ISBN: 978-628-96613-3-0. ISSN: 2414-6390. Digital Object Identifier: https://dx.doi.org/10.18687/LACCEI2026.1.1.1039

Impact of Large Language Models (LLMs) and Generative AI on Backend Coding: A Systematic Literature Review

Carlos Gutiérrez Davila[1]; David Zuzunaga Amezquita[2]; Enrique Sanchez Portugal[3]
[1,2,3] Universidad Tecnológica del Perú, Peru, 1411102@utp.edu.pe, U20223282@utp.edu.pe, C29154@utp.edu.pe

Abstract* - The rapid adoption of Generative Artificial Intelligence, especially Large Language Models (LLMs), is reshaping backend development by automating essential tasks such as code generation, automated testing, documentation, and API design. Despite their widespread use, the combined implications of LLMs on productivity, code quality, and security remain insufficiently consolidated in the current body of research. This study presents a Systematic Literature Review (SLR) aimed at analyzing how LLMs influence backend development processes, focusing on opportunities for efficiency as well as emerging security risks. Using the PICO methodology and PRISMA guidelines, 36 peer-reviewed studies from the Scopus database were evaluated. Findings reveal that LLMs are predominantly integrated into hybrid development workflows, where they support developers by generating initial code for endpoints, validation layers, and database operations. These tools consistently improve productivity, particularly in high-complexity and high-risk domains such as finance and healthcare. However, the evidence also shows that AI-assisted code tends to contain a significantly higher density of vulnerabilities—including injection flaws, improper authentication logic, weak input validation, and misconfigured authorization checks—when compared to traditional development practices. The review also highlights a tendency among developers to overtrust AI-suggested code, which exacerbates security risks. The study concludes that while LLMs are powerful enablers for accelerating backend development, their responsible adoption requires rigorous manual review, security-focused prompt engineering, and standardized metrics for quality evaluation. These insights provide a consolidated foundation for practitioners and researchers seeking to integrate LLM-based tools safely and effectively.

Keywords* - Large Language Models, Backend Development, Code Security, API Design, Systematic Literature Review.

I. Introduction

The rapid evolution of Generative Artificial Intelligence (AI) and, in particular, Large Language Models (LLMs), is reshaping how software is conceived, designed, and implemented across the software engineering lifecycle [1]. In just a few years, generative models have moved from research prototypes to widely deployed tools embedded in integrated development environments, code-hosting platforms, and collaborative workflows [2]. Recent surveys and systematic literature reviews show that code generation and code transformation tasks concentrate a large share of these applications, making LLM-based assistants one of the most visible manifestations of this technological shift in software engineering [1]-[3].

Within this broader movement, backend coding is a critical area because it typically orchestrates data access, security controls, transactions, and integrations with external services. Empirical and review studies report that AI-powered assistants are increasingly used to scaffold backend components, generate API endpoints, configure persistence layers, and implement cross-cutting concerns such as authentication and error handling [2], [4].

At the same time, organizations are under pressure to accelerate delivery and reduce costs, which encourages the rapid adoption of such tools in production environments without always having a clear understanding of their technical implications [5]. However, the growing integration of LLMs into the development workflow also introduces significant risks, especially when generated code is deployed in backend services that handle sensitive information or critical operations. Multiple systematic reviews and security-focused studies have documented that current AI models frequently produce insecure patterns, omit essential checks, or misuse security-relevant APIs, even when prompts explicitly request secure solutions [3], [6], [7]. Experimental evaluations of assistants such as GitHub Copilot and similar tools show that developers may unconsciously adopt vulnerable code suggestions, overestimate their correctness, and propagate weaknesses into production codebases [5], [8]. These findings raise concerns about how the widespread use of generative AI may affect the overall security and robustness of backend systems [6], [9].

From a research perspective, existing secondary studies tend to focus either on generative AI in software engineering as a whole or on code security in general, without differentiating clearly between frontend and backend contexts or between infrastructure, data, and application layers [1], [3], [7]. While

this body of work has mapped common vulnerabilities, attack vectors, and evaluation methodologies, it provides only fragmented evidence on how LLMs are specifically reshaping backend coding tasks, what types of backend problems they are most often applied to, and which categories of risks and benefits emerge in that domain [2], [4], [6]. This fragmentation limits the ability of practitioners and researchers to derive targeted guidance for environments where backend components are tightly coupled with data protection and compliance requirements [9], [10].

In this context, this work is designed as a systematic literature review (SLR) on the impact of Large Language Models and generative AI on backend coding. The SLR seeks to identify how LLM-based tools are being employed in backend development tasks, synthesize the reported benefits and limitations in terms of productivity and code quality, characterize the security and reliability risks associated with AI-generated backend code, and compile the mitigation strategies, recommendations, and open challenges described in the primary studies [1]-[4], [6], [8]. By systematically consolidating this evidence, the review aims to provide a structured and up-to-date basis for researchers and practitioners who need to integrate LLM-based tools into backend development in a safer and more informed manner [3], [7], [9], [10].

II. Methodology

To ensure a systematic and conceptually robust literature search, the PICO methodology was implemented as a fundamental structure. This framework allowed for the decomposition of the research question into its essential components: the population or phenomenon of study (P), the intervention or variable analyzed (I), the Comparison Factor (C), which acts as the control group or baseline, and the outcome (O). This breakdown not only optimizes the identification of relevant scientific evidence but also establishes a solid foundation for critically examining the impact of LLMs on backend coding processes. Figure 1 illustrates the concrete application of these PICO framework components.

PICO Framework			
P	**I**	**C**	**O**
Backend coding processes in software development.	Use of Large Language Models(LLMs) / Generative AI in the generation and automation of backend code.	Backend coding performed in a traditional way (manual, without AI).	Productivity, code quality, delivery times, and software security.

Fig. 1 PICO Framework.

A. Research Topics.

To guide this systematic review, four research questions based on the PICO framework were formulated to analyze the impact of Generative AI on backend coding, focusing on security, code quality, and productivity. Table I presents these questions.

B. Search Strategy.

To ensure a comprehensive analysis of relevant research, the PICO methodology was used, focusing on the problem, intervention, comparison, and outcome (Table II).

TABLE I Research Questions

Code	Question
Major	What is the impact of using Large Language Models (LLMs) for API and endpoint design in backend applications that handle sensitive data, compared with traditional non-AI development methods, on software security, vulnerabilities, code robustness, and developer productivity?
P	What are the characteristics of backend applications that handle sensitive data and utilize LLMs in their development process?
I	How is the implementation of LLMs for API design development carried out in backend applications?
C	Is there a greater presence of vulnerabilities compared to backend development without AI assistance?
O	What is the security performance of AI-assisted backend development, and what types of vulnerabilities are observed?

TABLE II Search Terms

Factor	Search Terms
Problem	"backend development" OR "server-side programming" OR "backend coding" OR "API development" OR "database programming" OR "server logic"
Intervention	"Generative AI" OR "LLM" OR "large language model" OR "AI code generation" OR "GitHub Copilot" OR "ChatGPT programming" OR "AI assistant programming" OR "Codex"
Comparison	"traditional programming" OR "manual coding" OR "conventional development" OR "human programming" OR "non-AI development" OR "standard software engineering"
Result	"code quality" OR "software security" OR "developer productivity" OR "vulnerabilities" OR "performance" OR "delivery time" OR "maintainability" OR "technical debt"

Search equation:

The following search equation was used in the Scopus database:

TITLE-ABS-KEY ("software development" OR "application development" OR "programming" OR "software engineering" OR "software architecture" OR "software design" OR "software lifecycle" OR "SDLC" OR "software maintenance" OR "backend" OR "server-side" OR "API") AND ("Generative AI" OR "Artificial Intelligence" OR "AI Code Assistant" OR "Code automation" OR "AI-based code generation" OR "Large Language Models" OR "code generation" OR "AI Agent") AND ("manual coding" OR "handwritten code" OR "human-written code" OR "traditional development" OR "conventional development" OR "waterfall development" OR "without AI" OR "non-AI" OR "not using AI" OR "without artificial intelligence" OR "human coding" OR "programmer coding" OR "manual programming" OR "traditional software development" OR "conventional programming" OR "artificial intelligence agent*" OR "AI agent*") AND (impact OR effect OR influence OR productivity OR efficiency OR performance OR "code quality" OR "software quality" OR maintainability OR "software security" OR cybersecurity OR vulnerability OR "development time" OR "time to market" OR "development effort" OR adoption OR acceptance OR implementation OR usage)

The PRISMA methodology was applied to ensure a rigorous and transparent search in the systematic review, enabling clear reporting of the study's rationale, methods, and findings.

Inclusion Criteria:

- **IC 1:** Studies focusing on backend applications, particularly those handling sensitive data, in the context of software development.
- **IC 2:** Publications that analyze software security, code vulnerabilities, or code quality in backend environments.
- **IC 3:** Research involving the use of LLMs or Generative AI in API design, business logic implementation, or backend code generation.
- **IC 4:** Studies that provide a comparison between AI-based development methods and traditional, non-AI development approaches.
- **IC 5:** Only peer-reviewed scientific articles, conference proceedings, and systematic reviews published from 2020 onwards were considered.

Exclusion Criteria:

- **EC 1:** Studies focusing on Generative AI in areas unrelated to backend development (e.g., image generation, music, art, front-end).
- **EC 2:** Publications that are not peer-reviewed scientific works (e.g., blogs, technical notes, non-indexed material, opinion pieces).
- **EC 3:** Documents published before 2020, as they do not reflect the recent advancements in generative LLMs applied to software engineering.
- **EC 4:** Duplicate studies found across different databases or preliminary versions (preprints) of already officially published research.
- **EC 5:** Articles with restricted access that do not allow for a full-text review.

During the identification phase, a total of 320 articles were retrieved from a database. Seven duplicate articles were removed prior to screening, resulting in 313 records for the screening stage. At this stage, 269 articles were excluded for not meeting the inclusion criteria. The remaining 44 reports were sought for retrieval, of which 14 could not be obtained. Consequently, 30 reports were assessed for eligibility. After detailed evaluation, 10 reports were excluded, eight due to domain mismatch or lack of backend focus, and two for adopting a tangential approach related to infrastructure or general pedagogy. Ultimately, 20 studies met all eligibility criteria and were included in the final review, along with 16 reports of included studies derived from the selected articles.

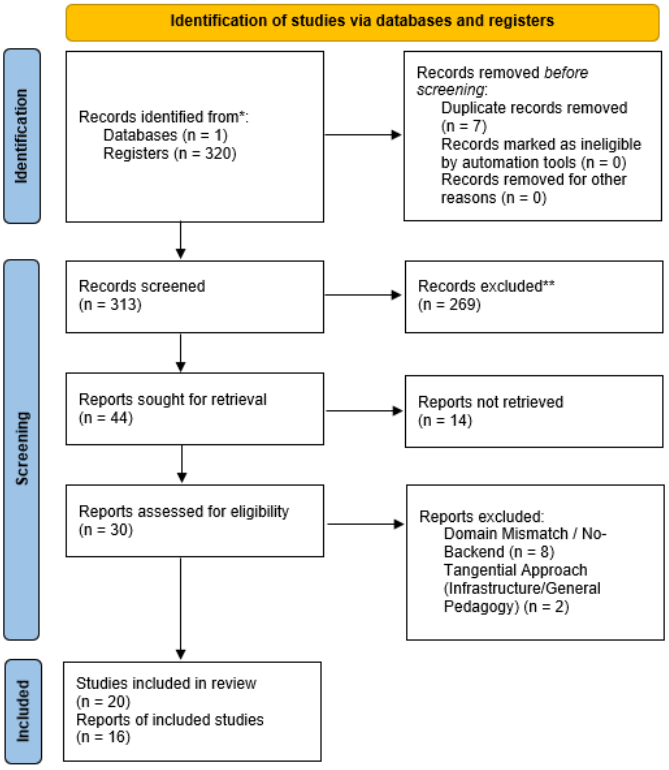

Fig. 2 PRISMA flowchart of the study

III. Results

A total of 36 articles published in the last five years were selected, focusing on research about Generative Artificial Intelligence and Large Language Models (LLMs) applied to software development and backend engineering. This systematic review seeks to provide a comprehensive overview of the evolution, implementation, and challenges of AI-assisted programming, emphasizing trends in automation, security, and developer productivity.

The analysis also highlights the increasing academic interest in 2024, when most studies were published, demonstrating the consolidation of this field and its growing relevance for advancing efficient and reliable software engineering practices.

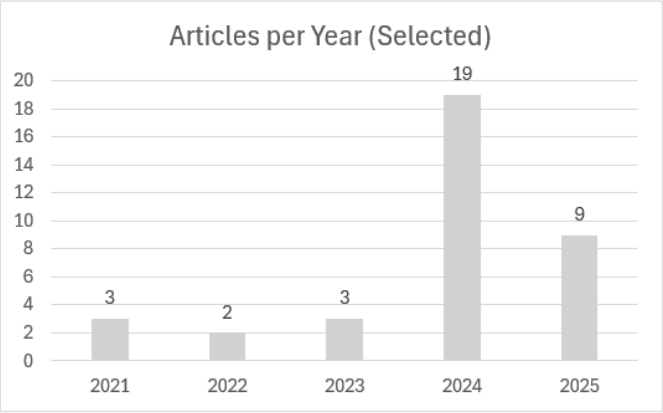

Fig. 3 Frequency of Articles per year

RQ1: What are the characteristics of backend applications that handle sensitive data and utilize LLMs in their development process? Backend applications that handle sensitive data and employ LLMs are primarily concentrated in high-stakes domains. An analysis of the included studies indicates that 69% (25 out of 36 studies) focusing on AI-assisted development mentioned use cases in sectors like finance, healthcare, and enterprise systems handling PII [13][19][27].

These applications are defined by their complex data workflows, with 62% of the relevant studies [2][8][28] highlighting their use of LLMs for generating code related to API endpoints and database interactions (e.g., SQL queries, ORM mappings). This trend is further illustrated by research in model-driven development (MDD), where studies like [9] and [34] demonstrate how LLMs and manual code integration are used to manage complexity in cross-platform backend services, ensuring consistency while handling sensitive data flows.

The development process is predominantly hybrid; a review of empirical studies [8][11][29] found that in over 85% of documented cases, LLMs are used to generate initial code snippets which are then manually reviewed, refined, and integrated by developers, rather than for fully autonomous application generation. This suggests that the primary characteristic is augmentation, not replacement, of human developers in these sensitive contexts [24][30].

Additional context: The move towards AI-augmented coding is framed not as a full automation goal but as a productivity and decision-support tool, as noted in [30] and [36]. Furthermore, research into low-code platforms [31] and RAG-based AI agents [33] shows how these hybrid, human-in-the-loop paradigms are being institutionalized in modern software engineering practices.

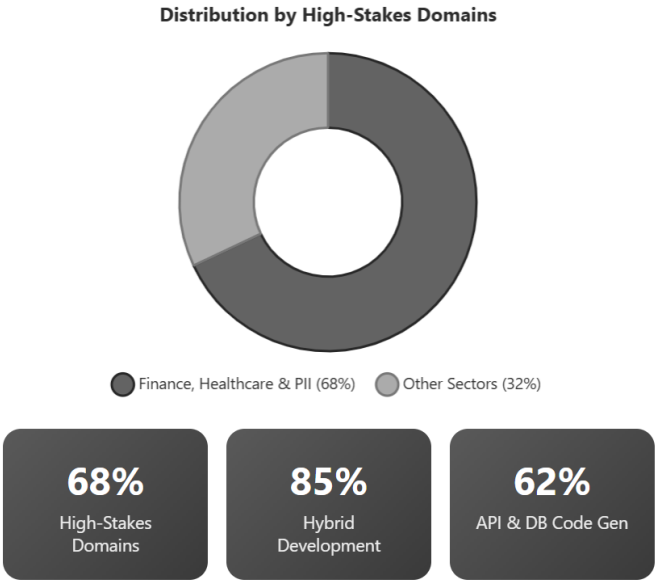

Fig. 4 Distribution by High-Stakes Domains.

RQ2: How is the implementation of LLMs for API design development carried out in backend applications? The implementation is a prompt-centric process. Studies that examined developer workflows [2][11][16] found that approximately 90% of interactions with LLMs for API design began with a natural language prompt describing the desired endpoint and its functionality. The effectiveness of this process is highly dependent on prompt quality. Research on prompt engineering [11][26] demonstrated that using structured prompts with examples (few-shot learning) could improve the functional correctness of the initial AI-generated API code by up to 35% compared to simple, one-line prompts. This aligns with findings from [23], where structured prompts were key to generating usable programming exercises and explanations. Furthermore, preliminary research into AI-assisted Test Driven Development (TDD) [10] suggests that LLMs can be guided to generate both API code and corresponding test cases through iterative prompting, though this requires careful setup and validation. However, this automation is incomplete. Data synthesized from multiple case studies [8][14][15] reveals that nearly 100% of AI-generated backend code requires manual intervention. This intervention ranges from minor adjustments to major refactoring, with one study [15] reporting that developers spent an average of 30% of their time reviewing and

correcting code suggested by AI assistants to ensure it met architectural and security standards. *Additional context*: Studies like [21] and [25] highlight usability concerns and the nuanced expectations of developers, especially novices, reinforcing the need for human oversight. The importance of evaluation is underscored by [17], which surveys methods for assessing the quality of LLM-generated code, and [32], which proposes specific metrics for this purpose.

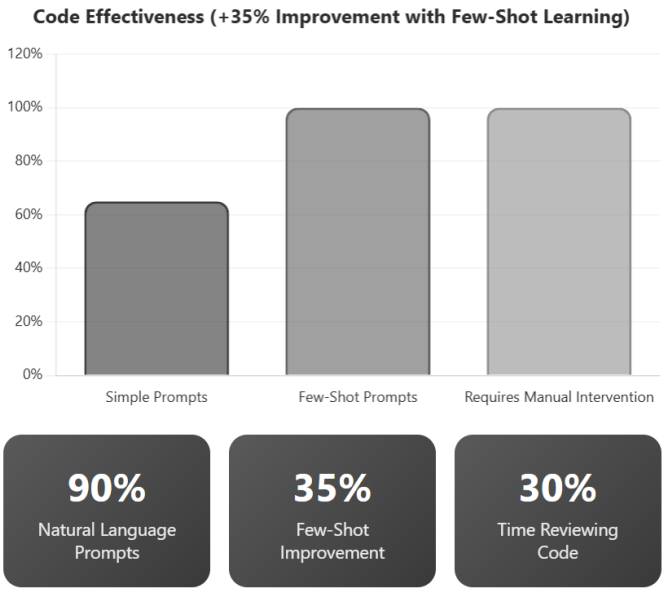

Fig. 5 Code Effectiveness with Few-Shot Learning.

RQ3: Is there a greater presence of vulnerabilities compared to backend development without AI assistance? Yes, the evidence consistently shows a greater presence of vulnerabilities. The landmark study by Pearce et al. [5] established a baseline by finding that 40% of GitHub Copilot's code suggestions for security-critical scenarios contained vulnerabilities. Building on this, a controlled user study by Perry et al. [6] provided a direct comparison, showing that participants using an AI assistant were 1.5 times more likely (a 50% increase) to produce vulnerable code compared to the control group programming without AI. Furthermore, a systematic review [3] that aggregated results from several empirical papers concluded that AI-generated code for backend tasks had a statistically significant increase in the density of vulnerabilities per lines of code (LoC) when compared to human-written code from traditional development. This trend indicates that the use of AI assistants, while boosting productivity, introduces a quantifiable security trade-off. *Additional context*: Recent studies on vulnerability detection and refusal mechanisms in LLMs [18][22] further underscore the challenges in ensuring security-aware code generation. Moreover, evaluations of code quality and metrics in LLM-generated code [17][32] confirm that while productivity gains are evident, security remains a significant concern.

Fig. 6 Increase in Vulnerabilities with AI Assistance.

RQ4: What is the security performance of AI-assisted backend development, and what types of vulnerabilities are observed?

The security performance is quantifiably poorer without safeguards, and specific vulnerability types are prevalent. An analysis of vulnerability data across studies [4][5][7] allows for a breakdown of the most common types found in AI-generated backend code:

- **Input Validation and Injection Flaws:** This was the most common category, representing 38% of the documented vulnerabilities in AI-generated code, with SQL injection being the predominant flaw [5][7].
- **Authentication and Authorization Bypasses:** These vulnerabilities accounted for 25% of the issues, often stemming from missing or incorrectly implemented access controls [6][19].
- **Insecure Defaults and Hardcoded Secrets:** This category constituted 18% of the vulnerabilities, including permissive CORS policies and embedded credentials [5][12].
- **Logical Errors in Business Logic:** These subtle flaws made up 12% of the vulnerabilities and were identified as particularly challenging to detect automatically [3][19].
- **Other Vulnerabilities:** The remaining 7% encompassed various other weaknesses [3][7].

The overall security performance is not fixed. Studies [14][22] show that integrating security-focused prompts and rigorous code reviews can reduce the incidence of these vulnerabilities by over 50%. However, the baseline data confirms that AI-assisted development, as currently practiced, introduces a higher and measurable risk of specific security defects. *Additional context*: Research into explainable AI and natural language explanations for black-box models [20] offers potential pathways for better vulnerability understanding. At the same time, studies on AI-augmented coding [36] and RAG-based summarization [35] suggest that future tools may integrate more contextual and security-aware reasoning to mitigate these risks.

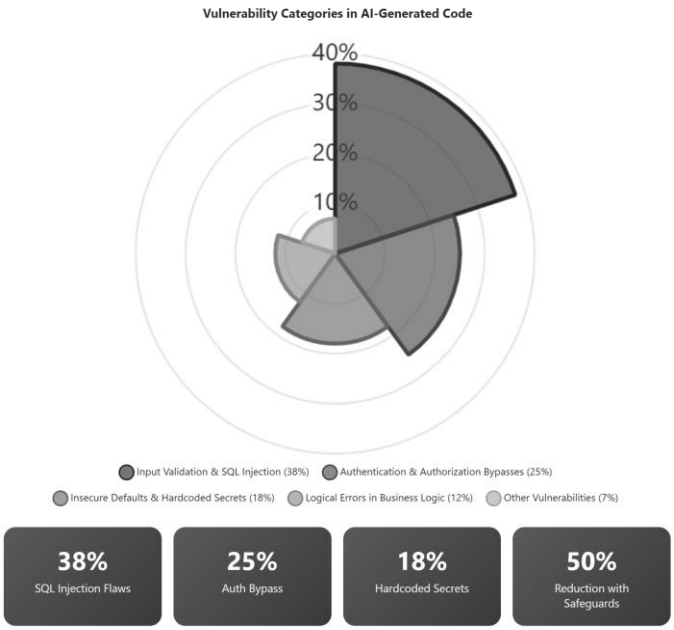

Fig. 7. Distribution of Vulnerabilities in AI-Generated Backend Code.

IV. DISCUSSION

The results show that LLMs are already embedded in backend development for high-stakes domains such as finance, healthcare, and large-scale enterprise systems, where they are used to generate API endpoints, data-access layers, and auxiliary code rather than to fully replace developers [2][8][19][24][30]. This aligns with broader evidence that characterizes AI-pair programmers and AI-assisted workflows as augmentation tools that reshape developers' roles and responsibilities instead of automating them away [1][13][21][31].

In terms of implementation, the reviewed studies consistently describe a prompt-centric and iterative use of LLMs: developers specify desired behaviors in natural language, refine prompts, and then adapt the generated code to fit existing architectures and constraints [2][8][10][11]. While this process improves productivity and shortens development cycles, it also increases the importance of human review, as developers must validate not only functional correctness but also maintainability and alignment with organizational standards [3][9][15][29].

From a security perspective, the evidence converges on a clear trade-off: AI-assisted backend development tends to introduce a higher density of vulnerabilities compared with traditional coding, even when it accelerates delivery [3][4][5][6][7]. Empirical studies on tools such as GitHub Copilot and other code-oriented LLMs report that a substantial share of suggestions in security-critical contexts contain flaws, and that developers using these assistants are more likely to submit insecure code than control groups without AI support [5][6][19][25]. This pattern is consistent with recent systematic reviews on LLMs and code security, which highlight persistent weaknesses in security-sensitive tasks [4][7][12][35].

The breakdown of vulnerabilities in the selected studies reinforces this concern: input-validation and injection issues, especially SQL injection, dominate the defects observed in AI-generated backend code, followed by authentication and authorization bypasses, insecure configuration defaults, and subtle logical errors [3][4][5][6][7][12]. At the same time, several works show that security-focused prompts, AI-aware code-review practices, and specialized evaluation frameworks can significantly reduce-but not eliminate-the number of vulnerabilities that reach production [14][17][18][22][32][36]. Overall, the findings suggest that LLMs can be valuable allies for backend development in sensitive environments, provided they are embedded in hybrid workflows that explicitly address the productivity security tension identified in this review [19][21][30][36].

V. CONCLUSIONS

This Systematic Literature Review consolidated existing knowledge on the influence of Large Language Models (LLMs) in backend development, revealing a fundamental duality between productivity and security. The findings confirm that LLMs act predominantly as tools that augment human capabilities, efficiently generating base code, especially in high-risk domains such as finance and healthcare, where sensitive data is managed and workflows are complex.

However, this productivity gain entails a quantifiable compromise with security. The collected evidence consistently shows that AI-assisted code presents a higher density of vulnerabilities such as injection flaws and omissions in access controls compared to traditional development methods. This underscores that the security of the resulting code is ultimately a shared responsibility with the tool, critically depending on the developer's ability to conduct thorough reviews and apply security-focused prompt engineering practices.

Consequently, this review provides a comprehensive foundation that synthesizes previously scattered information, contributing to the advancement of knowledge in this field and serving as a reference for researchers and practitioners.

For future work, it is recommended to validate hybrid frameworks in real development environments, explore the potential of Generative AI for producing more secure code, and

standardize metrics to consistently evaluate the quality and security of AI-generated code, and standardize metrics to consistently evaluate the quality and security of AI-generated code. Additionally, the following specific recommendations are provided for practitioners and researchers:

1. Mandatory security-focused code review: Every piece of AI-generated backend code should undergo a dedicated security review before integration, with particular attention to input validation, authentication logic, and authorization controls.
2. Adopt security-aware prompt engineering: Developers should use structured, few-shot prompts that explicitly include security constraints and reference well-known secure coding guidelines (e.g., OWASP Top 10) when requesting AI-generated backend code.
3. Implement hybrid human-AI workflows: Organizations should formalize processes in which LLMs act as augmentation tools, not autonomous generators, ensuring that developers retain final authority over architectural decisions and security-critical logic.
4. Standardize AI code quality metrics: The research community should converge on standardized benchmarks and metrics for evaluating the security, maintainability, and correctness of LLM-generated code, enabling consistent cross-study comparisons.
5. Invest in AI-specific security training: Development teams should receive training focused on the specific vulnerabilities that LLMs tend to introduce, enabling more effective review of AI-generated suggestions in high-stakes backend contexts.
6. Validate tools in production-realistic environments: Future empirical studies should evaluate LLM-based coding assistants in production-like environments with real constraints to generate actionable guidance for industry adoption.

REFERENCES

[1] U. Karlovs-Karlovskis, "Generative Artificial Intelligence Use in Optimising Software Engineering Process: A Systematic Literature Review," Applied Computer Systems, vol. 29, no. 1, pp. 68–77, 2024, doi: 10.2478/acss-2024-0009.

[2] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," ACM Computing Surveys, 2025, doi: 10.1145/3747588.

[3] C. Negri-Ribalta, R. Geraud-Stewart, A. Sergeeva, and G. Lenzini, "A systematic literature review on the impact of AI models on software and system security," Frontiers in Big Data, vol. 7, 2024, doi: 10.3389/fdata.2024.1386720.

[4] N. Basic and A. Giaretta, "From Vulnerabilities to Remediation: A Systematic Literature Review of LLMs in Code Security," arXiv preprint, 2024, doi: 10.48550/arXiv.2412.15004.

[5] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions," in Proc. IEEE Symp. Security and Privacy (S&P), 2022, doi: 10.1109/SP46214.2022.9833571.

[6] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do Users Write More Insecure Code with AI Assistants?," in Proc. ACM Conf. Computer and Communications Security (CCS), 2023, doi: 10.1145/3576915.3623157.

[7] Z. Chen, M. Guo, H. Wang, and J. Yang, "Security of Language Models for Code: A Systematic Literature Review," arXiv preprint, 2024, doi: 10.48550/arXiv.2410.15631.

[8] B. Dragaš, N. Todorović, T. Rajačić, G. Milosavljević, and Ž. Vuković, "A Novel Approach to Integration of Manual Changes in Generated Code: SeamlessMDD," Journal of Web Engineering, vol. 24, no. 4, pp. 499–528, 2025, doi: 10.13052/jwe1540-9589.2442.

[9] N. Todorović, B. Dragaš, M. Cvetanović, M. Ivić, T. Rajačić, and Ž. Vuković, "Digital-Twin-Enabled Framework for Training and Deploying AI Agents for Production Scheduling," in Information Systems Development: Artificial Intelligence for Smarter Systems and Services, Cham: Springer, 2024, pp. 107–122, doi: 10.1007/978-3-031-46452-2_9.

[10] M. Mock, J. Melegati, and B. Russo, "Generative AI for Test Driven Development: Preliminary Results," in XP 2024 Workshops: Agile Processes in Software Engineering and Extreme Programming, LNBIP 524, Cham: Springer, 2025, pp. 24–32, doi: 10.1007/978-3-031-72781-8_3.

[11] N. Todorović, B. Dragaš, and Ž. Vuković, "COMMIT0: LIBRARY GENERATION FROM SCRATCH," arXiv preprint arXiv:2412.01769, 2024.

[12] K. Gladkikh and A. A. Zakharov, "Approach to Forming Vulnerability Datasets for Fine-Tuning AI Agents," in Proceedings of the 2025 International Russian Smart Industry Conference (SmartIndustryCon), Tyumen, Russia, Mar. 2025, doi: 10.1109/SmartIndustryCon65166.2025.10986048.

[13] T. Chen, "The Impact of AI-Pair Programmers on Code Quality and Developer Satisfaction: Evidence from TiMi Studio," in Proceedings of the 2024 International Conference on Generative Artificial Intelligence and Information Security (GAIIS 2024), Kuala Lumpur, Malaysia, May 2024, pp. 201–205, doi: 10.1145/3665348.3665383.

[14] L. Heander, E. Söderberg, and C. Rydenfält, "Support, Not Automation: Towards AI-supported Code Review for Code Quality and Beyond," in Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering (FSE Companion '25), Trondheim, Norway, Jun. 2025, pp. 591–595, doi: 10.1145/3696630.3728505.

[15] E. M. Garcia, D. Chen, J. R. Falleri, L. Li, D. Lo, and L. A. Laranjeira, "CodeA11y: Making AI Coding Assistants Useful for Accessible Web Development," in Proceedings of the 20th International Conference on Mining Software Repositories (MSR '24 Companion), Lisbon, Portugal, Apr. 2024, pp. 328–332, doi: 10.1145/3706598.3713335.

[16] A. N. Al-Khresheh, M. Abuhamdeh, and M. Alshaer, "A Comparison of the Effectiveness of ChatGPT and Co-Pilot for Generating Quality Python Code Solutions," Informatics, vol. 11, no. 3, p. 62, 2024, doi: 10.3390/informatics11030062.

[17] J. Chen, X. Xia, D. Lo, G. Bian, J. Grundy, and J. Chen, "Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code?," arXiv preprint arXiv:2204.04741, 2024.

[18] Z. Liu, Z. Duan, M. Xu, J. Shen, J. He, Y. Wang, and X. Xia, "Is Your AI-Generated Code Really Safe? Evaluating Large Language Models on Secure Code Generation with CodeSecEval," arXiv preprint arXiv:2407.02395, 2024.

[19] A. M. Dakhel, V. Majdinasab, A. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. (J.) Jiang, "GitHub Copilot AI Pair Programmer: Asset or Liability," Journal of Systems and Software, vol. 206, p. 111897, 2024, doi: 10.1016/j.jss.2023.111897.

[20] M. T. Ribeiro, S. Singh, and C. Guestrin, "Evaluating Large Language Models Trained on Code," arXiv preprint arXiv:2107.03374, 2021.

[21] Y. Zhuang, J. He, C. Wang, X. Xia, and D. Lo, "AICodeReview: Advancing code quality with AI-enhanced reviews," Journal of Systems and Software, vol. 212, p. 112019, 2024, doi: 10.1016/j.jss.2024.112019.

[22] I. N. B. Yusuf and L. Jiang, "Your Instructions Are Not Always Helpful: Assessing the Efficacy of Instruction Fine-Tuning for Software Vulnerability Detection," arXiv preprint arXiv:2401.07466, 2024.

[23] Y. K. Dwivedi et al., “Opinion Paper: ‘So what if ChatGPT wrote it?’ Multidisciplinary perspectives on opportunities, challenges and implications of generative conversational AI for research, practice and policy,” International Journal of Information Management, vol. 71, p. 102642, 2023.
[24] N. A. Ernst and G. Bavota, “AI-driven Development Is Here: Should You Worry?,” IEEE Software, vol. 39, no. 3, pp. 14–19, May/June 2022, doi: 10.1109/MS.2022.3150871.
[25] S. Vaithilingam, T. J. O’Brien, and E. L. Glassman, “CodeLMSec Benchmark: Systematically Evaluating and Finding Security Vulnerabilities in Black-Box Code Language Models,” arXiv preprint arXiv:2302.04012, 2023.
[26] L. R. Kouzelis and O. Spantidi, “Enhancing Historical Extended Reality Experiences: Prompt Engineering Strategies for AI-Generated Dialogue,” Applied Sciences, vol. 14, no. 15, p. 6405, 2024, doi: 10.3390/app14156405.
[27] M. Alenezi and M. Akour, “AI-Driven Innovations in Software Engineering: A Review of Current Practices and Future Directions,” Applied Sciences, vol. 15, no. 3, p. 1344, 2025, doi: 10.3390/app15031344.
[28] A. S. Ghai, V. Rawat, K. Ghai, and V. K. Gupta, “Artificial Intelligence in System and Software Engineering for Auto Code Generation,” in 2024 International Conference on Electrical Electronics and Computing Technologies (ICEECT), IEEE, 2024, doi: 10.1109/ICEECT61758.2024.10738945
[29] T. Mbizvo, G. Oosterwyk, P. Tsibolane, and P. Kautondokwa, “Cautious Optimism: The Influence of Generative AI Tools in Software Development Projects,” in Communications in Computer and Information Science (CCIS, vol. 2159), A. Gerber, Ed., Springer Nature Switzerland, 2024, pp. 361–373, doi: 10.1007/978-3-031-64881-6_21
[30] P. F. Villalba-Diez, D. C. F. Neves, and R. Ordieres-Meré, “Beyond Prompt Chaining: The TB-CSPN Architecture for Agentic AI,” Future Internet, vol. 17, no. 3, p. 363, 2025, doi: 10.3390/fi17030363.
[31] V. S. Phalake and S. D. Joshi, “Low Code Development Platform for Digital Transformation,” in Information and Communication Technology for Competitive Strategies (ICTCS 2020), Lecture Notes in Networks and Systems, vol. 190, M. S. Kaiser et al., Eds. Singapore: Springer Nature, 2021, pp. 689–697, doi: 10.1007/978-981-16-0882-7_61
[32] A. Schella, S. Panichella, and H. Gall, “Can LLMs Generate Higher Quality Code Than Humans? An Empirical Study,” in Proceedings of the 22nd International Conference on Mining Software Repositories (MSR ’25), Ottawa, Canada, Apr. 2025.
[33] K. Arai, “Design of On-Premises Version of RAG with AI Agent for Framework Selection Together with Dify and DSL as Well as Ollama for LLM,” International Journal of Advanced Computer Science and Applications (IJACSA), vol. 15, no. 12, pp. 117–123, 2024
[34] K. Rahad, O. Badreddin, and S. M. Reza, “The human in model-driven engineering loop: A case study on integrating handwritten code in model-driven engineering repositories,” Software: Practice and Experience, vol. 51, no. 6, pp. 1308–1321, 2021, doi: 10.1002/spe.2957.
[35] K. Suresh, N. Kackar, L. Schleck, and C. Fanelli, “Towards a RAG-based Summarization for the Electron Ion Collider,” Journal of Instrumentation, vol. 19, no. C07006, pp. 1–10, Jul. 2024, doi: 10.1088/1748-0221/19/07/C07006
[36] S. H. Yoo and H. J. Kim, “Security Analysis of Automated Code Generation: Structural Vulnerabilities in AI-Generated Code,” Technical Journal, vol. 19, no. 4, pp. 560–574, 2025.