International Journal of Latest Research in Engineering and Management (IJLREM) *ISSN: 2456-0766*
www.ijlrem.org || Volume 10 - Issue 03 || May-June 2026 ǁ PP. 01-05

Programming Paradigms in the Age of AI: Prompt-Engineered Code vs. Hand-Written Code

Adewole A.P., Awoleye Bolaji Stephen, Augustine Peter Edoka,
Omikunle Oluwafisayo Yewande, Abe Oluwaseyi Elizabeth
Department of Computer Science,
University of Lagos, Nigeria

Abstract: This study compared prompt-engineered and hand-written code across 15 programming tasks in Python, Java, and JavaScript. An experienced developer and a prompt engineer using GPT-4o and Claude 3.5 Sonnet completed identical tasks, evaluated on development speed, functional accuracy, cyclomatic complexity, security, and maintainability. Although the development time with AI-based assistance decreased by around 82%, the code generated by AI showed a 14.7% decreased maintainability, a 61.9% increased cyclomatic complexity, and had no security risks compared with the manually written code. In conclusion, a Pareto-type of problem was revealed. The AI solves 80% of any task rapidly, while the 20% edge cases and architecture/security logic needs a real human being. A hybrid model is proposed wherein developers retain authorship of core architecture and business logic while delegating well-defined implementation patterns to AI tools**.**

Keywords: prompt engineering, AI-assisted programming, large language models, code maintainability, software security

Introduction

For more than fifty years, writing software meant a human being translated thinking into a formal language a machine could execute. That assumption is now under serious pressure. The same decade that gave birth to GPT-4 and Claude has spawned a generation of developers who specify desired behavior in natural language and let a model generate the code. Platforms like GitHub Copilot are integrated into the daily work of professional engineers, and the change raises concerns the industry has not yet totally resolved. Every big paradigm change in software history has transferred a component of developer cognitive strain to the toolchain. Procedural programming brought subroutines; object-oriented programming added domain modeling; functional and declarative programming enabled programmers to define results instead of methods. Prompt engineering is the most declarative interface yet invented—natural language as specification—but it also differs qualitatively: the tool now has sufficient world knowledge to fill substantial gaps on its own, creating capabilities and risks without clean historical precedent (Wegner, 1987). The enthusiasm surrounding AI coding tools has outpaced the evidence. Productivity gains are real, but the literature on what those gains cost in code quality, maintainability, and security remains thin. This study makes the trade-off clear by means of a controlled comparison throughout 15 tasks, assessing development time, functional correctness, cyclomatic complexity, security vulnerabilities, and maintainability index. The research questions that guided us were: (a) What is the difference in development time between paradigms, and does the difference get smaller as the projects get more complex? (b) Do the two strategies differ significantly in terms of conventional quality criteria? (c) Delegating logic to an LLM creates what security and reliability concerns? (d) How is the skill set of a working software engineer evolving?

Literature Review

Conventional Paradigms and AI-Assisted Development

Each programming paradigm (imperative, OO, functional, declarative) encapsulates an underlying theory to tackle software complexity. As Wegner (1987) noted, paradigms are theories about programming rather than merely syntax rules. The fact that unconstrained control flow leads to unchangeable code was established by Dijkstra (1968); the discipline provided by structured programming leads to software reliability. Booch (1994) introduced an idea of the enhanced modeling capability that the object-oriented programming approach provides, and Backus (1978), Abiteboul et al. (1995) provided an explanation on how the functional and declarative languages work by treating programming as specification of 'what' rather than 'how'. Transformer design (Vaswani et al., 2017) made the next leap by using natural language as the interface, and the implementation itself was then determined from the big training data.

Quality, Security, and Productivity

Chen et al. (2021) presented Codex, which achieved a 28.8% accuracy on HumanEval, and although not that accurate, it represented an important step in generating novel programming challenges. In realistic workflows, Ziegler et al. (2022) report that approximately 30% of users adopt GitHub Copilot and that they find that it shortens the time to

the first commit. Using a randomized controlled experiment (N=95), Peng et al. (2023) report a speed advantage of 55.8% which is greater for boilerplate tasks.

Quality costs are equally documented. Siddiq et al. (2022) found AI-generated code shows higher frequencies of structural anti-patterns, coining the term 'AI-induced technical debt.' Fowler (2018) noted that code is read far more often than written, meaning write-time savings are partially offset by read-time costs. The most alarming findings concern security: Pearce et al. (2022) found approximately 40% of Copilot-generated code in security-sensitive contexts contained exploitable vulnerabilities, including SQL injection and insecure defaults, because LLMs cannot distinguish between widely-used patterns and widely-used insecure patterns.

Table 1 synthesizes the major prior studies.

Table 1: Comparative Summary of Key Prior Studies

Author(s)	Year	Method	Key Finding	Limitation
Chen et al.	2021	Benchmarking	Codex: 28.8% HumanEval accuracy	Isolated functions only
Pearce et al.	2022	Security Analysis	~40% of AI suggestions insecure	Known CWE patterns only
Ziegler et al.	2022	User Study	AI reduces time-to-first-commit	Self-report bias
Siddiq et al.	2022	Static Analysis	AI code shows higher God Method frequency	Python/Java only
Peng et al.	2023	RCT (N=95)	55.8% faster completion of the task	Neglects long-term maintenance
Vaithilingam et al.	2022	Qualitative Study	Developers have difficulty with debugging the code used by AI	Small sample (N=24)

The dominant gap in this literature is longitudinal: almost all studies evaluate code at the moment of creation. This study extends the comparison to maintenance-phase behavior.

Methodology

We adopted a Comparative Experimental design where the same tasks could be tested across two settings while controlling other variables. The control condition consisted of a senior software developer coding manually and the experimental conditions consisted of a proficient prompt engineer prompting GPT-4o (primary) and Claude 3.5 Sonnet (secondary for cross-validation). All code was run on the same virtual machine (8-core CPU, 32GB RAM, NVIDIA T4 GPU; Ubuntu 22.04 LTS; VS Code). Git was used for session timestamping. All prompting used a Chain-of-Thought strategy—models first articulated logic in pseudocode before generating implementation code.

Fifteen tasks were selected to span industrial programming work from simple algorithms to complex multi-dependency integrations across Python, Java, and JavaScript. Table 2 shows a representative sample. Evaluation metrics appear in Table 3.

Table 2: Experimental Tasks (Representative Sample)

Task ID	Description	Difficulty	Language
T-01	Thread-Safe Singleton Pattern	Medium	Java
T-02	Recursive Fibonacci with Memoization	Easy	Python
T-03	RESTful API with JWT Authentication	Hard	JavaScript
T-04	Depth-First Search on Unweighted Graph	Medium	Python
T-05	SQL-to-NoSQL Database Migration Script	Hard	Python
T-06	JSON Parser for Deeply Nested Objects	Medium	JavaScript

Table 3: Evaluation Metrics

Metric	Description	Tool
Development Time	Task brief to all unit tests passing	Stopwatch (min)
Functional Accuracy	% of predefined unit tests passed on first run	% pass rate

Cyclomatic Complexity	Linearly independent code paths (McCabe scale)	Radon
Security Vulnerabilities	SQLi, XSS, improper error handling	Snyk / SonarQube
Maintainability Index	Composite: Halstead Volume + LOC	MI Score (0-100)

For hand-written tasks, developers could consult documentation but not AI tools. For prompt-engineered tasks, all corrections had to be achieved through re-prompting rather than direct code editing, maintaining a clean separation between conditions.

Results

Table 4 presents performance data. AI-assisted development was dramatically faster at the low end—T-02 dropped from 8 minutes to 90 seconds. At the high end, T-05 still favored AI (45 vs. 190 minutes), but at 70% accuracy and greater structural complexity.

Table 4: Performance Comparison: Hand-Written vs. AI-Assisted

Task	**HW Time (min)**	**AI Time (min)**	**Accuracy (%)**	**LOC: HW / AI**
T-01	12	2	100	45 / 38
T-02	8	1.5	100	15 / 12
T-03	145	22	85	450 / 512
T-04	35	4	95	65 / 72
T-05	190	45	70	820 / 940
T-06	55	8	90	110 / 135

Note. HW = hand-written; LOC = lines of code.

Code Quality and Security

Table 5: Shows maintainability metrics. With the 14.7% reduction in the maintainability index, and a 61.9% increase in the cyclomatic complexity, we see that AI-generated code is significantly harder to comprehend and is less secure to modify. The percentage of comments was reduced by 34.1% even though we assumed that AI-generated code is likely to produce well-documented code.

Table 5: Maintainability Metrics by Condition

Metric	**Hand-Written**	**AI-Generated**	**Delta**
Maintainability Index	84.5	72.1	-14.7%
Cyclomatic Complexity	4.2	6.8	+61.9%
Halstead Volume	1,250	1,580	+26.4%
Comment Density	18.5%	12.2%	-34.1%

Security vulnerabilities were absent in hand-written code for most tasks but appeared frequently in AI output (Table 6). In several JavaScript tasks, the AI suggested deprecated API methods, causing silent production failures—a structural consequence of the training data cutoff.

Table 6: Error Frequency and Root Causes by Condition

Error Category	**Hand-Written**	**AI-Generated**	**Root Cause (AI)**
Logic Flaws	Low	High	Model hallucination
Syntax Errors	Very Low	Medium	Context window cutoff
Security Holes	Minimal	High	Insecure training data
Library Conflicts	Moderate	High	Outdated API knowledge

Three trends cross-task were observed: (a) a Pareto problem- the first prompt addressed ~80% of a task but used ~80% of total prompting time for the other 20% of the task (b) verbosity bias- the AI code was, on average, 15% more

verbose than the handwritten equivalent; and (c) dependency drift- AI suggestions always lagged one (or more) behind the most recent release of the library.

The results are not simply interpretable as 'AI is fast but flawed.' For well-defined, pattern-rich tasks, AI tools performed well across every dimension, while indications suggest that performance significantly deteriorated for jobs requiring contextual judgment, architectural awareness, or security knowledge. This difference has real consequences: widespread adoption of artificial intelligence poses a general risk; focused application—AI for boilerplate, test creation, and documentation—captures most of the productivity advantage and directs human review where it is most needed.

When a human writes code, they form a mental map of intended logic. In prompt engineering, developers often assume AI output is correct if it passes basic tests, creating a 'trust but don't verify' culture that is dangerous in production. LLMs trained on publicly available repositories also inherit the insecure practices in the repositories themselves (Pearce et al., 2022). Moreover, debugging code not written by you has a tangible cognitive cost: programmers discarded much of the generated code altogether and rewrote it themselves, according to Vaithilingam et al. (2022), thus consuming much of the time initially saved.

Software engineers are moving toward the roles of Code Auditor and Systems Architect. The most important skills are now (a) requirement specification-detailing complex logic precisely in natural language; (b) verification and validation-skilled use of static analysis and automated testing tuned to the specific failure modes of LLMs; and (c) architectural overview-integrating AI-generated components into a large, scalable whole. Contrary to fears of replacing programmers, creativity in software is changing from skillful implementation toward design judgment (Brooks, 1987; Meyer, 2023).

Conclusion

Our empirical findings show that while AI-assisted development brings radical improvements in speed (averaging 82% decrease in development time), there is a trade-off to system integrity: 61.9% increase in cyclomatic complexity, 14.7% decrease in maintainability, and elevated vulnerability rates. This results in three main conclusions: 1) prompt engineering and human handwriting are complementary, suited for distinct problem domains. 2) Performance gains are greatest on defined pattern-based problems and minimal on truly novel, structurally complex, or security-critical problems. 3) Long-term AI-assisted development implies a transition away from programming and towards specification, verification, and design judgment.

Perhaps the biggest unanswered question is how the AI-generated code will stand up over multiple maintenance cycles of adding features and fixing bugs. Additional priorities include Explainable AI for Code (models providing rationale behind logic choices), long-context repository evaluation as million-token models become available, and human factors research on the psychological effects of sustained AI-assisted development

References

[1]. Abiteboul, S., Hull, R., &Vianu, V. (1995). Foundations of databases. Addison-Wesley.
[2]. Anthropic. (2024). Model card: Claude 3 [Technical report]. https://www.anthropic.com
[3]. Backus, J. (1978). Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM, 21(8), 613-641. https://doi.org/10.1145/359576.359579
[4]. Booch, G. (1994). Object-oriented analysis and design with applications (2nd ed.). Benjamin/Cummings.
[5]. Brooks, F. P. (1987). No silver bullet: Essence and accidents of software engineering. Computer, 20(4), 10-19. https://doi.org/10.1109/MC.1987.1663532
[6]. Chen, M., et al. (2021). Evaluating large language models trained on code. arXiv. https://arxiv.org/abs/2107.03374
[7]. Dijkstra, E. W. (1968). Letters to the editor: Go to statement considered harmful. Communications of the ACM, 11(3), 147-148. https://doi.org/10.1145/362929.362947
[8]. Fowler, M. (2018). Refactoring: Improving the design of existing code (2nd ed.). Addison-Wesley Professional.
[9]. Kabir, M. G., Williams-King, D., & Vahdati, S. (2023). On the reliability of ChatGPT for coding tasks. Journal of Systems and Software, 195. https://doi.org/10.1016/j.jss.2022.111533
[10]. Liu, P., et al. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys, 55(9). https://doi.org/10.1145/3560815
[11]. McCabe, T. J. (1976). A complexity measure. IEEE Transactions on Software Engineering, SE-2(4), 308-320. https://doi.org/10.1109/TSE.1976.233837
[12]. Meyer, B. (2023). AI and the future of software engineering: A skeptical view. IEEE Software, 40(1), 92-96. https://doi.org/10.1109/MS.2022.3212130
[13]. OpenAI. (2023). GPT-4 technical report. arXiv. https://arxiv.org/abs/2303.08774
[14]. Pearce, H., Ahmad, B., Tan, B., Krishnamurthy, P., & Khorrami, F. (2022). Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions. In Proceedings of the IEEE Symposium on Security and Privacy (S&P). IEEE.
[15]. Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of AI on developer productivity: Evidence from GitHub Copilot (NBER Working Paper No. w30923). National Bureau of Economic Research.

https://doi.org/10.3386/w30923

[16]. Reynolds, L., & McDonell, K. (2021). Prompt programming for large language models: Beyond the few-shot paradigm. In Proceedings of CHI Conference on Human Factors in Computing Systems. ACM.

[17]. Siddiq, S., Santos, J. C., Santos, G. L., &Khomh, F. (2022). Empirical study of code smells in transformer-based code generation techniques. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE.

[18]. Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. experience: Evaluating the usability of code generation tools based on large language models. In CHI Conference on Human Factors in Computing Systems. ACM.

[19]. Vaswani, A., et al. (2017). Attention is all you need. In Advances in Neural Information Processing Systems (NeurIPS). Curran Associates.

[20]. Wadler, P. (1998). The expression problem. Java-genericity mailing list.

[21]. Wegner, P. (1987). Dimensions of object-based language design. ACM SIGPLAN Notices, 22(12), 168-182. https://doi.org/10.1145/38807.38823

[22]. Ziegler, A., et al. (2022). Productivity assessment of neural code completion. In Proceedings of the 6th ACM SIGPLAN International Workshop on Machine Learning and Software Engineering. ACM.