The Impact of Large Language Models on Modern Software Development: An Evidence-Based Analysis of Code Generation, Productivity, and Security

Gregorio Sebastian Gualavisi Gonzalez, Andres Ricardo Guanolisa Plasencia, and Christian Humberto Pizanán Cárdenas
Pontificia Universidad Católica del Ecuador, Sede Esmeraldas
Esmeraldas, Ecuador
ORCID: 0009-0005-0351-2831, 0000-0001-5906-4832, 0009-0004-2608-270X
Email: {gsgualavasi, arguanoluisa, chpizanan}@pucese.edu.ec

***Abstract*—Large Language Models (LLMs) have emerged as transformative tools in modern software development, fundamentally changing how developers design, implement, test, and maintain software systems. Advanced AI systems such as code-generation assistants and intelligent development agents are increasingly integrated into software engineering workflows, enabling significant improvements in productivity, code quality, and development efficiency. This article examines the impact of LLMs on contemporary software engineering practices, focusing on automated code generation, debugging support, software documentation, code review processes, and collaborative development environments. To ground the discussion empirically, we conduct a structured synthesis of published experimental evidence, aggregating functional-correctness results on the HumanEval benchmark across eight representative models, controlled productivity experiments with AI pair programmers, and security-focused user studies. The synthesized evidence shows that state-of-the-art models solve up to 67% of HumanEval problems on the first attempt, that developers assisted by LLM-based tools completed a standardized task approximately 55.8% faster in a controlled experiment, and that roughly 40% of code suggestions produced in security-sensitive scenarios contained exploitable weaknesses. The study analyzes both the opportunities and challenges associated with adopting LLM-based tools, including concerns related to code reliability, security vulnerabilities, intellectual property, and developer overreliance. The findings suggest that while LLMs are unlikely to replace software engineers, they are becoming indispensable tools that augment human capabilities and reshape the future of software development.**

***Index Terms*—Large Language Models, Artificial Intelligence, Software Engineering, Code Generation, AI Assistants, Software Development, Human–AI Collaboration**

I. Introduction

SOFTWARE development has historically evolved through successive waves of abstraction and automation: high-level languages, integrated development environments (IDEs), version control, and continuous integration. The most recent wave is driven by Large Language Models (LLMs)—neural networks based on the Transformer architecture [1] and trained on massive corpora of natural language and source code [2], [3]. Unlike earlier code-completion systems that relied on static analysis or n-gram statistics [4], [5], LLM-based assistants can synthesize complete functions from natural-language descriptions, explain unfamiliar code, propose bug fixes, and generate tests and documentation.

The rapid industrial adoption of tools such as GitHub Copilot, ChatGPT, and successor coding agents has raised fundamental questions for the software engineering community. Do these tools produce functionally correct code? Do they measurably improve developer productivity? What risks do they introduce with respect to security, intellectual property, and skill erosion? Recent systematic reviews identify hundreds of primary studies addressing these questions [6], [7], yet the evidence remains fragmented across machine learning, software engineering, security, and human–computer interaction venues.

This paper makes three contributions:

1) A structured overview of how LLMs are being applied across the software development lifecycle (SDLC), covering code generation, debugging and program repair, documentation, code review, and collaborative development (Section III).
2) An evidence synthesis that aggregates published experimental results along three research questions—functional correctness, developer productivity, and security—using a transparent and replicable protocol (Sections IV and V).
3) A critical discussion of risks, limitations, and open problems, together with a roadmap for effective human–AI collaboration in software engineering (Sections VI and VIII).

II. Background and Related Work

A. From Language Models to Code Models

The Transformer architecture [1] enabled the scaling of language models to hundreds of billions of parameters, with GPT-3 demonstrating strong few-shot generalization across tasks

[2]. Chen et al. introduced Codex, a GPT model fine-tuned on public source code, together with the HumanEval benchmark of 164 hand-written programming problems evaluated by unit tests [3]. Subsequent open and proprietary models—including CodeGen [8], StarCoder [9], PaLM [10], Code Llama [11], and GPT-4 [12]—progressively raised the state of the art in program synthesis, while complementary benchmarks such as MBPP broadened evaluation coverage [13].

B. Empirical Software Engineering Research on LLMs

A growing body of empirical work evaluates LLM-based tools with the methods of evidence-based software engineering. Controlled experiments measure task completion time with and without AI assistance [14]; telemetry studies correlate suggestion acceptance rates with perceived productivity [15]; usability studies analyze how programmers interact with generated code [16], [17]; and security studies quantify vulnerabilities in generated code [18]–[20]. Systematic literature reviews consolidate this landscape and identify open challenges [6], [7]. Our work differs from prior surveys in that it organizes the evidence explicitly around three practitioner-facing research questions and reports the underlying quantitative results side by side to support decision-making.

III. LLMs Across the Software Development Lifecycle

A. Automated Code Generation

Code generation is the most mature application. Given a docstring, signature, or natural-language prompt, models synthesize candidate implementations that are validated against unit tests. The standard metric is *pass@k*: the probability that at least one of k sampled solutions passes all tests [3]. Beyond benchmarks, in-IDE assistants generate single lines, blocks, or whole functions inline, and observational studies show that programmers alternate between an "acceleration mode," where suggestions speed up code they already intended to write, and an "exploration mode," where suggestions are used to discover unfamiliar APIs [17].

B. Debugging and Automated Program Repair

LLMs have substantially advanced automated program repair (APR). Xia et al. showed that directly applying large pre-trained models to repair tasks outperforms many specialized APR techniques on standard defect benchmarks, by framing repair as conditional code generation over buggy context [21]. In practice, developers also use conversational models to explain stack traces, localize faults, and propose patches, shortening the diagnose–fix loop.

C. Software Documentation

Because LLMs are trained jointly on code and natural language, they translate fluently between the two. Common applications include generating docstrings and README files, summarizing legacy modules, and producing migration notes. Pre-trained bimodal models such as CodeBERT demonstrated early that code–text alignment supports documentation and search tasks [5], and systematic reviews report documentation generation among the most frequently studied downstream tasks [6].

D. Code Review and Quality Assurance

LLMs are increasingly used as a "first-pass reviewer" that flags style violations, suspicious logic, and missing edge cases before human review. They also generate unit tests, increasing coverage at low cost; however, empirical evaluation shows that generated tests and code require careful human validation, since plausible-looking output may encode subtle defects [22].

E. Collaborative Development Environments

The interaction paradigm is shifting from autocomplete to conversation and, more recently, to agentic workflows in which a model plans multi-step changes, edits multiple files, runs tests, and iterates. Prompt engineering has emerged as a complementary skill, with reusable prompt patterns documented for software tasks [23]. Usability research nevertheless shows a gap between expectation and experience: developers value assistants as starting points but report difficulty understanding and debugging generated code [16].

IV. Empirical Methodology

To move beyond anecdote, we performed a structured synthesis of published experimental evidence. We deliberately rely on primary studies with public artifacts so that every figure reported in Section V can be traced to its source and independently verified.

A. Research Questions

- RQ1 (Correctness): How functionally correct is LLM-generated code, and how has correctness evolved across model generations?
- RQ2 (Productivity): What is the measured effect of LLM assistants on developer productivity?
- RQ3 (Security): What is the prevalence of security weaknesses in LLM-generated code, and how does assistance affect the security of developer-written code?

B. Study Selection and Extraction Protocol

We selected primary studies according to four criteria: (i) peer-reviewed venue or widely replicated technical report; (ii) quantitative results on a public benchmark or a controlled/observational human study; (iii) explicit methodology (sample sizes, metrics, evaluation harness); and (iv) relevance to one of the three RQs. For RQ1 we extracted *pass@1* scores on HumanEval [3] as reported in the original model papers [3], [8]–[12], restricting extraction to zero-shot, single-sample settings to keep results comparable. For RQ2 we extracted effect sizes and sample sizes from the controlled experiment of Peng et al. [14] and the telemetry study of Ziegler et al. [15]. For RQ3 we extracted vulnerability rates and study designs from Pearce et al. [18], Sandoval et al. [19], and Perry et al. [20].

TABLE I
ZERO-SHOT FUNCTIONAL CORRECTNESS ON HUMANEVAL (PASS@1), AS REPORTED BY THE ORIGINAL MODEL PAPERS

Model	Year	Params	pass@1 (%)
GPT-3 [3]	2020	175B	0.0
Codex [3]	2021	12B	28.8
CodeGen-Mono [8]	2022	16B	29.3
PaLM-Coder [10]	2022	540B	36.0
StarCoder [9]	2023	15.5B	33.6
Code Llama [11]	2023	34B	48.8
GPT-3.5 [12]	2022	n/a	48.1
GPT-4 [12]	2023	n/a	67.0

C. Replication Protocol

For readers who wish to reproduce RQ1 locally, the procedure is: (1) obtain the 164 HumanEval problems and the official execution harness [3]; (2) query each model with the unmodified function signature and docstring at temperature t=0 (greedy decoding) for pass@1; (3) execute the generated body against the hidden unit tests in a sandbox; and (4) report the fraction of problems whose tests all pass. Identical prompts, decoding settings, and harness versions are essential, as small protocol deviations are a known source of variance across reports.

V. RESULTS AND ANALYSIS

A. RQ1: Functional Correctness

Table I and Fig. 1 summarize zero-shot pass@1 results on HumanEval as reported by the original model papers. Three observations stand out. First, the jump from general-purpose GPT-3 (which solves essentially none of the problems [3]) to the code-specialized Codex (28.8%) demonstrates the decisive effect of training on source code. Second, open models such as StarCoder (33.6%) [9] and Code Llama 34B (48.8%) [11] have progressively narrowed the gap with proprietary systems. Third, GPT-4 reaches 67.0% pass@1 [12], meaning that roughly one in three benchmark problems still fails on the first attempt—a reminder that generated code cannot be merged without verification.

B. RQ2: Developer Productivity

Table II summarizes the two principal quantitative studies. In the controlled experiment of Peng et al., 95 professional developers were randomly assigned to implement an HTTP server in JavaScript with or without GitHub Copilot; the treated group completed the task 55.8% faster, a large and statistically significant effect [14]. Complementarily, Ziegler et al. analyzed telemetry and surveys from more than 2,000 Copilot users and found that the rate of accepted suggestions is the strongest single predictor of self-reported productivity, with users accepting roughly 27–30% of shown completions during the study period [15]. Qualitative studies temper these results: Vaithilingam et al. observed that although most participants preferred working with the assistant because it provided useful starting points, assistance did not necessarily reduce completion time, since participants spent additional effort understanding and debugging generated code [16].

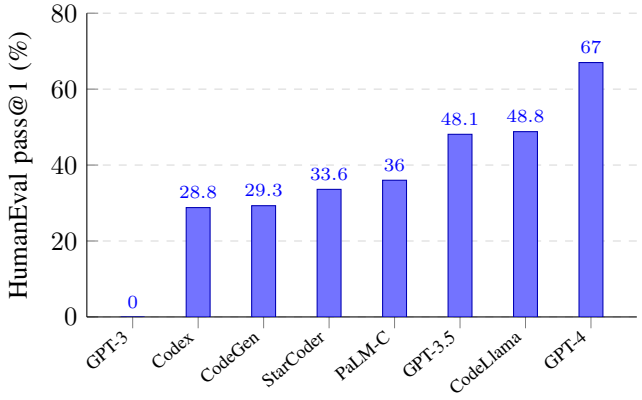

Fig. 1. Evolution of zero-shot functional correctness (pass@1) on the HumanEval benchmark across model generations. Values are taken verbatim from the original publications [3], [8]–[12].

TABLE II
QUANTITATIVE EVIDENCE ON DEVELOPER PRODUCTIVITY WITH LLM ASSISTANTS

Study	Design	Headline Result
Peng et al. [14]	RCT, n=95 devs, standardized task	Treated group 55.8% faster than control
Ziegler et al. [15]	Telemetry + survey, n>2000 users	Acceptance rate (≈27–30%) best predictor of perceived productivity
Vaithilingam et al. [16]	Within-subjects usability study, n=24	Preference for assistant; no significant time reduction; comprehension overhead

C. RQ3: Security of Generated and Assisted Code

Security results are summarized in Table III. Pearce et al. prompted Copilot in 89 scenarios aligned with MITRE's CWE Top-25 and analyzed 1,689 generated programs, finding that approximately 40% were vulnerable [18]. Two user studies refine this picture. Sandoval et al. found that, for low-level C programming with pointer and memory operations, assisted participants introduced security-relevant bugs at rates no more than 10% higher than the control group, suggesting a bounded marginal risk in that setting [19]. In contrast, Perry et al. found that participants with access to an AI assistant produced significantly less secure solutions across several security-sensitive tasks (encryption, signing, SQL) and—critically—were more confident that their insecure answers were correct [20]. The combination of plausible output and miscalibrated confidence constitutes the central security risk of LLM-assisted development.

D. Cross-Cutting Analysis

Read together, the three strands of evidence yield a coherent picture. Correctness has improved rapidly but remains far from guaranteed (RQ1); productivity gains are real and large

TABLE III
EMPIRICAL EVIDENCE ON SECURITY OF LLM-GENERATED AND LLM-ASSISTED CODE

Study	Design	Headline Result
Pearce et al. [18]	89 CWE scenarios; 1,689 programs	≈40% of generated programs vulnerable
Sandoval et al. [19]	User study, low-level C tasks	Assisted bug rate $\leq$10% above control
Perry et al. [20]	User study, security tasks	Assisted users less secure and more (over)confident

for well-scoped implementation tasks, but partially offset by comprehension and verification overhead (RQ2); and security outcomes depend strongly on task type, developer expertise, and whether output is independently validated (RQ3). The practical implication is that LLMs deliver the greatest net benefit when embedded in workflows with strong automated verification—unit tests, static analysis, security scanning, and mandatory human review—which convert probabilistic generation into engineering-grade artifacts.

VI. DISCUSSION: OPPORTUNITIES AND CHALLENGES

A. Code Reliability

Benchmark performance overstates real-world reliability: HumanEval problems are short, self-contained, and unambiguous, whereas industrial tasks involve large contexts, implicit requirements, and cross-file dependencies. Studies comparing Copilot against human solutions report that generated code is a useful draft but frequently suboptimal or subtly incorrect, requiring expert oversight [22]. Treating generation as hypothesis and testing as falsification is the appropriate engineering stance.

B. Security Vulnerabilities

The evidence in Table III implies that organizations adopting LLM assistants should pair them with security linters, software-composition analysis, and threat-model-aware review, particularly for authentication, cryptography, and input-handling code paths, where empirical vulnerability rates are highest [18], [20].

C. Intellectual Property and Licensing

Models trained on public repositories can occasionally reproduce memorized training fragments, raising questions about license compatibility and attribution that remain legally unsettled. Mitigations include provenance filters that block suggestions matching public code verbatim, organizational policies on permissible use, and preference for models with documented, permissively licensed training corpora such as StarCoder [9].

D. Developer Overreliance and Skill Formation

The overconfidence effect documented by Perry et al. [20] is most dangerous for novices, who lack the schemas needed to evaluate plausible output. Interaction studies suggest concrete countermeasures: encouraging deliberate switching between acceleration and exploration modes [17], requiring developers to write tests before accepting generated implementations, and teaching prompt construction and output auditing as explicit curriculum components [23].

VII. THREATS TO VALIDITY

Construct validity: pass@1 measures functional correctness on unit tests, not maintainability, performance, or security. Internal validity: the synthesized studies differ in model versions, prompts, and populations; we mitigated this by restricting RQ1 extraction to a single benchmark and decoding regime and by reporting study designs alongside results. External validity: most controlled experiments use short, well-specified tasks and may not generalize to large-scale maintenance work; benchmark contamination of training data is an additional concern acknowledged in the primary studies [11], [12]. Conclusion validity: given the heterogeneity of designs, we report results narratively rather than computing pooled meta-analytic estimates.

VIII. FUTURE DIRECTIONS

Four trends will shape the next phase of AI-assisted software engineering. First, agentic development: models that plan, edit, execute, and self-verify across whole repositories, shifting the human role toward specification and review. Second, repository-scale context: retrieval-augmented and long-context techniques that ground generation in project-specific conventions, reducing the correctness gap identified in RQ1. Third, verification-centric pipelines: tight coupling of generation with test synthesis, static analysis, and formal methods, addressing the reliability and security risks of RQ3. Fourth, evidence-based adoption: organizations will increasingly demand controlled measurements—of the kind synthesized here—before and after deployment, and the research community must respond with benchmarks that better reflect maintenance, evolution, and team-level outcomes [6], [7].

IX. CONCLUSION

This paper examined the impact of Large Language Models on modern software development through a structured synthesis of published experimental evidence. The results show steep and continuing gains in functional correctness, large measured productivity benefits on well-scoped tasks, and material—but manageable—security risks, dominated by the interaction between plausible output and developer overconfidence. LLMs are unlikely to replace software engineers; rather, they redistribute engineering effort from typing toward specifying, verifying, and integrating. Teams that institutionalize verification and treat the model as a capable but fallible collaborator capture the benefits while containing the risks. Human–AI collaboration, supported by rigorous empirical evaluation, is

thus emerging as the defining methodology of the next decade of software engineering.

REFERENCES

[1] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017, pp. 5998–6008.

[2] T. Brown *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[3] M. Chen *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[4] A. Svyatkovskiy, S. K. Lahiri, E. Alipour, and N. Sundaresan, "IntelliCode compose: code generation using transformer," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1433–1443.

[5] Z. Feng *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, 2020, pp. 1536–1547.

[6] X. Hou *et al.*, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–41, 2024.

[7] A. Fan *et al.*, "Large language models for software engineering: Survey and open problems," *arXiv preprint arXiv:2308.10620*, 2023.

[8] E. Nijkamp *et al.*, "CodeGen: An open large language model for code with multi-turn program synthesis," in *The Eleventh International Conference on Learning Representations*, 2023.

[9] R. Li *et al.*, "StarCoder: may the source be with you!," *arXiv preprint arXiv:2305.06161*, 2023.

[10] A. Chowdhery *et al.*, "PaLM: Scaling language modeling with pathways," *Journal of Machine Learning Research*, vol. 24, no. 240, pp. 1–113, 2023.

[11] B. Roziere *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[12] OpenAI, "GPT-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[13] J. Austin *et al.*, "Program synthesis with large language models," *arXiv preprint arXiv:2108.07732*, 2021.

[14] S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The impact of AI on developer productivity: Evidence from GitHub Copilot," *arXiv preprint arXiv:2302.06590*, 2023.

[15] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Peysakhovich, and C. R. Gonzalez, "Productivity assessment of neural code completion," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 21–29.

[16] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–7.

[17] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[18] H. Pearce, B. Ahmad, B. Raye, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? Assessing the security of GitHub Copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754–768.

[19] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at C: A user study on the security implications of large language model code assistants," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2205–2222.

[20] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with AI assistants?," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 2785–2799.

[21] C. S. Xia and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *Proceedings of the 45th International Conference on Software Engineering*, 2023, pp. 1482–1494.

[22] A. M. Dakhel, V. Majdianasl, E. Nikanjam, F. Khomh, M. C. Desmarais, and Z. M. Jiang, "GitHub Copilot as an AI programmer: Assessing its usefulness," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1061–1081, 2023.

[23] J. White *et al.*, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," *arXiv preprint arXiv:2302.11382*, 2023.