistency	Yin [5] notes that LLMs often show structural misalignment and context inconsistency at the repository level.	There were issues like module naming confusion, output order errors, and duplicated functionality. LLMs improved context coherence only after user correction and prompting, lacking self-correction ability.
Development Efficiency	Anthropic and Business Insider report that non-technical users can rapidly generate prototypes, but experienced developers see limited productivity gains [8].	Users successfully completed the shopping website from design to development and audit via natural language interaction, but the process required substantial user control and intermediate intervention (e.g., constraints, error correction).
AI Self-Testing Capability	Jiang [1] and Liu [2] indicate current models lack automatic error correction and self-verification mechanisms.	GPT did not actively check its own code for errors or completeness; auditing required external user or tool involvement.

The analysis and summary of the experimental results are presented in Table 3, with detailed conclusions as follows:

DOI: 10.54254/2755-2721/2025.AST27451

4.3.1. Insufficient functional coverage and implementation rate

Combining findings from the literature and the empirical results of this project, incomplete functionality remains a prominent issue in ChatGPT-generated code and is among the most common problems with current large models. Even with a complete design plan and modular decomposition to reduce development complexity, LLMs still show very limited capability in handling complex, multi-layered requirements. According to the auditing AIs' statistics, under different weighting schemes, the overall function implementation rate fails to reach 50%.

4.3.2. Security issues in the empirical project closely align with literature findings

Security audits conducted by five different AI LLMs revealed that the security vulnerabilities in ChatGPT-generated code match the typical weaknesses identified by Liu and Abbassi. This indicates that although current LLM-generated code incorporates some security measures, it still lacks a comprehensive security defense system.

4.3.3. Development efficiency

While the literature notes that LLMs significantly improve development efficiency for non-technical users, this empirical test showed that model outputs still require progressive user-initiated error correction. Additionally, users need a foundational understanding of code development and project structure; otherwise, the output tends to be fragmented and modular rather than a complete project file. Hence, the results suggest that if a user without relevant knowledge interacts with the LLM, the quality of generated code is unlikely to be satisfactory.

5. Conclusion

This paper focuses on the application of large language models (LLMs) to assist web development, constructing a user-centered workflow framework in which website code is generated through natural language prompts and subsequently evaluated for functionality and security by other AI LLMs serving as auditing tools. Taking an e-commerce website as a case study, we systematically assessed the capabilities and limitations of large models in code generation. The results show that although the models can generate runnable modular code under structured prompts and support basic interactions, the overall function implementation rate is only 41%, with multiple core features missing or incomplete. Security audits also revealed typical issues in the generated code, such as insufficient input filtering, lack of CSRF protection, and weak authentication mechanisms.

These findings align closely with existing research conclusions regarding LLM code generation, further confirming the practical challenge that such code often appears usable but is difficult to deploy effectively. However, this study also validates a promising direction: by leveraging LLMs' powerful generation abilities combined with structured prompting and automated auditing, even non-professional users can complete prototype development and preliminary verification. Looking ahead, we believe the "generate + verify" paradigm will become the core of trustworthy LLM-assisted development workflows. Integrating language models with testing tools and security analysis systems holds great potential to advance human-led intelligent collaborative development toward practical, real-world adoption.

DOI: 10.54254/2755-2721/2025.AST27451

References

[1] Jiang, J., Wang, F., Shen, J., Kim, S., & Kim, S. (2024). A survey on large language models for code generation. arXiv preprint arXiv: 2406.00515.
[2] Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In Advances in Neural Information Processing Systems.
[3] Abbassi, A. A., Da Silva, L., Nikanjam, A., & Khomh, F. (2025). Unveiling inefficiencies in LLM-generated code: Toward a comprehensive taxonomy. arXiv preprint arXiv: 2503.06327.
[4] Gu, Q. (2023). LLM-based code generation method for Golang compiler testing. In Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 2201–2203).
[5] Yin, X., Ni, C., Li, X., Chen, L., Ma, G., & Yang, X. (2025). Enhancing LLM's ability to generate more repository-aware unit tests through precise contextual information injection. arXiv preprint arXiv: 2501.07425.
[6] Joel, S., Wu, J. J. W., & Fard, F. H. (2024). A survey on LLM-based code generation for low-resource and domain-specific programming languages. arXiv preprint arXiv: 2410.03981.
[7] Anonymous. (2025, July). Anthropic AI breakthrough vibe coding revolution 2025? Business Insider.
[8] Anonymous. (2025, July 11). AI coding tools made some engineers less productive. Business Insider.
[9] Wikipedia contributors. (2025, June 12). Slopsquatting. Wikipedia, The Free Encyclopedia. Retrieved from https: //en.wikipedia.org/wiki/Slopsquatting
[10] Wikipedia contributors. (2022, January 15). Hallucination (artificial intelligence). Wikipedia, The Free Encyclopedia. Retrieved from https: //en.wikipedia.org/wiki/Hallucination_(artificial_intelligence)