

The Productivity Paradox: Strategic Implications of AI-Generated Technical Debt

Anil Poudyal

DBA Candidate, University of Wales Trinity Saint David

Abstract

This report critically investigates the implications of integrating Generative AI into software engineering. It examines the narrative that equates “operational velocity” with genuine productivity. Early positivist studies suggested massive efficiency gains, but recent literature reveals a profound “Productivity Paradox” defined by the emergence of cognitive technical debt. The findings from empirical study validates that the acceleration of syntax generation in software engineering is frequently offset by “integration friction”, including increased code churn, security vulnerabilities and the epistemic opacity of code that no human can fully comprehends.

The research argues that without rigorous governance, adoption of AI converts capital expenditure into opaque operating expenditure. Moreover, this report recommends software consultancies to bring a strategic shift in their organisational KPIs. The shift is to move away from “velocity” and “commit frequency” towards “system resilience” and “review density”. By expanding the idea of technical debt to cover both workforce deskilling and loss of cognitive competence on system design, this work helps senior engineers and software development stakeholders to use AI safely. Instead of letting it run unchecked as just a productivity booster, they can use AI as carefully managed tool that supports long-term strategy to build much more scalable and maintainable software systems.

Keywords: Generative AI, Technical Debt, Cognitive Load, Software Governance, Developer Productivity.

Introduction

Background and Context

Artificial Intelligence (AI) and Machine Learning (ML) has now become a good part of a life now directly or indirectly. From swapping card in train station to watching movies online. And so is the case for software engineers and software stakeholders. AI has touched and is being used by many sectors from finance to construction and from entertainment to education. It is now often referred to be industrial revolution 4.0 (Tai, 2020, pp. 339–343). Use of AI has become unavoidable, and a study suggests that AI are more trustworthy than search engines in various cases citing the nature of search engines with biased and promoted results (Kim and Priluck, 2025, p. 93).

And software industry is also one most influenced sector by these models. Improvement in productivity, faster deployments and early to market are the major promises that LLMs have made and in many cases, those promise are delivered. Many software engineers today use AI-

tools as their development assistants like ChatGPT, GitHub Copilots, Gemini, Jules, CodeWhisperer and so on; and these tools have promising results to suggest fixes, automate routine tasks and all in all to autocomplete code (Lyu *et al.*, 2025). These tools are powered by models which are trained on vast code repositories and promote productivity. A survey conducted in 2024 by GitHub showed that over 97% of respondent responded using AI for some point in time for productivity.

Not only these tools are used for software development to generate code snippet and code completion, but these are also even used for project planning and architecture planning. This shows how in depth the use of AI is in to development a software solution. And many users report that these tools are very productive: study shows Codeium, Copilot, and ChatGPT scored highly for ease of use and user satisfaction (Overflow, 2023).

Problem Statement and Research Focus

Even though AI assisted coding tools promise faster development speed and faster to production, a survey conducted in 2025 showed contrasting results, it showed that developers become slower when assisted with AI (Becker *et al.*, 2025). The major problem with AI assisted code generators and code completion tools is that they are not at all good at planning a big picture, research suggests that AI tools are not visionary enough to design long-term structure of software project (Gu *et al.*, 2025). Not only this the use of AI in development has raised serious concerns of technical risks and these have already begun to surface.

Strategic Rationale

There is emergence of new strategic risk due to divergence between perceived velocity and actual maintenance costs, this is cognitive technical debt. Traditional technical debts come from human shortcuts where developer is accountable for his/her messy to shift faster code but there persisted cognitive ability to understand the work. The logic and algorithm might have been messy and ugly but it's still legible. This could be refactored later because the intent was clear and there was existence of cognitive ability. But with AI-generated solutions that developers accepted the cognitive connection between the creator and code is severed. And this is backed by recent empirical studies and shows acute disconnect, for instance (Yeverechyahu, Mayya and Oestreicher-Singer, 2024) suggests that while there is good volume increase in contribution the innovation quality and long-term maintainability often degrade, reciting for long-run technical debt.

This transformation redefines the fundamental role of the software engineer. There is sheer shift from direct building and evaluating of logics to validation, evaluation of oversighted results produced by machine learning models. And this transformation introduces new challenges for accountability, traceability and quality assurance. This happens as existing governance, compliance and risk management frameworks are developed with the thought that human have direct involvement not human as mediator. Consequently, these frameworks are not well equipped to accommodate the oversight demands of systems in which behaviour emerges from models rather than explicit design. This leaves a significant gap between engineering practice and institutional mechanisms of control.

Furthermore, the strategic distribution of these risks is uneven. (Cui *et al.*, 2025) highlights that generative AI significantly aids lower-skilled workers, however it often disrupts the workflows of high-skilled experts. This introduces reverse mentorship a dynamic where senior engineer must spend more amount of time to debug small, hallucinated errors committed by junior level engineers. And this phenomenon aligns well with the findings of (Paradis *et al.*, 2025, pp. 618–629), who found out that in controlled corporate environments, the variance of work done speed creates unpredictability in delivery timelines and delivery quality.

Therefore, this research is philosophically grounded in the tension between the positivist and interpretivist reality. Here the positivist ground is used by management for example, lines of code, commit frequency, number pull request and number of merges. And interpretivist reality is experienced by developers for example loss of context and increased cognitive load. Current literature is predominantly focused on measuring speed of generative AI and language model adaption. This report argues what without a strategically valid framework to govern the cost of AI adoption, specifically in long run there will be significant amount of cognitive technical debt in the form of interest. This will be true as organisations risk maximising short-term efficiency at the expense of long-term system resilience.

Critical Literature Review

Productivity: benefits and the “paradox”

Many peer-reviewed studies and industry reports agree that AI code-assistants can boost productivity and reduce tedious work. Most of the experienced developers are found to be using AI assistants to generate boilerplate code and to document their work faster, so they can focus on complex issues. In one of the controlled experiments by (Peng *et al.*, 2023a) showed that the AI assisted development was more than 50% faster than those without, with no significant loss of solution quality; in this experiment developers were asked to implement HTTP server in JavaScript.

Another study done under Microsoft affiliate and GitHub’s own internal studies (Peng *et al.*, 2023b) reinforce this narrative of productivity boost: study suggests that developers report less mental fatigue and increase in perception of productivity gain using the tool Copilot. These studies however rely on self-reported measures and may not reflect complex, multi-staged software projects.

A critical examination of these evidence behind reported productivity gains reveals a clear methodological divide, and this is referred as the Productivity Paradox. Many of the literatures claim dramatic improvements but these figures rely on highly controlled experimental settings. And machine learning models can produce quickly and confidently, creating the appearance of substantial productivity gains. This seems gain initially but introduces technical debt in long run.

This claim of paradox is supported by significant evidence from (Becker *et al.*, 2025), in a randomised controlled trial the productivity gain was 19% which is less than the expected gain

of 24%. This study was done assessing 1.1 million lines of code committed across open-source projects. This suggests that while AI accelerates the generation of syntax, but it decelerates the integration of logic understanding and debugging code. (Cui *et al.*, 2025) suggests that while AI compresses the skill gap for junior developers it induces cognitive tax on senior engineers.

Therefore, the paradox is defined by the disconnection between perceived speed and actual quality delivery. For software firms, relying on speed and ignoring the quality delivery constitutes a significant strategic risk.

Technical Debt Injection by LLMs

While the previous theme highlighted the cognitive cost of AI adoption, another equally critical theme is direct injection of technical debt through code churn, and hallucinated dependencies. This section critically examines how this nature of generative AI transforms the codebase itself into a liability.

A common characteristics of AI-assisted software development is sharp increase in code churn; it is the phenomenon where the proportion of code that is rewritten or removed shortly after it is created. According to (Xiao *et al.*, 2025) the code churn has doubled in 2024, and authors interpret this as an early warning signal of declining code quality. Coding assistant such as Copilot can generate large number of lines of syntactically valid output at high speed but it is frequently incomplete, contextually inappropriate and subtly incorrect. As a result, engineers are bound to immediately revise, refactor or discard significant number of lines of code generated by AI leading to net zero or negative productivity and progress gain. Here the activity of engineer increases but durable contributions do not lead to almost non productivity gain and addition of technical debt. Ultimately this creates a downward pressure on maintainability and long-term software quality.

Therefore, this creates productivity illusion where metrics such as commits per day and number of line changes per day increases but the system's quality degrades. (Kodus, 2025) argue that this "hastily added code" acts as "caustic" for future maintainers and future project continuers. Human originated technical debt is often considered for trade off its speed and can be set lower priority for fix as it usually does not disregard the business logic, but AI-generated churn is often unintentional and is byproduct of the developer's inability to fully verify the massive volume of code produces.

Security Debt

AI assisted development has introduced new form of security debt described as package hallucination. This happens when large language models recommend software libraries or dependencies that do not actually exist or already obsolete contributing to the security debt (Al-Zo, Al-Zofi and Al-Zoq, 2025). Attackers can exploit this by registering these hallucinated names (slopsquatting) and hence injecting malicious code directly into the software supply chain. This risk is not just theoretical; (Spracklen *et al.*, 2024) found that 21.7% of code generated by open-source large language models contained references to non-existent packages. This represents a governance crisis.

These vulnerabilities often remain unseen during development and syntactically correct and only surface after deployment in production environments. At this point the remediation is costly and also will have chance of service disruption; this can also pose threat of data exposure. This reveals that there is lack of control rather than capability of the developers and other stakeholders. Governance frameworks designed for human authored code struggle to contain risks introduced directly or indirectly by AI systems operating outside traditional control and review assumptions.

Vibe Coding

The literature also points to the degradation of architectural discipline. (Fawzy, Tahir and Blincoe, 2025) examines how developers increasingly accept AI-generated code based on perceived correctness. Here developers do not usually gain understanding leading to excessive boilerplate and structural redundancy. This leads to large volumes of repetitive code with minimal functional differentiation. Rather than accelerating meaningful development, this practice increases complexity and also introduced obscures architectural intent. (Fawzy, Tahir and Blincoe, 2025) frames this as a shift from deliberate programming to passive code consumption, warning that such fluency bias undermines future maintainability and raises long-term costs, particularly in large, evolving codebases.

(Shah and Bajpai, 2025, pp. 0056–0061) observe that AI tools perform well when producing standardised patterns, however these systems lack the situational and architectural context. This is very crucial to meaningfully refactor, generalise or abstract logic over time. The outcome is progressively bloated codebase, here the duplication persists, and structural consolidation rarely occurs. This reinforces the central argument of this report: when AI is adopted without proper and strict governance it does not eliminate engineering cost but redistributes it. Effort to reinforced and well calibrated version of traditional model is necessary to help reduce future maintainability issues.

Shift in Developer Competency

With the progress and increase in use of code generation tools, a clear divide is emerging between junior and senior engineers. This split raises deeper epistemological questions about what expertise means in software engineering. When foundational work is automated, emerging engineers risk losing opportunities to build good knowledge through practice and failure, while senior engineers are pushed further into roles centred on judgement, verification and system-level reasoning. This imbalance questions the long-term sustainability of a workforce where fewer practitioners fully understand the systems they are expected to maintain. This also threatens the continuity of skill development within the profession.

In traditional settings junior engineers honed their expertise though legitimate peripheral participation. During process engineers learn with low-risk tasks that slowly shape their understanding of system architecture. These tasks acted as a learning scaffolded, and developers learned by doing with safe boundaries. However, with the introduction of generative AI this pathway got disrupted, it automated very tasks that once helped juniors develop intuition and judgement. As a result, learning is skipped rather than accumulated and skill

formation becomes thinner and more uneven. (Acharya, 2025) provides quantitative evidence of this shift. This study reports 16.3% drop in junior-level job vacancies in 2024, relative to senior roles. This points to structural risk to how expertise is formed in the software engineering workforce it is critical than the short-term displacement and imbalance of expertise.

This dynamic is framed as Deskilling Trap, (Crowston and Bolici, 2025, pp. 1009–1023) argues that AI delivers short-term productivity gains for emerging developers. And it is often 27-39% higher than those observed for senior engineers. However, these gains come at cost, by removing the need to grapple with errors and incomplete understanding. AI prevents juniors from engaging in the productive struggle that underpins deep learning. This shows significant risk of junior developer becoming prompt operators rather than software engineers. They can generate large volumes of code but lack the ability to maintain, trace failures, explain reason for failures and adapt logic under pressure. This concern aligns with (Aguilar-Calderón, 2025, pp. 397–426) which warns that sustained reliance on tools such as GitHub Copilot degrades foundational skills in syntax, debugging and problem decomposition. Over time, this weakens the talent pipeline, producing future senior engineers. This creates a gap in industries and leads to insufficiently prepared to govern complex, safety-critical systems.

The Epistemological Crisis

A deeper philosophical challenge emerging in recent studies is the “Epistemology of Not Knowing.” In traditional software engineer with no to very AI assisted development, the software engineer who writes the code has justified true belief about its behaviour. As they are the one developing and testing logic, they can explain the true reason behind each decision and can explain why it works. With involvement of ai-generated code this connection is broken, and developers often integrate and rely on code without fully understanding its internal logic or edge cases. Over time, over reliance on large language models risks producing codebases where nobody genuinely knows how parts of the system operate. This undermines accountability, development expertise and long-term system reliability.

(Thanh, 2025) identifies this phenomenon as “Logic Opacity” where engineers accept AI-selected algorithms without understanding the trade-offs or underlying reasoning. This leads where the prototype works but the human maintainers cannot explain how and why. Similarly, (Cito and Bork, 2025) labels this practice “Vibe Coding” where software is developed from opaque, black-box fragments rather than deliberately designed components. This practical consequence is the accumulation of Epistemic Technical Debt. This debt renders refactoring, debugging and security auditing probabilistically uncertain.

Analysis of the Selected Study

The study selected for in-depth critical analysis is Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity (Becker *et al.*, 2025). This paper marks a clear break from the techno-optimist narratives that dominated latest literatures. Rather than relying on controlled benchmarks or predefined predictable task, it examines use case of AI assisted coding in the wild. It examined mature open-source projects characterised by high complexity and long-lived codebases. The findings are counter-intuitive, instead of productivity gains the study shows otherwise. It shows friction, rework and coordination costs

that reduced net productivity gains. By examining the fundamental factors in real development contexts, the paper challenges simplified claims about AI-driven efficiency and provides a more credible foundation for assessing the true organisational and technical impact of AI-assisted software engineering.

Research Objective and Context

The primary goal of (Becker *et al.*, 2025) was to quantify the speedup factor (S). It particularly investigated contemporary AI tools like Cursor Pro and Claude 3.5 Sonnet when used by experienced developers. Unlike earlier studies that relied on set of predefined and isolated tasks such as LeetCode style problems this study employed Randomised Controlled Trial integrated in mature open-source projects. The participants had average of five years of prior experience with the codebases involved. This context is critical as it closely reflects conditions typical to enterprise software consultancies. These enterprise systems are shaped by legacy systems, strict maintainability requirements and huge architectural dependencies. Here the productivity is constrained less by code generation speed and more by comprehension, coordination and risk management.

Key Findings

The initial estimation by experienced developers was that AI assisted development would increase productivity by 24% in terms of speed. In estimation external experts were more confident with economist estimating 39% of gain and machine learning experts estimating 38% of gain.

The study showed clear divergence between perceived and actual productivity gains. It was the opposite when AI assisted development was used the time increased by 19% corresponding to speedup factor of approximately $S=0.84$. Rather than accelerating delivery, AI slowed experienced developers down.

The qualitative analysis explains why, although AI reduced the time needed to produce initial syntax and functions, it introduced integration friction. The generated code contained defects and hallucinations what were difficult to defect. This is common problem with large language models. So, the use of AI in development and engineering requires a strict quality and review standards with extensive verification, testing and debugging. In highly complex and huge codebases, this overhead outweighed any gains from faster code generation. Rather it turned apparent speed into net delay.

Strategic Relevance

For this report (Becker *et al.*, 2025) marks as the focal study as it provides a rare empirical validation of technical debt introduced by AI assisted development. This study directly challenges the positivist assumption that increased code output automatically translates into increased value. But it showed the opposite it constrained the productivity. The findings force a re- evaluation of how AI tools interact and how its output can be used in complex environments. This study also shows that the knowledge of experts that is gained through long-term engagement with complex systems is far superior to the outputs those are results of less contexted and narrow prompts.

Ontology & Epistemology

To fully understand the implications of authors it is necessary to first debunk the philosophical foundations that guides the study. Ontology concerns the nature of reality, it is about understanding what exists and what can be said to be real. But epistemology in contrast concerns the nature of knowledge, focusing on how we come to know that reality and what counts as valid understanding. These concepts shape how research questions are framed, how evidence is interpreted and also how conclusions are justified. In context of AI-assisted software development this is very important because technology challenges both what is considered work and who can legitimately claim knowledge of how that work is produced.

Ontological Stance: Critical Realism

The study adopts critical realist ontological position. Here the authors do not consider productivity as a single, directly observable variable and reject positivist view in which productivity is reduced to clock time or output volume. Instead, they assume a stratified reality in which different entities and phenomena coexist and interact.

At the empirical level, the study measures observable outcomes for instance time taken to close a GitHub issue. At the accrual level it accounts for events that occur regardless of observation. This includes developers writing code and large language models generating outputs. Most importantly, at the real level, the authors examine to uncover the underlying generative mechanisms that produce the observed slowdown. These mechanisms are not directly observable, but they are treated real and in fact pose significance.

Some of the examples include the cognitive load involved in reviewing and validating AI-generated code, the ongoing calibration for alignment of trust between human developer and large language models' outputs and strategic knowledge required to work safely with complex and legacy software infrastructures. By rejecting a simplistic equation of AI with speed, the authors argue that software engineering productivity emerges from the tactical interaction between human knowledge and machine produced output. In this view, productivity is not static and cannot be comprehended properly without proper governance, but it can be socially mediated and shaped by context, expertise and organisational constraints.

Epistemological Stance

Epistemologically, the paper adopts a hybrid position that is positivist in method but interpretivist in analysis. Generally, it aligns well with positivism. The use of randomised controlled trial often considered as the best standard of empirical research signals an attempt to produce objective and generalisable knowledge. And the calculation of speed factor i.e. $S = 0.84$ reinforces appearance of measurement-driven truth.

However, the authors' expectation of these results departs from a purely positivist stance. Rather than treating the findings where 19% slowdown as technical failure, the author seeks to understand it through the lived experience of the software engineers. The analysis also focused on subjective phenomena this includes integration friction, trust on generated code and cognitive effort required to verify the code.

There is a clear epistemological gap as there is a discrepancy between developer's expectations of being 24% faster and their actual speed being only 19% faster. The subjective feeling of productivity diverges from measurable outcomes. This distinction is central as it shows that governing AI adoption is not only about managing code and metrics but also about managing perceptions, confidence and control within expert teams.

Axiological Assumptions (Values)

Axiology addresses the question of what is considered valuable. In this study, (Becker *et al.*, 2025) implicitly challenge the dominant industry value system what only looks for velocity above all other factors. The narrative is set where majority of developers and other stakeholders assume that faster code generation is inherently desirable and that alone is the factor for increased productivity. But in contrast, the authors choose to evaluate AI impact in mature, high-complexity projects worked on by experienced developers.

This choice reflects a different set of values, and this study prioritises robustness, maintainability and long-term system integrity over short-term speed. Productivity is often looked with how quickly the code can be produced. But conversely the productivity metrics should have been looked where how safely and coherently it can be integrated into existing architectures. The study findings recommends that in highly critical environments, the value lies in correctness, reviewability and alignment with architectural intent. And there are precisely the dimensions where current AI tools are not performant enough. In doing so, the study concludes that AI is not universal accelerator but a technology whose value is highly subjective, context-dependent and constrained by governance and quality standards.

Reliability, Validity, and Generalisability

A robust critical review must therefore examine whether the findings of (Becker *et al.*, 2025) are methodologically sound and whether their conclusions can be reliably applied beyond the narrowed context of the study. This requires critical and strategically valid research design, choice of sample population, the tasks selected, and assumptions embedded in the measurement of productivity. It also demands an assessment of external validity: the extent to which results observed in the context of mature open-source projects with involvement of highly experienced software engineer contributes can be generalised to other organisational settings. These settings can include enterprise teams, consultancies or greenfield development environments.

Reliability is very crucial for papers of this importance. The most obvious limitation of this study is small sample size. With only 16 developers the generalisability of this study can be questioned. In quantitative social science, the sample size would normally imply low statistical power and low sample size can lead to risk where genuine effects may go undetected.

However, the authors partially mitigate these limitations by not just relying on single observation per participant. The study analyses 246 distinct tasks in total. Each software engineer completed approximately fifteen tasks, with agenda to allow performance to be assessed across multiple observations. This approach reduces the influence of single variability and also helps to stabilise the estimates of productivity effects. While the small sample size still constrains generalisability the internal reliability of the findings is strengthened by the

density and structure of the collected data. And unlike (Peng *et al.*, 2023b) which often relied on self-reported survey data, (Becker *et al.*, 2025) created an auditable digital trail. This makes the measurement highly reliable even though the sample size limits the statistical robustness.

Ethical Considerations

The findings of (Becker *et al.*, 2025) highlight ethical concerns that extend well beyond the scope of productivity measures and also looks for maintainability and long-term sustainability of software product. This way the dissuasions does not limit to just how AI can make developers faster but whether it is morally feasible to integrated into the critical infrastructures. These tools generate outputs that are computed using probability theory rather than accountability and their suggestions become production code. And when a failure occurs it becomes unclear who is answerable: the developer, the organisation, the model or the owner of the model. Ethical governance therefore demands more than efficiency claims.

The study also highlights to a non-trivial risk of model hallucinations where large language models generate code that is credible but functionally incorrect.(Becker *et al.*, 2025) primarily quantifies this risk in terms of time lost to debugging and verification. As highlighted in the wider literature including (Al-Zo, Al-Zofi and Al-Zoq, 2025) , large language models frequently recommend non-existent software packages. This behaviour is not an ignorable technical fault; it constitutes a form of automation misinformation. When software engineers trust an AI tool that is known for fabrication of dependencies, they may be unknowingly propagating false information onto the software supply chain. This can be threat as it creates opportunities for malicious actors through practices such as slopsquatting. From an ethical standpoint, this undermines the duty of care that underpins engineering profession. In doing so, the responsibility of safety is displaced from governance structures to individual software engineer operating under unrealistic productivity pressures.

The study also only emphasis on experienced developers with average tenure of five years. This also masks a more troubling ethical implication for the workforce pipeline. And this productivity analysis is not same for other engineers with fewer years of experience then the ones in sample size. This corrective labour reflects their capacity to judge when the AI was wrong. Junior developers do not share this advantage due to a smaller number of experience and possible not enough expertise. As junior developers are more likely to accept Ai output without rigorous verification due to limited experiential grounding. This produces a two-tier ethical risk.

Exploitation of Juniors

Organisation may pressure less experienced developers to use AI to meet the organisational velocity targets. This may expose them to failure when unverified code breaks in production. Responsibility is then individualised, despite structural causes.

Erosion of Agency

With automation the function what was used as training mechanisms for emerging developers the industry undermines the ladder of skill acquisition. And this might raise a question of

intergenerational justice. Currently the gains are extracted by senior leadership, while the long-term capacity of the profession is quietly depleted.

Discussion & Contribution

The synthesis of literature review with methodological analysis of (Becker *et al.*, 2025) exposes a significant “Governance Gap”. The industry continues to operate under what can be termed as Positivist Delusion: the assumption that faster code generation automatically produces greater value. And this belief contradicts with the evidence from highly complex and mature software projects. The velocity gained at the point of generation are frequently offset by downstream cost, this includes time spent on verifying, debugging and integrating AI-generated output. This trade-off constitutes a form of cognitive technical debt. In this trade-off the productivity is borrowed from the future. Without deliberate governance mechanisms, organisations and individuals risk collecting invisible liabilities that compromise maintainability, developer expertise and system resilience. The gap is not only technical it includes ethical epistemological and organisational, highlighting the urgent need for governance frameworks that manage both large language models’ output and its long-term consequences.

The major finding of this research is that it does not eliminate labour entirely, but it transforms it by augmenting tasks. The working patten and workload shift now from construction such as writing syntax to verification, including reviewing, debugging and integrating AI-generated code. Current organisational structures, which emphasise metrics like velocity and commit frequency, are poorly suited to capture or manage this transformed labour. And with these senior developers have an added workload to validate and correct stochastic large language model’s output acting as bottleneck. This directly explains the 19% slowdown observed by (Becker *et al.*, 2025). Rather than increasing velocity of work, AI redistributes effort making measurement of value more complex and highlighting necessity of governance mechanisms. These mechanisms should account for the cognitive and verification costs inherent in AI assisted software development.

The study also extends the technical debt to three different extensions. Following are the extensions.

Traditional Technical Debt

Traditional technical debt is typically intentional and visible; tech team and other stakeholders accept the functioning feature over optimised solutions at the very first of deployment. This known debt can be fixed with ease. This trade-off is explicit and managed.

AI-Generated Technical Debt

By contrast, AI-generated technical debt is largely unintentional and opaque. It shows up through practices such as slopsquatting where large language models introduce hallucinated packages and fault code into supply chain risking the pipeline. In these cases, the debt accumulates without conscious acknowledgement.

Theoretical Proposition

AI generate code should not be treated as finished asset by default. Instead, it must be understood as liability until proven otherwise and should go through rigorous testing and review. This reframes the ontological status of AI output from value to risk. This also challenges the techno optimist expectations that is currently dominate management and productivity discourse.

Conclusion

This report has examined the strategic implications of generative AI in software engineering. This report looked beyond the narrow narrative of operational efficiency. Ai reshapes how knowledge, responsibility and risk are distributed within software development stakeholders. Without proper governance apparent gains in speed can increase long-term costs in verification, review, skill erosion and system fragility.

The synthesis of recent literature clearly points to Productivity Paradox. Early studies framed AI as accelerator factor for productivity but these claims rest on synthetic and low complexity testing environments. This challenges that assumption. The detailed methodological critique of Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity (Becker *et al.*, 2025) provides a corrective and empirical evidence. In mature codebases the AI has done exactly the opposite in productivity gains, the velocity was reduced by 19%. This is due to net reduction of time due to time consumption in reviewing and verifying AI-generated code snippets.

Theoretically, this report redefines the meaning of developer competency in AI-mediated environment. The deskilling trends shows growing risk of hollowing out the middle layer of expertise. These are engineers who traditionally bridge junior learning and senior architectural judgement. Without this layer the profession loses the capacity of producing future architects and tech leaders.

The true cost of AI is not just the subscription fee, but it poses long term risk on maintainability and scalability of systems due to unverified code. The governance of this phenomena should come before adoption and this adoption should be strategic adoption with clear review of prospects and issues. Without clear policies and proper control, AI-generate code in production can be optimisation task for today but the system can become paralysis in long term. The organisations need to put attention away from raw velocity and towards system resilience, maintainability, and accountable knowledge.

References

- Acharya, V. (2025) “Generative AI and the Transformation of Software Development Practices.” Available at: <https://doi.org/10.48550/arxiv.2510.10819>.
- Aguilar-Calderón, J.A. (2025) The Deskilling of Software Development and the Impact of AI Chatbots on Programmers’ Skills and Roles. United States: IGI Global (Generative AI in Software Engineering), pp. 397–426. Available at: <https://doi.org/10.4018/979-8-3373-0370-3.ch013>.
- Al-Zo, W., Al-Zofi, W. and Al-Zoq, W. (2025) “AI-Induced Supply-Chain Compromise: A Systematic Review of Package Hallucinations and Slopsquatting Attacks.” Springer Science and Business Media LLC. Available at: <https://doi.org/10.21203/rs.3.rs-8007192/v1>.
- Becker, J. et al. (2025) “Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity.” Available at: <https://doi.org/10.48550/arXiv.2507.09089>.
- Cito, J. and Bork, D. (2025) “Lost in Code Generation: Reimagining the Role of Software Models in AI-driven Software Engineering.” Available at: <https://doi.org/10.48550/arxiv.2511.02475>.
- Crowston, K. and Bolici, F. (2025) “Deskilling and upskilling with AI systems,” 30(iConf), pp. 1009–1023. Available at: <https://doi.org/10.47989/ir30iConf47143>.
- Cui, K.Z. et al. (2025) “The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers *.”
- Fawzy, A., Tahir, A. and Blincoe, K. (2025) “Vibe Coding in Practice: Motivations, Challenges, and a Future Outlook – a Grey Literature Review.” Available at: <https://doi.org/10.48550/arxiv.2510.00328>.
- Gu, A. et al. (2025) “Challenges and Paths Towards AI for Software Engineering.” Available at: <https://doi.org/10.48550/arXiv.2503.22625>.
- Kim, S. and Priluck, R. (2025) “Consumer Responses to Generative AI Chatbots Versus Search Engines for Product Evaluation,” 20(2), p. 93. Available at: <https://doi.org/rg/10.3390/jtaer20020093>.
- Kodus (2025) “How AI-Generated Code is messing with your Technical Debt.” Kodus (Kodus), 3 March. Available at: <https://kodus.io/en/ai-generated-code-is-messing-with-your-technical-debt/>.
- Lyu, Y. et al. (2025) “‘My productivity is boosted, but ...’ Demystifying Users’ Perception on AI Coding Assistants.” Available at: <https://doi.org/10.48550/arXiv.2508.12285>.
- Overflow, S. (2023) Stack Overflow 2023 Developer Survey. Available at: <https://survey.stackoverflow.co/2023/#overview>.
- Paradis, E. et al. (2025) “How Much Does AI Impact Development Speed? an Enterprise-Based Randomized Controlled Trial,” in. IEEE (IEEE/ACM International Conference on Software

Engineering: Software Engineering in Practice (Online)), pp. 618–629. Available at: <https://doi.org/10.1109/ICSE-SEIP66354.2025.00060>.

Peng, S. et al. (2023a) “The Impact of AI on Developer Productivity: Evidence from GitHub Copilot.” Available at: <https://doi.org/10.48550/arxiv.2302.06590>.

Peng, S. et al. (2023b) The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. Ithaca: Cornell University Library, arXiv.org. Available at: <https://www.proquest.com/docview/2776278155>.

Perry, N. et al. (2023) “Do Users Write More Insecure Code with AI Assistants?,” in. New York, NY, USA: ACM (ACM Conferences), pp. 2785–2799. Available at: <https://doi.org/10.1145/3576915.3623157>.

Shah, J.A. and Bajpai, G. (2025) “Responsible Generative AI for Software Development Life Cycle,” in. IEEE (2025 IEEE World AI IoT Congress (AIIoT)), pp. 0056–0061. Available at: <https://doi.org/10.1109/AIIoT65859.2025.11105309>.

Spracklen, J. et al. (2024) “We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs.” Available at: <https://doi.org/10.48550/arxiv.2406.10279>.

Tai, M. (2020) “The impact of artificial intelligence on human society and bioethics,” 32(4), pp. 339–343. Available at: https://doi.org/10.4103/tcmj.tcmj_71_20.

Thanh, N.H. (2025) “The Epistemology Crisis in AI-Assisted Development.” Medium (Medium), 28 October. Available at: <https://blog.nguyenthanh.asia/the-epistemology-crisis-in-ai-assisted-development-when-not-knowing-becomes-the-new-normal-fc8c8e2a1788>.

Xiao, T. et al. (2025) “Self-Admitted GenAI Usage in Open-Source Software.” Available at: <https://doi.org/10.48550/arxiv.2507.10422>.

Yeverechyahu, D., Mayya, R. and Oestreicher-Singer, G. (2024) “The Impact of Large Language Models on Open-source Innovation: Evidence from GitHub Copilot.” Available at: <https://doi.org/10.48550/arxiv.2409.08379>.