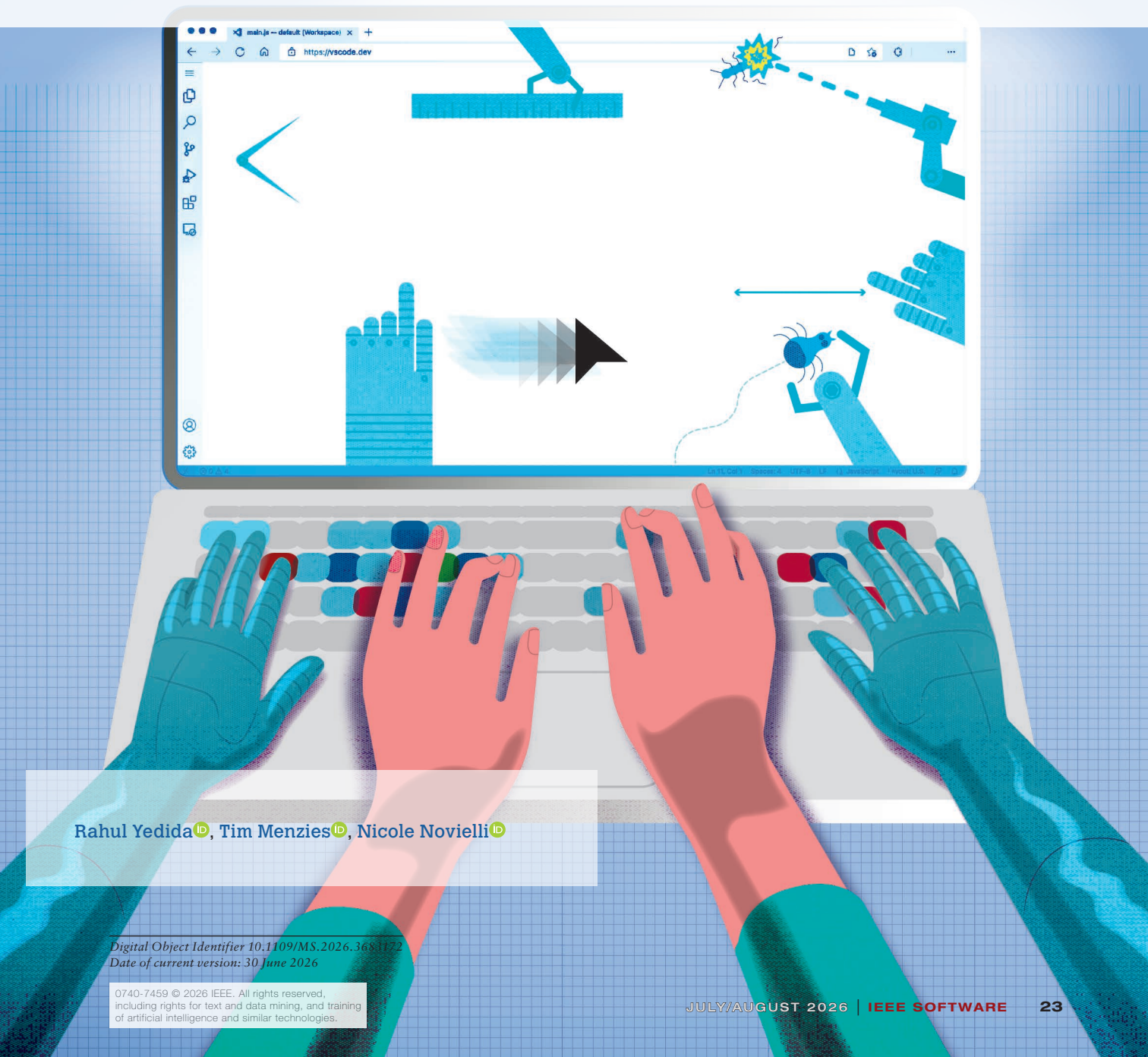
The Impact of AI on Productivity and Code



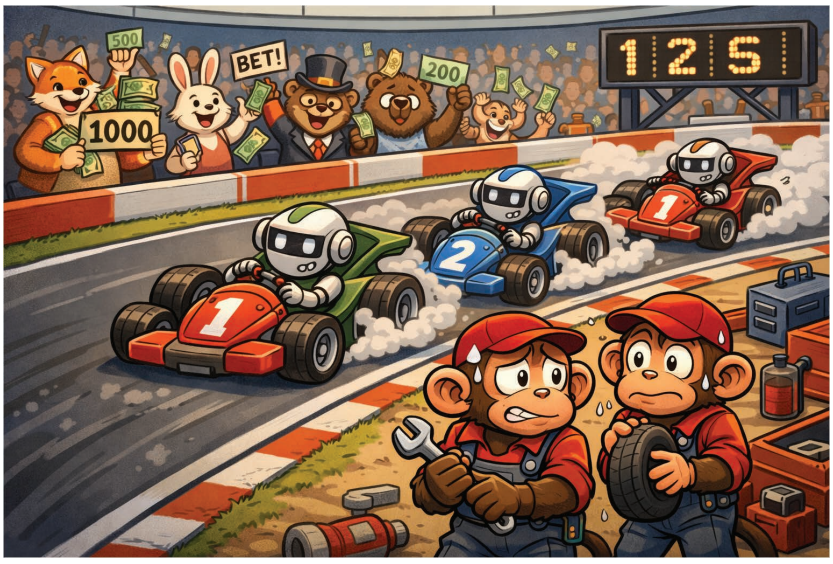
Rahul Yedida, Tim Menzies, Nicole Novielli

Digital Object Identifier 10.1109/MS.2026.3681172
Date of current version: 30 June 2026

SOFTWARE PRODUCTIVITY CAN be measured in many ways: decreased time to ship new products, customer satisfaction, maintenance cost, number of new ideas discovered, lines of code written per day, …(and that last one is very controversial; Linus Torvalds recently proclaimed anyone who uses it for evaluation is "too stupid to work at a tech company").[1] However it is measured, the open question of 2026 must be does AI [specifically, large language models (LLMs)] make software engineering more productive?

The rapid advancement of large language models (LLMs) has fundamentally transformed code generation, creating both significant opportunities and notable challenges in software development. Much has been claimed about this new technology, often by the vendors trying to sell us the services of large language models. But where is the empirical evidence of the value (or otherwise) of this exciting new technology?

To answer that question, this issue of *IEEE Software* asked for contributions on the productivity effects of AI in modern software engineering. We asked readers and researchers for articles that examine the current state of LLM-generated code, highlighting its benefits and persistent issues.



AI has its supporters. But who is checking?

Benefits of LLM-Generated Code

Enhanced Developer Productivity

LLMs have demonstrated remarkable capabilities in automating various software engineering tasks, leading to substantial productivity gains. GitHub Copilot, one of the most prominent AI-powered programming assistants, has been adopted by over one million developers and more than 20,000 organizations, generating more than three billion accepted lines of code.[2] The tool has proven effective across diverse applications, from code generation and summarization to bug fixing and software testing.[3]

Automated Code Quality Assurance

LLMs have shown promise in providing "assured LLM-based software engineering," where generated code comes with automated verifiable guarantees about semantic properties. This approach addresses hallucination concerns by providing semantic guarantees that are often as strong as those provided by human engineers.[4] The technology works in concert with human engineering effort rather than replacing it, building on existing code and tests to provide measurable improvements.

Practical Industrial Applications

Real-world deployment evidence demonstrates the practical value of LLM-generated code. Meta's TestGen-LLM tool, which uses LLMs to automatically improve existing human-written tests, achieved notable success in industrial settings. During Meta's Instagram and Facebook test-a-thons, the tool improved 11.5% of all classes to which it was applied, with 73% of its recommendations being accepted for production deployment by Meta software engineers.[4] In evaluation on Instagram's Reels and Stories products, 75% of TestGen-LLM's test cases were built correctly, 57% passed reliably, and 25% increased coverage.

To be complete, it is perhaps somewhat unfair to mention such numbers without a baseline comparison. The lack of baselines is a common problem in this era of LLMs and SE. For example:

- In April 2025, Microsoft CEO Satya Nadella famously proclaimed that "maybe 20%, 30% of the code that is inside of our (Microsoft) repos today…are probably all written by software." While this sounds like a ringing endorsement of AI technologies, it is known that Microsoft has been automatically generating its code, via traditional build systems, for decades.[5]
- For another example, in a recent systematic review [80] of 229 SE papers using large language models (LLMs), only 13/229 ≈ 5% of those papers compared LLMs to other approaches. This is

a methodological error since other methods can produce results that are better and/or faster.[6]

Significant Issues and Limitations

While the previously discussed benefits are undeniable, every new technology has its issues, and that includes AI.

Code Bloat and Technical Debt in LLM-Assisted Development

The "all-too-easy" nature of LLM-based editing frequently results in significant code bloat as models prioritize local task completion over global architectural efficiency. Automated suggestions often introduce redundant logic and boilerplate, leading to a 30% increase in repository volume without a proportional gain in functionality.[7] This "lazy integration" bypasses modular refactoring, creating fragmented codebases that are increasingly difficult to maintain. Consequently, the ease of generation masks a growing technical debt, where the sheer volume of low-entropy code complicates future debugging.

Security and Quality Concerns

Empirical studies reveal serious security implications of LLM-generated code. Research indicates that over 60% of model-generated C code contains security vulnerabilities,[8] with studies showing a 10% increase in security flaws compared to human-written code.[3] Roth et al.[9] identified a higher incidence of vulnerabilities in LLM-generated code compared to human-written code, while Asare et al.[10] found that GitHub Copilot replicated security vulnerabilities in 33.3% of cases, though it successfully mitigated 25.5% of them.

Domain-Specific Performance Limitations

LLMs exhibit significant performance variations across different application domains. Research on MultiCode-Bench reveals that all studied LLMs perform poorly on domain-specific tasks, with closed-source models like GPT-4 and GPT-3.5 showing relatively suboptimal performance in domain-specific application development tasks, sometimes falling below open source alternatives like DeepSeek-Coder-33B.[2] Common failure factors include lack of relevant domain knowledge, absence of repository context, and challenges in handling third-party libraries and their APIs.

Complex Instruction Following and Long-Context Challenges

LLMs struggle significantly with complex instructions and multitool usage. Extensive evaluation of 60 LLMs on BigCodeBench showed that models are not yet capable of following complex instructions to use function calls precisely, with scores up to 60%, significantly lower than human performance of 97%.[11] Additionally, long-context performance remains a critical weakness, with dramatic performance drops observed as context length increases—for example, Claude 3.5 Sonnet's accuracy on LongSWE-Bench drops from 29% to 3% when context length increases from 32,000 to 256,000 tokens.[12]

Academic Integrity and Evaluation Challenges

The widespread use of LLMs has created significant challenges for academic integrity and talent evaluation. More than 60,000 scientific articles in the past year alone have shown evidence of machine-generated content, undermining educational integrity as professors unknowingly grade machine-generated submissions.[13] This necessitates the development of robust detection mechanisms to maintain accountability in educational and professional settings.

Environmental and Computational Costs

The computational expense of LLMs raises significant sustainability concerns. Training models like StarCoder (7 billion parameters) resulted in 16.68 tons of CO_2 emissions, equivalent to four round-trip flights between Los Angeles and Rome.[3] Moreover, inference costs often surpass training costs over prolonged periods, highlighting the need for efficient deployment strategies.

Current Detection and Quality Assessment

When faced with problems, software engineers build tools to detect, then even mitigate those issues. This is true in many areas, including AI.

Machine-Generated Code Detection

Research has developed sophisticated methods for detecting LLM-generated code. Studies show that detection models can achieve up to 97.56% accuracy in identifying machine-generated code, with even simpler models like SVM and CatBoost performing considerably well.[13] However, performance drops significantly in unseen domains and hybrid generation scenarios, highlighting the ongoing arms race between generation and detection capabilities.

LLM-as-a-Judge for Code Evaluation

The emergence of LLM-as-a-judge paradigms offers promise for scalable code evaluation. This approach provides execution-free, reference-free, and multifaceted evaluation capabilities that can assess intrinsic code qualities traditionally requiring human judgment, such as readability and usefulness.[14] However, current implementations face limitations including reliance on small-scale data sets for validation and vulnerability to adversarial attacks.

Quantization and Optimization

Research on quantization techniques shows promise for reducing computational costs while maintaining code quality. Studies indicate that quantization can be applied to large code models without significant degradation in quality, though practical tradeoffs must be carefully considered, particularly regarding readability and maintainability.[2]

In This Special Issue

To say the previous statement another way, there is much we need to study about LLMs. The following articles in this special issue explore many aspects of this exciting, yet challenging, new technology. Our article is divided into three groups.

Productivity and Developer Experience

Fortes et al. ("The Productivity Paradox of AI-Powered Development")[A1] investigate AI coding assistants in an industrial setting, finding a fundamental tension: the same tools that accelerate routine tasks can hinder developers when cognitive demands are high or interaction patterns are poorly matched to the work. Productivity, they argue, is not a property of the tool but of the context.

Winckler et al. ("AI-Powered Collaboration: Exploring Developer Experience with GitHub Copilot and Windsurf")[A2] ground this question in a fintech company, combining usage metrics with qualitative interviews. Developers report genuine gains in productivity, learning, and code structure—but the study is equally clear that AI suggestions require constant human scrutiny, and that uncritical adoption introduces real risk.

Scholtz et al. ("Productivity Gains, Community Strains: AI Training on Open-Source Knowledge and its Impact on Stack Overflow")[A3] zoom out to the ecosystem level, using Netnography to trace how Stack Overflow contributors have reacted to the platform's AI initiatives. The findings are sobering: declining participation, moderator strikes, and deep concerns around attribution, trust, and compensation suggest that productivity gains at the individual level may be coming at a collective cost to the open source knowledge commons those tools were trained on.

Evaluation and Trust

Esirik and Gokalp ("QAID: A Comprehensive Framework for Evaluating AI Coding Assistants Beyond Vendor Claims")[A4] argue that vendor benchmarks are inadequate guides for tool selection and propose QAID—a structured evaluation framework adapted from ISO/IEC 25010—to fill the gap. Applied to ChatGPT, Gemini, Copilot, and DeepSeek across six quality dimensions, the framework reveals that no tool dominates: ChatGPT leads on generation quality, Gemini on sustainability, and the right choice depends heavily on organizational context.

Swaraj and Garg ("Trust But Verify: Measuring the Trade-offs of AI-Generated Code")[A5] approach the trust question empirically, quantitatively analyzing ChatGPT's answers to 4,000 Stack Overflow queries for both quality and security issues, while also mining developer forums for real time sentiment. Their headline finding is that AI and human responses are broadly comparable in competence—but developer trust remains fragile and context-sensitive, shaped by concerns that raw quality metrics alone do not capture.

Safety and Correctness

Pandey and Garg ("Secure Code Generation with Open Source Generative AI Models")[A6] conduct a systematic security audit of six open source LLMs, using prompts with varying levels of security awareness and the static analysis tool Bandit to surface vulnerabilities. The results are consistent and concerning: security weaknesses recur across models and domains, and no amount of careful prompting fully compensates. The authors conclude that postgeneration validation with robust static analysis tools is not optional but essential.

Read together, the articles of this special issue paint a consistent picture. AI coding tools are genuinely useful—they accelerate work, support learning, and lower barriers to entry. But they do not substitute for human judgment. Whether the question is security, correctness, trustworthiness, community health, or simply knowing when to override a suggestion, the human remains indispensable. The field's central challenge is not whether to adopt these tools, but how to build the oversight structures, evaluation frameworks, and community norms that make adoption responsible.

References

1. M. Müller, "Elon Musk is "too stupid to work in a tech company", claims Linux founder," *Notebookcheck*, Dec. 5, 2025. Accessed: Apr. 28, 2026. [Online]. Available: https://www.notebookcheck.net/Elon-Musk-is-too-stupid-to-work-in-a-tech-company-claims-Linux-founder.1178775.0.html
2. D. Zheng, Y. Wang, E. Shi, H. Zhang, and Z. Zheng, "How well do LLMs generate code for different application domains? Benchmark and evaluation," *Proc. ACM*, vol. 1, no. 1, pp. 1–22, 2024.

3. S. Afrin, B. Xu, and A. Mastropaolo, "Is quantization a deal-breaker? Empirical insights from large code models," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol. (ICSME)*, 2025, pp. 1–13, doi: 10.1109/ICSME64153.2025.00049.
4. N. Alshahwan et al., "Automated unit test improvement using large language models at Meta," in *Proc. 32nd ACM Symp. Found. Softw. Eng. (FSE)*, 2024, pp. 185–196, doi: 10.1145/3663529.3663839.
5. E. Kocaguneli, T. Zimmermann, C. Bird, N. Nagappan, and T. Menzies, "Distributed development considered harmful?" in *Proc. 35th Int. Conf. Softw. Eng. (ICSE)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 882–890, doi: 10.1109/ICSE.2013.6606637.
6. T. Menzies, "Retrospective: Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 51, no. 3, pp. 858–863, Mar. 2025, doi: 10.1109/TSE.2025.3537406.
7. M. L. Siddiq, S. H. Majumder, M. R. Mim, S. Jajodia, and J. C. S. Santos, "An empirical study of code smells in transformer-based code generation techniques," in *Proc. 22nd Int. Working Conf. Source Code Anal. Manipulation (SCAM)*, 2022, pp. 71–82, doi: 10.1109/SCAM55253.2022.00014.
8. T. Bisztray et al., "I know which LLM wrote your code last summer: LLM generated code stylometry for authorship attribution," in *Proc. 18th ACM Workshop Artif. Intell. Secur. (AISec)*, 2025, pp. 28–39.
9. S. Roth. "AI-generated code contains 2.74x more vulnerabilities than human code." svenroth.ai. Accessed: May 20, 2026. [Online]. Available: https://www.svenroth.ai/post/ai-generated-code-vulnerabilities-2-74x-4c9a7.
10. O. Asare, M. Nagappan, and N. Asokan, "Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code?" *Empirical Softw. Eng.*, vol. 28, Sep. 2023, Art. no. 129, doi: 10.1007/s10664-023-10380-1.
11. T. Y. Zhuo et al., "BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions," in *Proc. Int. Conf. Learn. Representations*, 2025.
12. S. Rando et al., "LongCodeBench: Evaluating coding LLMs at 1M context windows," 2024, *arXiv:2505.07897*.
13. D. Orel, D. Azizov, and P. Nakov, "CoDet-M4: Detecting machine-generated code in multi-lingual, multi-generator and multi-domain settings," in *Proc. Findings Assoc. Comput. Linguistics (ACL)*, 2024, pp. 10,570–10,593.
14. J. He et al., "From code to courtroom: LLMs as the new software judges," 2025, *arXiv:2503.02246v1*.

ABOUT THE AUTHORS

RAHUL YEDIDA is a senior data scientist with LexisNexis, Raleigh, NC 27606 USA. Contact him at 27606ryedida@ncsu.edu.

TIM MENZIES is a full professor at North Carolina State University, Raleigh, NC 27606 USA. Contact him at timm@ieee.org.

NICOLE NOVIELLI is a full professor at University of Bari A. Mori, 70125 Bari, Italy. Contact her at nicole.novielli@uniba.it.

Appendix: Related Articles

A1. L. Fortes, G. V. Pereira, A. Coelho, R. Prikladnicki, and K. Kohl, "The productivity paradox of AI-powered development," *IEEE Softw.*, vol. 43, no. 4, pp. 28–37, Jul./Aug. 2026, doi: 10.1109/MS.2026.3658651.
A2. S. C. Winckler, J. F. Ribeiro, L. Machado, J. Kroll, "AI-assisted collaboration: Exploring developer experience with GitHub Copilot and windsurf," *IEEE Softw.*, vol. 43, no. 4, pp. 38–46, Jul./Aug. 2026, doi: 10.1109/MS.2026.3671677.
A3. D. Scholtz, A. Griva, E. D. Zamani, and K. Conboy, "Productivity gain, community strain: Stack overflow's community response to its AI initiatives," *IEEE Softw.*, vol. 43, no. 4, pp. 47–55, Jul./Aug. 2026, doi: 10.1109/MS.2026.3665306.
A4. B. E. Esirik and E. Gökalp, "Quality assessment for AI development tools: A comprehensive framework for evaluating AI coding assistants beyond vendor claims," *IEEE Softw.*, vol. 43, no. 4, pp. 56–65, Jul./Aug. 2026, doi: 10.1109/MS.2026.3678750.
A5. A. Swaraj and S. Kumar, "Trust but verify: Analyzing the tradeoffs of AI-generated code," *IEEE Softw.*, vol. 43, no. 4, pp. 66–75, May/Jun. 2026, doi: 10.1109/MS.2026.3667154.
A6. M. Pandey and S. Kumar, "Secure code generation with open source generative AI models," *IEEE Softw.*, vol. 43, no. 4, pp. 76–86, Jul./Aug. 2026, doi: 10.1109/MS.2026.3672189.