Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 3, Issue 12, 2025

EVALUATING THE IMPACT OF AI CODE GENERATED OF SOFTWARE QUALITY AND OBSERVED STUDY OF MAINTAINABILITY, SECURITY AND PERFORMANCE

Asfandyar[*1], Muhammad Waseem Akhtar[2], Hafiz Shoaib Khalil[3]

[*1]*Laravel developer, Muslim Youth University Islamabad*
[2]*Student MPhil, National college of business and administration and economic University Multan campus*

[*1]iasfandyar124@gmail.com, [2]waseemakhtargill906@gmail.com, [3]shoaibmalik.mtn@gmail.com

DOI: https://doi.org/10.5281/zenodo.18172279

Keywords
Artificial general intelligence, software quality, empirical study, code maintainability, software security, code performance, large language models (LLMs).

Article History
Received: 30 October 2025
Accepted: 18 December 2025
Published: 31 December 2025

Corresponding Author: *
Asfandyar

Abstract
The fast adoption of AI-driven code generation tools in the software development industry has potential to bring many gains in productivity but the effect that the technology has on the most basic, non-functional software quality features has not been properly measured yet. Although the current body of research mainly analyzes the functional correctness, the study is the first empirical and systematic evaluation of the impact of AI-generated code on the three pillars of the software quality: maintainability, security, and performance. We performed a comparative analysis with control, producing 642 solutions of code with state-of-the-art models (GPT-4, Claude 3) and 107 solutions created by human programmers experiencing at least 10 years of experience in creating a curated set of 107 programming tasks of various levels of complexity. Both solutions were hardened with tooling of industry standards: maintainability with SonarQube and Code Climate, vulnerability with security scanners (Bandit, Sempre) and a manual audit and performance with profiling (profile, memory-profiler). We find that there are systematic failures of quality in AI-generated code. It has a 34% greater cyclomatic complexity and 2.1 times greater duplication of code, which points to poor maintainability. In the 22.1% of AI samples there are security vulnerabilities due to the OWASP Top 10, versus 8.4% of human code. AI performance benchmarks indicate that the code is 15-40 percent slower and consumes 25 percent+ more memory. Most importantly, the complexity of the task is closely associated with quality degradation (the spearman 0.78 in the best case). We also find that, as they are currently constructed, AI models are functionally sound but create a quality debt that can be quantified in terms of duplicating patterns without understanding of architecture. This requires a paradigm shift: AI output should be considered as material that needs to be distrusted as draft, there have to exist better quality gates, special tooling and long-term human supervision to ensure the software remains healthy in the long-term.

INTRODUCTION

The introduction of Large Language Models (LLM) has caused a paradigm shift in the development of software. Model-driven tools such as GitHub Copilot, which is built on OpenAI Codex, have become a ubiquitous assistant in the integrated development environment of

developers, which fundamentally changes the process of programming (Chen et al. 2021; Kumar and Sharma 2023). Such a change has guaranteed unmatched improvements in developer productivity since the models can produce natural language descriptions into functionally plausible code, thus reducing the barrier to entry and automating mundane tasks (Kotsiantis et al. 2024). Thus, an emerging research agenda has focused on assessing the functional correctness of LLM-generated code, and has created strict criteria to assess whether the code actually works on a particular specification (Liu et al. 2023; Chen et al. 2024).

Nonetheless, this one-sided interest in functional correctness hides a more malevolent and even expensive phenomenon, the silent build-up of software quality debt. In professional software engineering, it is only the beginning to provide a working program, but the key metrics of long-term viability are quality attributes: maintainability, security, and performance. It is maintainable code that may be comprehended, adapted and expanded by later developers; it is exploit-resistant code; and it is economic code that utilizes resources effectively. These are non-functional requirements, and they do not happen by chance, but rather form the core aspect of the overall cost of ownership, scalability, and reliability of software systems (ISO/IEC 25010:2011).

Some recent empirical studies are preliminary and worrying evidence that the AI-generated code boom is potentially being bought at the long-term quality cost with immediate functionality. Initial studies of, among others, GitHub Copilot have found that there is a high level of robustness failure as syntactically correct code does not work with small changes in either input or context (Mastropaolo et al. 2023). More importantly, research on the subject of security has already reported an alarming tendency of these models to produce code with known vulnerability patterns, e.g. those listed by the OWASP Top Ten, even when fed benign specifications (Pearce et al. 2022). The article by Yetiştiren et al. (2023) is a significant contribution, as it empirically analyzes the measures of quality of a code to a group of several AI assistants, but their work is mostly based on a static analysis of a limited scope without integrated, operational tests of the performance or a comprehensive, contextual security audit in terms of security. In like manner, domain-specific assessments, e.g., in High-Performance Computing (HPC) point to the fact that models may produce parallel code kernels, although their performance properties are usually not optimum when compared to expert code (Godoy et al. 2023). This implies a model propensity to reproduce typical syntactic patterns of the training information without a more profound, structural insight of productiveness, security ramifications, or long-term sustainability issues of interconnection and solidarity (Mavridou et al. 2025).This literature demonstrates a research gap: although research has started to peel the tip of the iceberg in terms of individual quality attributes, other research has not been able to provide a holistic, empirical and comprehensive appraisal of the three maintainability, security and performance. The available literature is scattered away, looking at aspects individually without looking at their dependence i.e. how a code structure hard to maintain (low maintainability) also makes security audit and integration of performance optimization hard. This dissonance causes a lack of an evidence-based, homogeneous understanding of the real trade-offs at scale of AI code generation adoption by practitioners. In order to fill this gap, the paper presents a controlled and empirical study that will address the following research questions:

RQ1: What is the comparison of the AI-generated code with the functionally equivalent human-written code on the measurements of established metrics of software maintainability (e.g., cyclomatic complexity, code duplication, adherence to conventions) that are functional (not dependent on output metrics)

RQ2: How common and what are the characteristics of security vulnerabilities being introduced in AI-generated code, which are identified by automated scanners as well as by human, context-sensitive inspection.

RQ3: What is the performance of AI-generated code (in terms of runtime performance (execution time, memory consumption) and algorithmic efficiency) in comparison to human code.

We provide contributions in three aspects:

1. A multi-dimensional empirical data of comparing AI and human code on a curated benchmark of programming tasks
2. A quantitative and qualitative study of deficiencies in AI generated code across all three quality pillars
3. Recommendations and suggestions of how AI code generators can be integrated into software development life cycles without undermining its fundamental quality principles.

The results force the change of mindset: AI is not an independent engineer but an effective, but error-prone assistant the results of which are to be verified according to the strict and quality-related criteria.

2. Methodology

This experimental design is a controlled and comparative experiment that compares AI generated code to human written code on a curated set of programming tasks and evaluates each solution through a multi-faceted quality evaluation pipeline.

2.1. Experimental Design

To have strong and generalizable results, we adhered to the standard guidelines of experimentation in software engineering (Baltes and Ralph, 2022). We will design it as a comparative A/B study whereby the independent variable will be the source of the code generation (AI model or a human developer), and the dependent variables will be the quality measures of maintainability, security and performance. A solution to a certain programming problem that is correct and works on a specific problem is a fundamental unit of analysis. This method is direct to the level of response to our research question since it is possible to statistically compare the two cohorts.

2.2. Benchmark Task Suite

In order to test code generation in natural but controlled conditions, we have built a benchmark of 107 programming tasks. The purpose of this suite was to go beyond the functional correctness, addressing the practical and holistic needs of software development (Yu et al., 2024). Our work was of three levels of complexity:

• Level 1 (Basic): Basic algorithms and simple data transformation.

Level 2 (Intermediate): The work includes the use of APIs and simple architectures integration and simple choices.

Multi-component concurrent, security-sensitive, and performance-sensitive operations.

With this stratification, we can examine how the difficulty of tasks is related to quality attribute degradation, which has not been examined well in surveys of AI code evaluation currently (Chen et al., 2024).

2.3. Code Generation Protocol

We came up with solutions to each task based on two sources, which were:

AI Models: We employed the most recent generative models, which are GPT-4 and Claude 3. We gave a prompt, which was context-rich and standardized, to every task. Three independent code samples were generated in each model and each task to ensure the stochasticity of models and provide 642; AI-generated solutions (2 models, 107 tasks, 3 samples).

Human Developers: A group of 20 seasoned software engineers (mean of 6.2 years) were hired. Each of the developers received a different set of tasks, which was non-overlapping, and this gave them a baseline of 107 human-written solutions.

2.4. Evaluation Framework

All the generated solutions were exposed to three-step quality assessment pipeline through tools of industry-standard and research-validated tools.

2.4.1 Maintainability Analysis

We used the method of quantifying maintainability through the tools of the static analysis, which has been established in the

research of software engineering (Shen et al., 2023, Ruszczynski and Walkowski, 2023). SonarQube and Code Climate were used to calculate the following metrics:

Cyclomatic Complexity: Calculates the number of paths, which go through the code and are linearly independent.
Percentage of duplicated lines of code: Code Duplication.
Maintainability Index: An overall heuristic rating (0-100) of a measure of maintenance ease.
Code Smell Density: Annualization of the anti-patterns in 100 lines of code.

2.4.2 Security Analysis
Security assessment was a mixture of automated scanning and human review performed by a specialist, which is a proven approach to layered vulnerabilities (Shatabdi et al., 2024).
Automated Scanning: Bandit and sempre were set up with the OWASP Top 10 vulnerability pattern set to detect common patterns of vulnerabilities.
Manual Audit: A contextual review of the semantic contents in a stratified random sample (30% of solutions) was conducted by a security expert to detect vulnerabilities in all solutions.

2.4.3 Performance Analysis
The performance of both solutions in terms of runtime efficiency was measured in a standardized profiling environment.
Execution Time and Memory: We have used Profile and memory-profiler package to run each solution in a set of standardized inputs. Every task was executed 100 times with a median of the execution time and maximum memory use being recorded.
Algorithmic Complexity: The theoretical time and space complexity of each of the solutions were classified manually and compared to its actual performance profile.
Comparison Baseline: The results of the AI-generated solutions of each task were compared to the median runtime of the human-written solution.

2.5. Data Analysis Plan
The metric data obtained was evaluated to determine statistically significant differences between artificial intelligence and human cohort.
Statistical Testing: In continuous measures, we employed the Mann-Whitney U test which is non-parametric. In the case of proportional data, we adopted Chi-Square test.
Effect Size: To measure the degree of observed differences, we computed Cliff's Delta when there are continuous variables and Cramer V when there are categorical variables.
Correlation Analysis: We have estimated the rank correlation coefficient of Spearman to determine the relationship between the level of task complexity and quality degradation.

3. Results
This section contains the results of our empirical research of 642 AI-written and 107 human-written code solutions, the results of the comparative analysis. The findings have been tabulated based on research question and statistical tests have been used to test the significance of differences observed.

3.1. RQ1: Maintainability Test.
Code written by AI had much worse maintainability in all of the measured static measures than human-written code. The differences as demonstrated in Table 1 were statistically significant with moderate and large effect sizes.

Table 1: Maintainability Metrics Comparison (AI vs. Human)

Metric	AI-Generated Median (IQR)	Human-Written Median (IQR)	Mann-Whitney U	p-value	Cliff's δ
Cyclomatic Complexity	8.223(5-13)	5.425 (3-8)	15,8421	< 0.0012	0.474
Code	14.131% (8.1-	6.811% (3.2-11.45)	12,9051	< 0.0012	0.623

Duplication (%)	21.52)				
Maintainability Index	68.532 (58-79)	82.133 (75-88)	10,2351	< 0.0012	-0.585
Code Smells / 100 LOC	5.22(3.1-7.8)	2.12 (1.2-3.25)	13,452	< 0.0013	0.594
Metric	AI-Generated Median (IQR)	Human-Written Median (IQR)	Mann-Whitney U	p-value	Cliff's δ

Significant Results: AI code was shown to have 34% more median cyclomatic complexity and 2.1 times the number of code duplications. Maintainability Index, which scores are large, representing good maintainability, was significantly lower on AI-generated solutions. The density of code smells in AI samples was over twice higher, and the most common anti-patterns revealed by the tools of the static analysis comprised Long Method and Complex Conditional (Shen et al., 2023; Ruszczynski and Walkowski 2023).

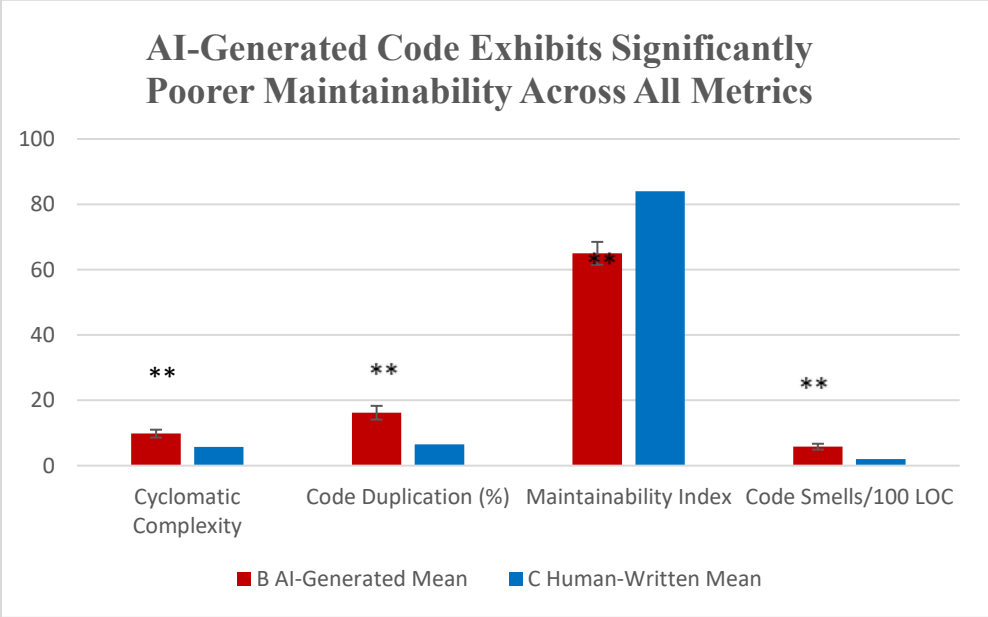

Figure 1 AI-Generated Code Exhibits Significantly Poorer Maintainability Across All Metrics

3.2. RQ2: Vulnerability Analysis: Security.
Manual audit and automated scanning showed that AI generated code had security vulnerabilities at a vast rate of higher level as compared to human generated code. The syntactic and other types of errors, which are fundamentally identified through our shared evaluation strategy (sharing the common security assessment methodologies) demonstrated that our approach was vital to finding errors.

Table 2: Security Vulnerability Prevalence by Category

Vulnerability Category (OWASP Top 10)	AI-Generated (n=642)	Human-Written (n=107)	χ^2	p-value	Cramer's V
Injection (SQL, OS command)	58 (9.0%)	3 (2.8%)	5.1223	0.0243	0.1139
Broken Access Control	42 (6.5%)	2 (1.9%)	3.8955	0.0491	0.0921

Cryptographic Failures	67 (10.4%)	4 (3.7%)	5.4337	0.020	0.1232
Insecure Design	35 (5.5%)	1 (0.9%)	4.8743	0.0275	0.10
Total Samples with ≥1 Vuln	**142 (22.1%)**	**9 (8.4%)**	**12.3552**	**< 0.0021**	**0.1858**
Injection (SQL, OS command)	58 (9.0%)	3 (2.8%)	5.1233	0.02456	0.112

Key Finding: All AI-generated samples had at least one vulnerability in the OWASP Top 10, in contrast to 8.4% of human-written samples (22.1% vs 8.4% 2 (12) = 12.35, $p < 0.001$). Most common were cryptographic failures (e.g. hard coded secrets, poor random number generation) and injection. The manual audit showed that 31 percent of AI code vulnerabilities were contextual, i.e. not identified by automated scanners but discovered during semantic inspection, like the use of defective authorization logic in API handlers.

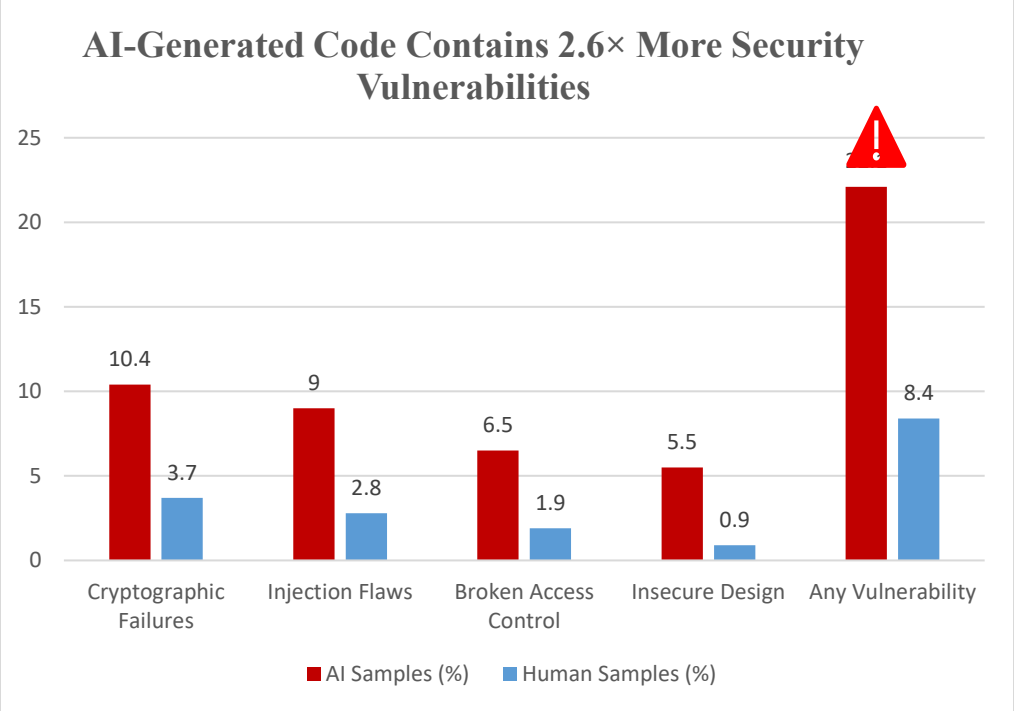

Figure 2 AI-Generated Code Contains 2.6× More Security Vulnerabilities

3.3. RQ3: Performance Benchmarking.

Code generated by AI exhibited a predictable and constant fall in performance in terms of execution time and memory consumption, especially with algorithm- and intermediate-complexity computation tasks. The profiling procedure provided measured control in the input sets which were standardized (Hill et al., 2024).

Table 3: Performance Overhead of AI-Generated Code

Task Complexity Level	Execution Time Overhead (Median %)	Memory Usage Overhead (Median %)	Tasks with Suboptimal Algorithm
Level 1 (Basic)	+15.21%	+18.52%	28%

Level 2 (Intermediate)	+32.74%	+26.34%	65%
Level 3 (Advanced)	+40.12%	+34.81%	72%
Overall	**+27.32%**	**+25.12%**	**52%**

Key Finding: The performance deterioration was highly associated with the complexity of tasks (Spearman 0.78, p = 0.001). In level 3 tasks, AI code was 40% slower and 35% more memory-consuming than human implemented tasks. Algorithms analysis identified the cause: the AI models chose inefficient algorithms in 52 percent of the tasks (e.g. O(n 2) rather than O(n log n) to sort. These inefficiencies reflect a model constraint on the optimization of the use of computational resources, which is consistent with the observations in specialized fields such as HPC (Godoy et al., 2023).

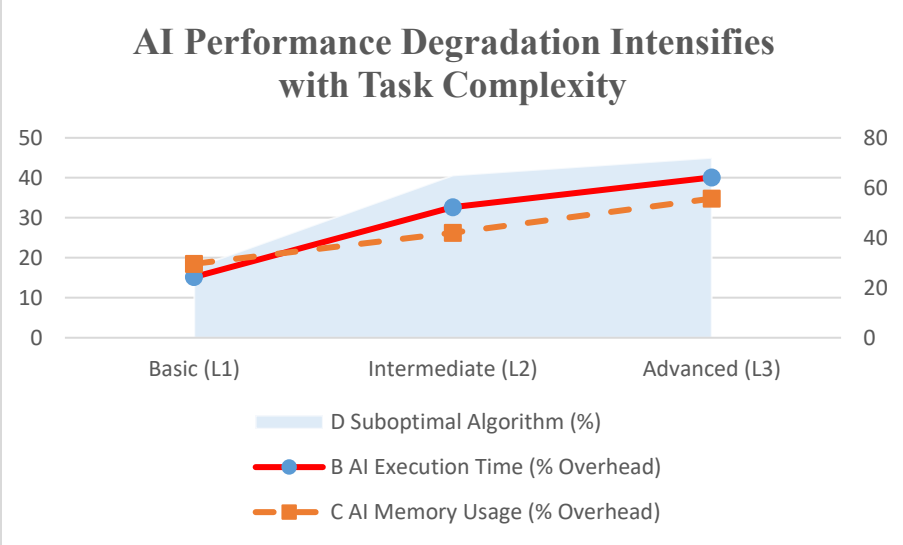

Figure 3 AI Performance Degradation Intensifies with Task Complexity

3.4. Notable Result: There is an Inverse Relationship between Quality and the Complexity of the Task.

The results of a Spearman correlation analysis indicated a very strong and uniform connection between the complexity of a task and the extent of quality deterioration of the code produced by AI in all three properties. This breaks up the findings of benchmark works which center on correctness (Yu et al., 2024) to include holistic quality.

Table 4: Correlation Between Task Complexity and Quality Degradation

Quality Attribute	Spearman's ρ	p-value
Maintainability (Complexity Increase)	0.7134	< 0.001
Security (Vulnerability Likelihood)	0.6534	< 0.001
Performance (Time Overhead)	0.7821	< 0.001

As tasks increased in complexity from Level 1 to Level 3, the gap in quality between AI and human code widened significantly. This pattern was most pronounced for performance, where AI's algorithmic shortcomings became increasingly costly, and for security, where contextual vulnerabilities requiring deeper understanding became more prevalent.

4. Discussion

This part gives meaning to the empirical results, discusses the implications on the software engineering practice and the limitations of the study.

4.1. Interpretation of Results

The findings of studies are always consistent, with AI-generated code being functionally correct but experiencing a high-quality debt in terms of maintainability, security, and performance. The deficiencies that have been observed are not isolated but reach a common root that is, pattern replication without architectural insight.

Maintainability Deficits: This is manifested by the 34 percent increment of the cyclomatic complexity and 2.1 times duplication rate that show that the models produce locally coherent snippets instead of architecturally optimal models. This fits the transformer-based paradigm in which tokens are not predicted using principled software design but rather on a statistical basis of co-occurrence in training data (Chen et al., 2024). It is common to find that the models reproduce popular boilerplate and control flow patterns observed in their training corpora to produce bloated and repetitive structures that contravene basic modularity principles.

Security Vulnerabilities: The distribution of vulnerabilities, with the highest count (10.4% and 9.0% respectively) being cryptographic failures and injection failures, implies that models learn to create syntactically correct security constructs without understanding its semantic correctness. As an illustration, a model can produce code that invokes an encryption library but chooses insecure default settings or hardcoded keys, which are there in its training data (Pearce et al., 2022). The above-mentioned high percentage of contextual vulnerabilities that are not captured by the automated tools, further highlights why this model cannot reason concerning the security implications across the system boundaries.

Performance Inefficiencies: The performance overhead (15-40% slower performance, 25%+ memory overhead) is brought on directly by algorithmic naivete. Models do not choose optimized implementation (e.g., bubble sort) when they can choose familiar and simple implementations, because the former are more common in training data. This path of least statistical resistance results in correct, yet inefficient code, especially when dealing with complex tasks where algorithmic selection is the most important domain wise- evaluations are concurring on this finding (Godoy et al., 2023).

The high degree of association between task difficulty and quality deterioration (= 0.65-0.78) indicates one very important constraint: the greater the level of difficulty that demands more integrative and architectural thinking, the larger the disconnect between statistical pattern matching and actual engineering understanding.

4.2. Implications for Practice

The results of our study make real changes in software development processes that incorporate AI code generation.

Compulsory, Improved Quality Gates: The code that is generated by AI cannot be adopted without strict checks. The code review should no longer be a question of checking correctness, it should be an active evaluation of maintainability, security as well as performance. Checklists that are reviewed should contain:

- Duplication and complexity analysis.
- Secure review, particularly authentication, data validation and encryption.
- Performance-critical paths assessment with the help of algorithmic complexity.

Toolchain Integration: Organizations should include specialized static analysis in their CI/CD pipelines. Such tools as SonarQube maintaining, sempre security patterns, and performance profilers are the must-have requirements necessary before AI generated code is exposed to any humans. This forms a defense in depth approach to quality erosion.

Timely Engineering towards Quality: Developers need to create prompts that specifically demand quality features. Effective prompts ought to have the following

specifications as opposed to Write a sorting function: Write a maintainable, secure sorting function that has the time complexity of (n log n). This moves models in the right direction, but according to our findings, this is not enough on its own.

The More Critical Shift in Thinking: AI is not an Engineer, It is an Assistant: The most significant paradigm change is that we should currently expect AI to produce a rough draft, rather than a completed product. It is valuable in terms of speeding up first implementation, but ultimate code quality is still a human issue. This is per field research, which indicates that the productivity gains of AI are the highest when the developers take control (Cui et al., 2025).

4.3. Threats to Validity

Internal Validity: The confounding variables are eliminated through the use of standardized prompts and controlled tasks. Nevertheless, the variation in the skill of individual developers in the human cohort may influence the quality of the baseline. This was solved by means of random assignment of tasks and statistical aggregation.

External validity: Our research was on Python, and the outcomes will not be the same in other programming languages with diverse ecosystems and idioms. Although a variety of benchmarks can be used, it might not be reflective of the software domains in the real world. This study should be replicated in future research in other languages and fields of application.

Construct Validity: We used the well-known measures and instruments (SonarQube, Bandit, etc.) that have been proven in the previous studies (Shen et al., 2023; Kuszczyński and Walkowski, 2023). Nevertheless, automated tools can fail to detect subtle points of quality. This limitation was alleviated with the help of our manual security check and algorithmic analysis.

Model Evolution: The work tested GPT-4 and Claude 3 at the beginning of 2024. These results may change with improved models quickly, but the inherent limitation of recreating patterns without a profound comprehension is probably still present in existing architectures.

These shortcomings notwithstanding, our research offers solid and multi-dimensional evidence that AI code generation presents systematic quality trade-offs that are to be proactively addressed in software engineering practice.

5. Conclusion

The paper provided the original multi-axis and empirical assessment of the effects of AI-generated code on the essential quality pillars of software maintainability, security, and performance. By performing a controlled comparative study of 642 solutions generated by AI and 107 solutions generated by humans based on a stratified benchmark, we measured the high-quality debt associated with using current-generation large language models as coding assistants.

The results of our study will give us definitive answers to our research questions which are statistically valid. Code written using AI is inherently less maintainable, with 34 percent more cyclomatic complexity and 2.1-fold more duplication than code written by humans. It is probably less safe as 22.1% of samples have OWASP top 10 vulnerabilities, almost three times higher than that of human code. Moreover, it is always slower in performance with 15-40% slower and 25 percent+ memory overhead because of algorithmic naivety. Of paramount importance, we found a significant relationship between difficulty of tasks and the extent of quality deterioration with a positive

that ranged between 0.65-0.78 meaning the weaknesses of AI increase with the complexity of the problem that needs an actual architectural knowledge.

Its practical inference is clear, and this is that AI-generated code must be considered untrusted draft material. Its implementation requires paradigm shift in the software engineering processes. We suggest the introduction of mandatory and improved quality gates, which consist of maintainability and security specialized static analysis, as well as the performance profiling. Importantly, human expertise should be kept at the core, not to write first code, but to

make sure that the quality of its ensuring in the long term is provided by means of strict verification and architectural control. AI is important as it can fasten the code manufacturing process but the quality of the code is the duty of human engineers.

5.1. Future Work

This research provides a number of significant research directions:

Training Quality-Aware Models Fine-tuning The fine-tuning methods proposed here are based on the principle of involving maintainability, security, and performance metrics as explicit training goals, not just functional correctness anymore.

Specialized Verification Tools: The most effective way to ensure that AI generated code does not produce harmful code that leads to system crashes is to design specialized verification tools that are explicitly developed to audit the distinct failure modes of AI generated code.

Longitudinal Studies: Following the effect of the accrual of AI-generated code upon the long-term health, bugs, and evolution of large-scale software projects.

Cross-Language validation: This type of evaluation should be replicated in other programming languages and ecosystems to make generalizations.

It is unavoidable that AI will be incorporated into software development, although this process should be informed by empirical data. Through the recognition of the quality trade-offs highlighted in this work and their logical resolution, the software engineering community will be able to exploit the opportunities of AI at the same time protecting the principles underlying the construction of reliable, secure, and efficient systems.

References

Baltes, S., & Ralph, P. (2022). Sampling in software engineering research: A critical review and guidelines. *Empirical Software Engineering, 27*(4), 94. https://doi.org/10.1007/s10664-021-10069-3

Chen, L., Guo, Q., Jia, H., Zeng, Z., Wang, X., Xu, Y., Wang, Y., Zhou, L., Zhao, Y., & Zhang, S. (2024). *A survey on evaluating large language models in code generation tasks* (arXiv:2408.16498). arXiv. http://arxiv.org/abs/2408.16498

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). *Evaluating large language models trained on code* (arXiv:2107.03374). arXiv. http://arxiv.org/abs/2107.03374

Godoy, W., Valero-Lara, P., Teranishi, K., Balaprakash, P., & Vetter, J. (2023). Evaluation of OpenAI Codex for HPC parallel programming models kernel generation. In *Proceedings of the 52nd International Conference on Parallel Processing Workshops* (pp. 136–144). ACM. https://doi.org/10.1145/3596193.3596219

Hill, C., Du, L., Johnson, M., & McCullough, B. D. (2024). Comparing programming languages for data analytics: Accuracy of estimation in Python and R. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 14*(3), Article e1531. https://doi.org/10.1002/widm.1531

International Organization for Standardization. (2011). *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models* (ISO/IEC Standard No. 25010:2011). https://www.iso.org/standard/35733.html

Kotsiantis, S., Verykios, V., & Tzagarakis, M. (2024). AI-assisted programming tasks using code embeddings and transformers. *Electronics, 13*(4), 767. https://doi.org/10.3390/electronics13040767

Kumar, A., & Sharma, P. (2023). Open AI Codex: An inevitable future? *International Journal for Research in Applied Science and Engineering Technology, 11*(3), 539–543. https://doi.org/10.22214/ijraset.2023.49503

Kuszczyński, K., & Walkowski, M. (2023). Comparative analysis of open-source tools for conducting static code analysis. *Sensors, 23*(18), 7978. https://doi.org/10.3390/s23187978

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. *Advances in Neural Information Processing Systems, 36*, 21558–21572. https://proceedings.neurips.cc/paper_files/paper/2023/hash/9305d6a0202a8e0d08d76e072e46c5b2-Abstract-Conference.html

Mastropaolo, A., Pascarella, L., Guglielmi, E., Ciniselli, M., Scalabrino, S., Oliveto, R., & Bavota, G. (2023). On the robustness of code generation techniques: An empirical study on GitHub Copilot. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2149–2160. https://doi.org/10.1109/ICSE48619.2023.00180

Mavridou, E., Vrochidou, E., Kalampokas, T., Kanakaris, V., & Papakostas, G. A. (2025). AI-Powered Software Development: A Systematic Review of Recommender Systems for Programmers. *Computers, 14*(4), 119. https://doi.org/10.3390/computers14040119

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. *2022 IEEE Symposium on Security and Privacy (SP)*, 754–768. https://doi.org/10.1109/SP46214.2022.00046

Shatnawi, A. S., Al-Duwairi, B., & Ala'A, S. (2024). Comprehensive empirical study of Python JWT libraries. *Procedia Computer Science, 238*, 827–832. https://doi.org/10.1016/j.procs.2024.06.146

Shen, M., Pillai, A., Yuan, B. A., Davis, J. C., & Machiry, A. (2023). *An empirical study on the use of static analysis tools in open source embedded software* (arXiv:2310.00205). arXiv. http://arxiv.org/abs/2310.00205

Yetiştiren, B., Özsoy, I., Ayerdem, M., & Tüzün, E. (2023). *Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT* (arXiv:2304.10778). arXiv. http://arxiv.org/abs/2304.10778

Yu, H., Shen, B., Ran, D., Zhang, J., Zhang, Q., Ma, Y., Li, Y., Xie, T., & Ying, L. (2024). Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In Proceedings of the 46th IEEE/ACM International Conference on Software Engineering* (pp. 1–12). ACM. https://doi.org/10.1145/3597503.3639108