Beyond Productivity: Evaluating the Hidden Costs of Generative AI in Software Development

Edward G. Anderson * Geoffrey G. Parker † Burcu Tan ‡

December 1, 2025

Abstract

AI-assisted programming tools are increasingly adopted to enhance developer productivity by automating routine tasks and accelerating software development. While early evidence highlights short-term productivity gains, our study shows that excessive reliance on AI can unintentionally accelerate the accumulation of technical debt, leading to escalating maintenance costs and eventual revenue erosion. Drawing on interviews with software engineers and insights from the literature, we build a dynamic simulation model to examine the long-term impacts of AI-assisted coding under varying conditions of project horizon, legacy-system interaction, developer skill, and market dynamism. Our results reveal that productivity benefits follow an inverted-U pattern: there exists a "sweet spot" where AI assistance improves overall performance, but beyond this threshold, compounded technical debt undermines system maintainability and profitability. We further show that greenfield environments and fast-evolving markets can tolerate higher levels of AI assistance, whereas brownfield (legacy) settings and teams with less skilled developers are substantially more susceptible to long-term decline. We identify technical-debt retirement as a strategic guardrail that can help sustain AI benefits over time. These findings demonstrate that AI-assisted development is not simply a choice of productivity tool, but a strategic decision that must consider programming environment constraints, system longevity, skill composition, and market factors.

Keywords: Human-AI Interaction, AI-Assisted Coding, Software Development, Technical Debt, System Atrophy, Dynamics

*McCombs School of Business, The University of Texas at Austin
†Thayer School of Engineering, Dartmouth College
‡Anderson School of Management, The University of New Mexico

1 Introduction

Generative AI has entered the software development landscape with ambitious promises. Tools like GitHub Copilot, Cursor, and Claude Code offer to write code, craft tests, and even design systems. Early evidence suggests that these tools can indeed accelerate coding tasks, reduce time spent searching documentation, and assist in the generation of test cases. Experimental evaluations and industry reports frequently tout productivity gains ranging from 20% to 55% (Peng et al. 2023, Cui et al. 2024, Paradis et al. 2025, Bakal et al. 2025), and several organizations have begun integrating AI tools into their development processes with the expectation of significant cost savings and performance improvements. Some industry leaders have reinforced these expectations: for example, Dario Amodei, CEO of Anthropic, has predicted that "we will be there in three to six months, where AI is writing 90% of the code. And then, in 12 months, we may be in a world where AI is writing essentially all of the code" (Amodei 2025). Similarly, Jacob Jackson, founder of the startup that created TabNine, argued in a 2024 interview that "there's no such thing as junior developers anymore because AI basically elevates everybody to be beyond that" (Li 2024). At Meta, Mark Zuckerberg recently announced that they "are going to have AI that can be a mid-level engineer," with the expectation that over time a large fraction of the company's software will be built by AI "engineers" rather than human ones (Marks 2025).

Despite these optimistic statements, the real-world experiences of professional software developers paint a more nuanced picture. Interviews that we conducted with veteran developers, discussions in the trade press and online engineering forums, and early field studies suggest that AI assistance can sometimes degrade code quality, increase complexity, and introduce new forms of risk, despite improving short-term output. Christine Yen, CEO of Honeycomb, recently observed that even in the most AI-amenable projects, productivity improvements appear to be capped at around 50%, and often far less. As she explained, "AI just frankly isn't good enough yet to be additive... The hard part about building software systems isn't just writing a lot of code. Engineers are still going to be necessary, at least today, for owning that curation, judgment, guidance and direction" (Wired 2025). Moreover, not all projects benefit equally from AI assistance. In particular, teams working on what Daniel Jackson of MIT calls "serious software" (i.e., large-scale software development) report far smaller gains from AI tools than in simpler tasks (Wired 2025). These mixed outcomes appear to depend heavily on factors such as the nature of the development environment, system complexity, and the expertise of the development team (Anderson et al. 2025).

We propose that technical debt (TD) offers a mechanism for explaining these divergent outcomes. Technical debt refers to the difficulty in revising, reusing, scaling, or maintaining

code that results from poor programming practices during software development (Tom et al. 2013, Woodard et al. 2013, Ramasubbu & Kemerer 2016). These practices include–among other factors–quick fixes, inefficient structure, inadequate commenting, or poor architecture that result in code that works in the short-term but is poorly aligned with evolving long-term design objectives. While AI-assisted coding accelerates feature development and increases code output, it also increases the rate at which such debt is generated (Aslan 2024*a*, Doerrfeld 2025, Gupta 2025). AI-generated code is often syntactically correct, yet semantically misaligned with system architecture, security requirements, or maintainability goals, making it harder to refactor or integrate later. As one senior engineer we interviewed expressed: *"AI creates a lot of subtle problems that are difficult to catch... The problem with AI is that it can't see the big picture."* Indeed, several experienced practitioners have observed that the current wave of AI-assisted development is generating technical debt at an unprecedented pace, faster than many have seen in decades of software engineering (Doerrfeld 2025).

Technical debt is not merely a theoretical concept—it has been at the root of several large-scale system breakdowns. The Y2K crisis illustrated how deferred architectural cleanup, quick fix "point solutions," and shortcut design decisions accumulated into deeply embedded risk, ultimately requiring an estimated $300 billion in global remediation (Kapani 2025). More recently, Southwest Airlines' 2022 operational meltdown, which stranded over 16,900 flights and cost the firm more than $750 million, was traced to outdated scheduling software that could no longer handle modern operational demands (US Department of Transportation 2023). Similarly, the 2024 CrowdStrike outage triggered widespread failures in healthcare, aviation, banking, and government services due to brittle dependencies embedded in system architectures (Vogel 2024). These incidents demonstrate that unmanaged technical debt can silently accumulate until it becomes an operational and financial liability and the accelerated code generation enabled by AI tools only increases the urgency of managing it proactively.

Security risks are also driven by the use of AI assisted coding. The mechanisms by which AI assisted coding increases security risks are related to technical debt acceleration. For example, Andrew Stiefel lists five key areas of vulnerability driven by AI coding: architecturally invisible flaws, dependency-related vulnerabilities from outdated suggestions, dependency hallucinations, injection risks due to weak input handling, and authentication gaps stemming from underspecified prompts (Stiefel 2025). Industry participant Zahra Timsah notes that "AI tools are not designed to exercise judgment ... They do not think about privilege escalation paths, secure architectural patterns, or compliance nuances. That is where the risk comes in ... Code gets shipped faster, but if oversight is thin, enterprises are effectively automating risk at scale" (Leyden 2025).

It is clear from these observations that AI assistance must be deployed carefully and

strategically. However, scholarly guidance on this remains limited. To address this gap, we investigate the following questions:

1) What are the long-term impacts of using Generative AI tools to increase software development productivity?

2) How much AI assistance should be used across the software development lifecycle to balance short-term productivity and long-term sustainability?

3) Under what conditions can AI allow junior developers—and potentially even non-development personnel—to achieve performance outcomes approaching those of senior developers?

4) How do factors such as development environment (greenfield vs. brownfield), market dynamism, and technical debt management strategies shape these decisions?

To examine these questions, a standard closed-form analytic model is infeasible. The system's state variables interact through delayed effects that accumulate over extended time horizons, producing outcomes that evolve differentially across multiple points in time depending on prevailing environmental conditions. We therefore develop a differential-equation simulation model, grounded in interviews with software engineers, documented industry cases, trade-press reporting, and prior research on technical debt, software evolution, and AI-assisted programming. This approach allows us to evaluate the intertemporal trade-offs between AI-assisted productivity gains and the accumulation and retirement of technical debt across varied environments, and to identify conditions under which AI assistance enhances long-term performance versus those under which it undermines it.

Our simulation results show that, although AI-assisted coding provides early productivity gains expanding product and service offerings, these benefits are not uniformly sustainable. Heavy reliance on AI accelerates technical debt accumulation by increasing code volume and generating context-insensitive, error-prone code, which over time reduces developer productivity and raises maintenance costs. These dynamics create a "sweet spot" in AI usage: moderate assistance improves long-term profitability and maintainability, whereas excessive reliance triggers a self-reinforcing debt spiral. In environments marked by legacy-system interactions, architectural complexity, and limited capacity for debt retirement, the negative effects of AI emerge earlier, compound more rapidly, and become harder to reverse, causing the optimal "sweet spot" for AI assistance to shift downward.

Our simulation results also reveal critical boundary conditions under which AI assistance either amplifies or undermines performance. In short-horizon projects, AI can be used more

aggressively because technical debt has insufficient time to build up or trigger downstream consequences. Fast-evolving markets show similar tolerance, as rapid product turnover limits the long-run impact of accumulated debt. In contrast, slow-evolving markets such as complex enterprise systems are far more vulnerable: here, technical debt persists, compounds, and ultimately erodes performance unless AI adoption is managed much more cautiously.

We find that developer expertise plays a similarly decisive role. While pairing junior developers with high levels of AI assistance can initially match or even exceed senior-level productivity without AI, juniors accumulate technical debt more quickly due to less proficient programming skills (Aslan 2024*a*, Osmani 2024). As a result, their early gains erode over time, ultimately reducing long-term productivity and profits. Counterintuitively, this implies that junior developers should rely on lower levels of AI assistance than senior developers in the long run.

Finally, we show that dedicating sufficient resources to technical debt retirement can serve as an effective guardrail, helping organizations sustain AI benefits over time. Together, these insights demonstrate that AI-assisted development is not simply a productivity problem but a strategic decision that must consider system longevity, architectural resilience, skill composition, and market factors.

The remainder of the paper proceeds as follows. Section 2 reviews the related literature. In Section 3, we introduce the simulation model, while in Section 4, we analyze the model to explore the effect of AI-assisted coding on productivity, technical debt, maintenance cost under different development environments. Finally, implications of the results are discussed in Section 5, along with the limitations of the current analysis and possible extensions. We conclude in Section 6.

2 Literature Review

This paper is mainly relevant to three streams of research: Human-AI Interaction, Technical Debt and System Atrophy, and AI-assisted software development.

2.1 Human-AI Interaction

As AI is increasingly used to augment human capabilities, human–AI interaction has emerged as a central concern in the information systems (IS) literature, with Berente et al. (2021) calling it "perhaps the key managerial issue of our time." An important tension in this domain lies in the distinction between automation and augmentation (Rai et al. 2019, Queiroz et al. 2024). Automation emphasizes replacing human labor with machine capabilities, whereas

augmentation focuses on enhancing human capabilities through intelligent systems. Recent IS research stresses AI–human complementarity, focusing on how humans and intelligent systems can collaborate effectively. This view aligns with the human-in-the-loop paradigm (Zanzotto 2019), which highlights the importance of keeping humans engaged in the design and implementation of AI-augmented systems (Dietvorst et al. 2015, Ge et al. 2021, Fügener et al. 2021).

Several studies have examined when AI helps to improve individual and organizational outcomes. For example, Fügener et al. (2022) show that hybrid decision-making systems where AI delegates tasks to humans can outperform either humans or AI working alone. However, when humans delegate to AI, the benefits diminish, underscoring the need for careful coordination. Revilla et al. (2023) find that human intervention adds the most value when AI-driven predictions involve long time horizons and low uncertainty, whereas automation may be more effective for predictions involving short horizons and high uncertainty. Similarly, Chen & Chan (2024) find that LLMs improve creative output for non-experts when used as sounding boards, yet may constrain creativity when used as a ghostwriter, particularly for expert users. These findings suggest that the key question is not whether AI is helpful, but rather how it can be effectively integrated into human workflows to improve outcomes. Our study explores this question in the context of software development through a prescriptive simulation framework.

A growing stream of human-AI complementarity research investigates how individual experience and expertise shape the effectiveness of human–AI collaboration (Jussupow et al. 2021, Luo et al. 2021, Wang et al. 2024). Multiple studies suggest that domain expertise influences both trust in and reliance on AI systems, which in turn affects performance outcomes. For example, in a study involving physicians using clinical decision support systems, Jussupow et al. (2021) find that experienced professionals are more likely to reject algorithmic recommendations, showing higher "algorithmic aversion" (Dietvorst et al. 2015). Allen & Choudhury (2022) observe an inverted-U pattern in performance benefits from AI: As domain experience increases, users initially benefit more, but eventually algorithmic aversion kicks in and performance gains decline. Wang et al. (2024) refine this further by distinguishing task-specific experience from broad experience based on seniority. They find that task-specific expertise enhances productivity when paired with AI, but seniority-related mistrust may offset these gains. Finally, Brynjolfsson, Li & Raymond (2025) note that junior workers benefit from LLMs through improvements in both speed and quality, while more experienced workers experience small declines in quality with marginal gains in speed. Altogether, this literature reveals that the benefits of human-AI collaboration are contingent upon the user's experience, trust, and collaborative strategy, highlighting the need for more

nuanced approaches to managing human–AI interactions.

We contribute to this literature by providing a nuanced understanding of how best to manage human-AI interactions in software development workflows. Our model not only distinguishes between different development environments ("greenfield" vs. "brownfield") but also incorporates the impact of skill level ("junior" vs. "senior" developers), exploring the long-term impact of AI assistance on performance outcomes.

2.2 Technical Debt and System Atrophy

Technical debt refers to the future maintenance obligations that arise when software developers or system designers take expedient but suboptimal design or coding shortcuts that violate established engineering standards (Ramasubbu & Kemerer 2021, Kruchten et al. 2012). Originally coined as a metaphor to capture the trade-off between short-term speed and long-term system health, technical debt has become a central concept in the software engineering literature (Brown et al. 2010, Tom et al. 2013, Li et al. 2015, Allen & Choudhury 2022), but remains a relatively nascent construct in the IS field (Woodard et al. 2013, Ramasubbu & Kemerer 2016, 2021). Manifesting in the form of poor documentation, entangled ("spaghetti") code, architectural rigidity, or system performance degradation, technical debt tends to increase over time and can severely undermine system reliability (Eick et al. 2002, Li et al. 2015, Ramasubbu & Kemerer 2016). In their empirical analysis, Ramasubbu & Kemerer (2016) find that accumulated technical debt significantly increases the probability of enterprise system failures. Similarly, Banker et al. (2021) show that technical debt negatively affects firm performance with the impact compounding as systems age.Technical debt also has significant cybersecurity implications (Ozkaya et al. 2021, BARR Advisory 2022).

A growing body of IS research highlights that technical debt not only impairs operational performance but also limits a firm's ability to innovate. Woodard et al. (2013) introduce the concepts of "design capital" and "design moves" to explain how firms strategically manage the technical architecture of digital products. Their two-by-two framework illustrates the ways architectural choices influence the balance between digital options and technical debt, shaping a firm's capacity for future innovation. Rolland et al. (2018) extend this perspective by examining the management of digital options and debt within evolving platform ecosystems. Rinta-Kahila et al. (2023) explore the unintended constraints that arise during legacy system replacement, showing how the interplay between social inertia, technical inertia, and the pursuit of digital options can trap firms in a debt-constrained state. This stream of research frames the interplay between digital options and technical debt as a dynamic process: digital options may be adopted at the cost of incurring new debt, while existing debt may

hinder the realization of digital options in the future. These dynamics have close parallels with the broader phenomenon of system atrophy theorized by Tiwana & Safadi (2024), in which information systems degrade into brittle and unmaintainable architectures over time. Tiwana & Safadi (2024) argue that though atrophy and technical debt are conceptually distinct, they are intertemporally linked, as unresolved debt can gradually lead to atrophy hollowing out the system's capacity for change. Taken together, this literature underscores the importance of viewing technical debt not merely as a technical artifact but as a structural constraint on organizational agility and innovation.

We contribute to this literature by showing that AI-assisted coding can act as a new accelerant of technical debt, particularly in brownfield projects or teams with less skilled developers. Embedding these effects in a system dynamics simulation allows us to explore how early productivity gains can mask future maintenance burdens, an aspect that has not yet been explicitly modeled in the literature.

2.3 AI-Assisted Software Development

The emerging literature on AI-assisted coding[1] presents a nuanced view of its effects on software development. Some of the earlier empirical studies (Peng et al. 2023, Cui et al. 2024, Paradis et al. 2025, Bakal et al. 2025) have reported substantial productivity gains, with developers completing tasks 20 to 55 % faster. Generative AI tools also support ideation, documentation, and have been positively received in greenfield or enterprise contexts. However, more recent findings highlight hidden costs (Anderson et al. 2025). Studies have documented issues such as low-quality output, limited acceptance rates, the need for manual adjustments, reliability concerns, and integration difficulties (Dakhel et al. 2023, Cámara et al. 2023, Russo 2024, Coutinho et al. 2024). Anecdotal evidence suggests that less experienced developers are particularly susceptible to propagating errors (Osmani 2024, Aslan 2024*b*). Recently, through a randomized trial, Becker et al. (2025) have found that AI assistance actually slowed experienced contributors working on mature codebases by approximately 19%. Only 44 % of AI-generated suggestions were accepted, and about 9 % of task time was spent reviewing and cleaning up AI output. Interestingly, developers thought they were experiencing a 24 % speedup despite the evidence to the contrary. In a similar vein, Medappa et al. (2025) find that AI-assisted programming may decrease productivity of experienced developers by increasing maintenance burden, showing that productivity increase is not a guaranteed outcome.

Overall, while productivity improvements are clearly documented in controlled settings,

[1]We would like to note that throughout the remainder of the paper, the term AI-assisted coding refers specifically to *Generative AI*–based assistance.

real-world usage of AI coding assistance introduces context-dependent challenges that can undermine efficiency, especially in legacy environments. It can also bring about severe cybersecurity concerns (Ji et al. 2024, Haque et al. 2025). We contribute to this research by examining the long-term consequences of AI use in software development and uncovering managerial insights for optimal usage levels. Specifically, we differentiate conditions under which AI assistance yields sustainable productivity advantages (greenfield, experienced developers) versus those in which it generates too much liability that outweighs the productivity benefits (brownfield, junior developers). By articulating these boundary conditions, our study complements existing empirical findings through a prescriptive framework and provides a dynamic perspective on how AI assistance may reshape software engineering trajectories.

3 The Model

The evolution of AI-assisted software development is characterized by intertwined feedback mechanisms (such as the self-reinforcing accumulation of technical debt), nonlinear relationships (such as the impact of debt on maintenance cost and productivity), and delayed effects (such as the gradual loss of service relevance). The simultaneous presence of these factors prohibits closed-form solutions for dynamic decision trajectories (Sterman et al. 2015). Hence, to study the dynamic evolution of software development processes in the presence of AI assistance, we build a system dynamics (SD) simulation model. System dynamics methodology is well suited to this problem because it encourages a broad model boundary that explicitly accommodates feedback loops, delays, and nonlinear relationships (Sterman 2000). In addition, its near-continuous-time, differential-equation-based state-space formulations closely resemble those in optimal control theory (Anderson, Keith & Lopez 2022), enabling us to incorporate formulations grounded in information economics literature. Finally, in contrast to analytical models that focus on equilibrium outcomes, system dynamics models explicitly capture disequilibrium and transitional dynamics. (Sterman et al. 2015). This allows us to study the evolution of services and maintenance needs, analyzing how the path to the equilibrium is shaped by managerial decisions as well as the software development environment characteristics[2].

[2]For a detailed discussion of how system dynamics methodologies help overcome some limitations of traditional analytical frameworks, please see Esenduran et al. (2022) and Anderson, Keith & Lopez (2022).

3.1 Simplified Causal Loop Diagram

In this section, we present a simplified causal loop diagram for the model in order to explain the key model components and feedback loops (Figure 1). At the heart of our model are two state variables, Services and Technical Debt (TD). We consider a firm whose revenue depends on the software "services" it develops. Here, we use the term services generically to refer to the firm's codebase. Services could include mobile apps, enterprise software, etc., as well as internal code that supports other revenue generating functions.

Any software development activity generates some TD, which accumulates over time. As TD accumulates, service development productivity goes down because developers must then navigate and work around tangled, outdated, or poorly structured ("spaghetti") code (Tom et al. 2013, Besker et al. 2019, 2020). Moreover, as the system grows more complex, even small changes require extra effort to understand dependencies. With decreasing service development rate, new TD generation rate also goes down, closing a balancing feedback loop (*Bad code reducing productivity loop*). TD generation rate also depends on the current level of TD in the codebase. As commonly acknowledged in software development circles, TD tends to compound on itself (Tom et al. 2013, McKinsey & Company 2020, 2023). This is mainly because TD accumulation can lead to a downward spiral of design shortcuts leading to more TD accumulation (Ramasubbu & Kemerer 2016). Thus, as TD increases, it triggers more TD generation, leading to a reinforcing feedback loop (*TD begets TD loop*), a vicious cycle.

AI-assistance in coding can accelerate the TD vicious cycle in two ways: First, AI-coding tools boost up the service development rate (Peng et al. 2023, Cui et al. 2024), increasing the number of services (or in general, the size of the codebase), all else being equal. While, this is beneficial to the firm's immediate revenue stream, higher service development rate adds more to the technical debt. Second and more importantly, AI-coding tools directly accelerate the rate of TD generation (GitClear 2025) due to AI's lack of context awareness (Aslan 2024*b*, Doerrfeld 2025, Gupta 2025). This can manifest as code duplications, dependency conflicts, and poorly structured code. The acceleration of TD generation exacerbates the TD vicious cycle. Moreover, both increasing number of services and accumulating technical debt increases the total maintenance cost for services (Woodard et al. 2013, Ramasubbu & Kemerer 2016, Besker et al. 2019), which hinders firm profitability.

Finally, though not shown in Figure 1 for clarity, there is a balancing effect that can keep the vicious cycle in check: Services become obsolete over time, losing their relevance. As they obsolesce, some of their associated technical debt becomes irrelevant as well (Tom et al. 2013). This reduction in TD helps slow down further accumulation of TD.

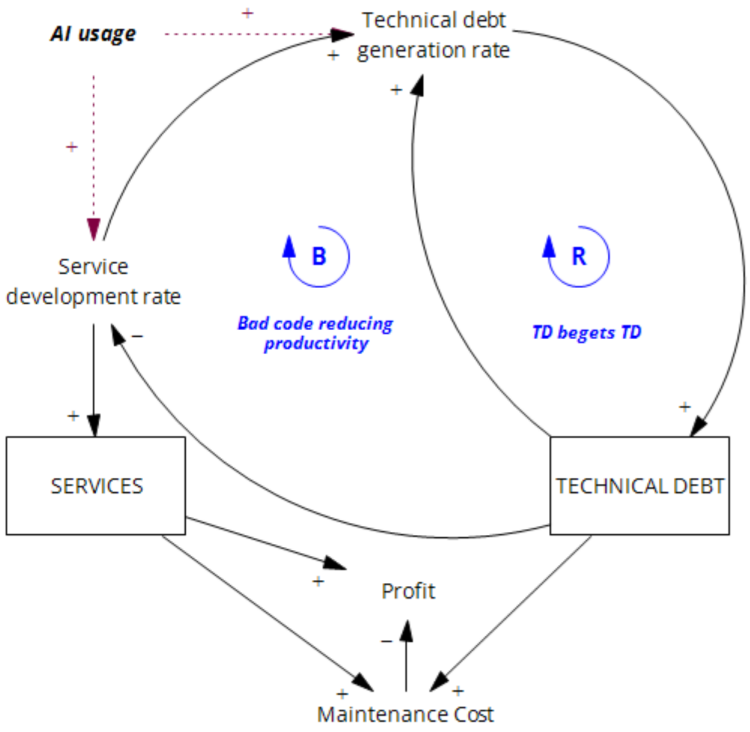


Figure 1: Simplified Causal Loop Diagram

Notes. In a Causal Loop Diagram (CLD), arrows represent causal relationships between variables. A "+" sign on an arrow denotes a positive relationship: when the independent variable increases (or decreases), the dependent variable changes in the same direction, assuming all else remains constant. A "–" sign indicates a negative relationship: when the independent variable increases (or decreases), the dependent variable shifts in the opposite direction. Feedback loops formed by these links can be either reinforcing or balancing. A reinforcing loop amplifies change—an increase (or decrease) in one variable triggers further increases (or decreases), creating a virtuous or vicious cycle. A balancing loop counteracts change—an increase (or decrease) in a variable eventually produces the opposite effect, stabilizing the system.

3.2 Brief Overview of the Model Structure

In this section, we provide a brief overview of the model structure. The notation related to the base model is summarized in Table 1. A summary of key model assumptions is as follows:

Summary of Assumptions

A1) At $t = 0$, number of services is zero (i.e. $S(0) = 0$). Technical debt is also initially zero (i.e. $TD(0) = 0$), unless the firm operates in a brownfield environment.

A2) Total number of developers is constant and normalized to 1. A fraction P_S of developers is assigned to service development and the remaining $P_{TD} = 1 - P_S$ is assigned to technical debt retirement.

A3) Cost of using AI, C_{AI} is normalized to zero. This assumption allows us to isolate the core dynamics of AI-assisted software development without speculating on the evolution of AI-assistance costs over time.

3.2.1 Service Development

We consider a firm that develops software services (e.g. mobile apps, enterprise software, etc.). Service development rate $SDR(t)$ primarily depends on the developer time assigned to service development, P_S and the *effective productivity* of developers at time t, $\tilde{\alpha}_D(t)$.

$$SDR(t) = P_S * \tilde{\alpha}_D(t) \tag{1}$$

Developers have a base productivity of α_D, which can be improved with AI coding tools, as empirically shown in the literature (Peng et al. 2023, Cui et al. 2024). However, the technical debt $TD(t)$ and the resulting complexity of the codebase can slow down development productivity (Tom et al. 2013, Besker et al. 2019, 2020). Thus, effective productivity at time t, $\tilde{\alpha}_D(t)$, is given by:

$$\tilde{\alpha}_D(t) = \alpha_D * (1 + AI(t)) * f_{TD}(t) \tag{2}$$

where $AI(t)$ denotes the full time equivalent (FTE) of productivity boost from AI usage at time t, and $f_{TD}(t)$ is a decreasing function of technical debt to capture the negative effect of technical debt on productivity. The sensitivity of $f_{TD}(t)$ reflects the system's underlying complexity, encompassing both individual service complexity and the emergent complexity generated by their interactions, each of which can amplify the effects of poor programming practices.

Due to obsolescence, services lose their relevance after a period of time represented by τ^S_{obs}. In line with standard system dynamics formulations (e.g., Anderson Jr et al. 2023), we assume that the depreciation amount is proportional to the current number of services $S(t)$, resulting in diminishing returns to software development.

Altogether, the change in the number of services at time t is given by:

$$S'(t) = P_S * \alpha_D * (1 + AI(t)) * f_{TD}(t) - S(t)/\tau^S_{obs} \tag{3}$$

3.2.2 Technical Debt Accumulation

As services are developed, new technical debt is introduced at a fractional rate of $\tilde{\gamma}(t)$, such that increase in TD at time t, $TDR(t)$, is given by $TDR(t) = SDR(t) * \tilde{\gamma}(t)$.

The rate of TD generation depends on several factors. We assume that programmers have a base fraction of TD generation, γ_H. This fraction depends on the programmers' experience and skill level. TD generation rate also depends on the level of AI reliance in coding. There is emerging empirical evidence suggesting that the rise of AI coding tools is associated with increased code duplication and declining code quality (GitClear 2025). Developers we interviewed expressed concerns about code duplications, dependency conflicts, and AI's lack of context awareness, echoing anecdotal reports in software development circles (Aslan 2024*b*, Doerrfeld 2025, Gupta 2025). Thus, we assume that AI usage introduces TD at a fraction of γ_{AI} where $\gamma_{AI} > \gamma_H$ in the base model. Finally, TD generation rate is also influenced by the current level of TD in the code base. Software developers have acknowledged time and again that TD has a way of compounding on itself (Tom et al. 2013, Ramasubbu & Kemerer 2016, McKinsey & Company 2020, 2023). Ramasubbu & Kemerer (2016) explain how TD accumulation can lead to a downward spiral of design shortcuts leading to more TD accumulation. To account for this compounding effect of TD, we assume that TD generation fraction $\tilde{\gamma}$ increases with existing TD. This effect is captured by $g_{TD}(t) = 1 + (\frac{TD(t)}{TD_{ref}})^2$, where TD_{ref} is a scaling factor that decreases in the system's underlying functional complexity. Thus, the effective TD generation rate $\tilde{\gamma}(t)$ is given by:

$$\tilde{\gamma}(t) = (\gamma_H + \gamma_{AI} * AI(t)) * g_{TD}(t) \tag{4}$$

TD can be retired by programmers through refactoring and other code revision efforts (Brown et al. 2010, Besker et al. 2019, Woodard et al. 2013). We assume that a certain amount of developer time, P_{TD}, is allocated to retiring technical debt. In practice, P_{TD} tends to be lower than 20% (McKinsey & Company 2020) of programming time, with 15% suggested as a good allocation (Accenture 2024). We assume that programmers have a base productivity for retiring technical debt, α_{TD}. Thus, the rate of TD retirement is given by $P_{TD} * \alpha_{TD}$.

As services become obsolete over time, certain elements of their associated technical debt naturally become obsolete as well (Tom et al. 2013). However, due to interdependencies within the codebase, some of that debt continues to impact active services and must still be managed. Accordingly, we assume that technical debt obsolesces more slowly compared to services and the disparity in obsolescence times is determined by the level of dependencies in the codebase. Specifically, $\tau_{obs}^{TD} = r \times \tau_{obs}^{S}$ where $r \geq 1$ is a proxy for the codebase dependency level. Finally, similar to service obsolescence, we assume that TD obsolescence amount is proportional to the current level of TD.

Altogether, the change in $TD(t)$ is given by:

$$TD'(t) = SDR(t) * \tilde{\gamma}(t) - P_{TD} * \alpha_{TD} - TD(t)/\tau_{obs}^{TD} \tag{5}$$

where software development rate $SDR(t)$ is defined in equation (1) and $\tilde{\gamma}(t)$ is defined in equation (4).

3.2.3 Service Maintenance

After a software service is built and deployed, it needs to be maintained to keep it running reliably. Software maintenance keeps software operational and in line with changing user needs. The cost of maintenance can be the largest component of a software's total cost of ownership, with some estimates suggesting it can account for 50-90% of the total cost (Bennett et al. 2002).

A code base that is riddled with technical debt is challenging to maintain (Tom et al. 2013, Woodard et al. 2013, Ramasubbu & Kemerer 2016, Besker et al. 2019). This is because when shortcuts, quick fixes, and tangled dependencies ("spaghetti code") accumulate, the system becomes more brittle. In addition, messy, inconsistent, or duplicated code is harder for developers to understand, which slows down maintenance efforts. In short, technical debt can increase the cost of maintenance significantly. Accordingly, we model the maintenance cost per service, $c(t)$, as:

$$c(t) = \bar{c} * (1 + \beta * TD(t)^2) \tag{6}$$

where $\bar{c}$ is the cost of maintaining a service in the absence of technical debt, and β is a scaling factor that increases in the system's underlying functional complexity.[3]

$$C_M(t) = c(t) * S(t) \tag{7}$$

3.2.4 Firm Profit

The firm earns a revenue of p per service per month[4] and incurs the wage cost, the maintenance cost, and the cost of using AI services.

The firm pays software developers a wage of w per month. In the base model, total

[3]We also considered a linear cost function for c(t); i.e. $c(t) = \bar{c} * (1 + \beta * TD(t))$. The results are qualitatively similar.

[4]In the base model, p is constant; however we also tested an extension where p decreases with excessive technical debt to reflect the effect of declining quality of services. Unsurprisingly, this additional effect on revenue makes AI assistance less appealing.

programmer time is constant and normalized to 1. A fraction P_S of that programmer time is allocated to service development and the remaining fraction P_{TD} is allocated to retiring TD; i.e. $P_S + P_{TD} = 1$. Thus, the total wage cost per month is w. The firm incurs the maintenance cost, $C_M(t)$, per month as defined in Section 3.2.3. Finally, C_{AI} is the cost of using AI tools per unit time. We normalize this cost to zero throughout the analysis of the base model. Altogether, the firm's profit at time t is given by:

$$\Pi(t) = p * S(t) - C_M(t) - w - C_{AI} \tag{8}$$

Throughout the analysis, we consider cumulative *discounted* profit as the key performance measure. A high discount rate favors earlier and quicker development, capturing the essence of schedule pressure. This is particularly relevant for startups, where limited runway, investor expectations, and first-mover advantages make early returns proportionally more valuable than long-term gains.

4 Analysis

We built the model using the software package Vensim DSS. We run the simulations for a time horizon of 200 months. Our simulations use the Euler numeric integration method with a time step of 0.25. Note that this time step is less than one quarter of the shortest delay in the model in line with the guidance from the literature (Sterman (2000), pp. 905–908). We tested for integration errors by running the model with different time steps and observed that the dynamic behavior remained the same, showing robustness with respect to the choice of time step.

We conducted various tests for checking model validity based on the guidelines in Barlas (1996) and Sterman (2000). Given the theoretical orientation of our study, validation did not involve "behavior pattern tests" aimed at evaluating the model's ability to replicate real system behavior. Rather, we emphasized structure validity, assessing dimensional consistency, behavioral mode plausibility, and robustness under extreme conditions, and complemented these checks with extensive sensitivity analyses of the model parameters.

In what follows, we first discuss the dynamic behavior of the base setting with parameter values provided in Table 1. Then, we evaluate the impact of AI assistance under various conditions, such as different developer skill level (junior vs. senior), project environment (greenfield vs. brownfield), and market conditions (fast vs. slow-evolving industry). Next, we discuss mitigating the negative effects through certain guardrails, such as allocating more developers to technical debt retirement. Finally, we analyze the optimal level of AI assistance

for an organization. Throughout the analysis, we summarize the most important insights as *Results*.

Table 1: Notation

	Decision Variables
$AI(t)$	AI usage in FTE
	State Variables
$S(t)$	Number of active services at time t
$TD(t)$	Technical debt at time t
	Parameters and Their Base Values
α_D	Base Developer Productivity; base value= 3
α_{TD}	Productivity for TD retirement; base value= 0.5
γ_H	Fraction of TD generation by programmers; base value= 0.001
γ_{AI}	Fraction of TD generation by AI; base value= 0.004
τ_{obs}^{TD}	Technical debt obsolescence time; base value= 60
τ_{obs}^{S}	Service obsolescence time; base value= 12
r	Codebase dependency level; base value= 5
P_{TD}	Allocation of programmer time to TD retirement; base value= 20%
P_S	Allocation of programmer time to service development; base value= 80%
$\bar{c}$	Base maintenance cost per service; base value= 0.001
β	Codebase functional complexity; base value= 0.07
p	Revenue per service per time; base value= 0.1
w	Software developer wage per time; base value= 0.5
	Other Calculated Variables
$SDR(t)$	Software development rate at time t
$C_M(t)$	Cost of maintenance at time t
$f_{TD}(t)$	Effect of TD on software development rate at time t
$g_{TD}(t)$	Effect of TD on TD generation rate at time t

4.1 Dynamic Behavior of the Base Model

When we simulate the base model without any AI assistance ("No AI" in Figure 2), we observe that the number of (relevant) services $S(t)$ rises initially and then reach an equilibrium. This is mainly because the service obsolescence rate is proportional to the number of services, which results in diminishing returns to service development. In addition, the modest rise in technical debt $TD(t)$ has a negative impact on developer productivity, though the effect is immaterial in this case.

If developers boost their productivity slightly with using AI tools ("Low AI" in Figure 2),

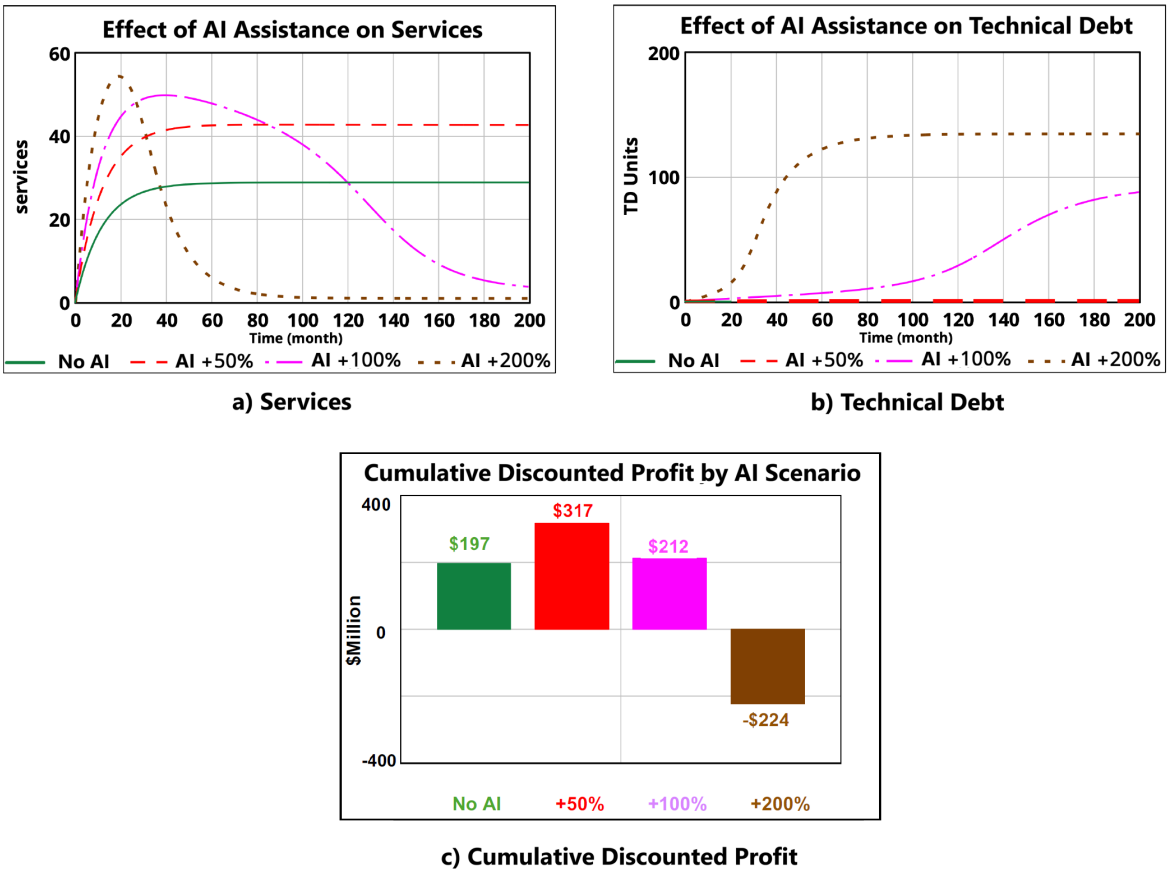


Figure 2: Effect of AI Assistance on Services, Technical Debt, and Discounted Cumulative Profit

the increased productivity leads to a higher number of services at equilibrium compared to the base setting with no AI usage. In this case, technical debt is also higher, since AI assistance in coding results in higher rates of technical debt generation (Aslan 2024*b*, Doerrfeld 2025, Gupta 2025). Still, the negative impact of TD is not significant enough to offset the benefits of using AI assistance. The firm ends up with a higher cumulative discounted profit.

As the firm increases its reliance on AI assistance ("Medium AI" in Figure 2), the dynamics start shifting. In the short term, higher reliance leads to a significant increase in the number of services generated. However, during this "turbocharged" development phase technical debt increases, first gradually and then suddenly. This is because TD compounds on itself if not retired, due to the positive feedback loops discussed in Section 3.1. Once technical debt exceeds a certain threshold, its negative impact on developer productivity becomes acute. Even with AI assistance, developers' service development rate falls because of the challenges introduced by technical debt (e.g. "spaghetti code", code duplications

that create testing challenges, poor documentation). Together with ongoing service obsolescence, the number of relevant services falls down eventually, leading to a revenue collapse. Moreover, with increased technical debt, maintenance costs rise sharply (Tom et al. 2013, Woodard et al. 2013, Ramasubbu & Kemerer 2016, Besker et al. 2019), further eroding long term firm profit. These negative effects are worsened when AI reliance is further increased ("High AI" in Figure 2), leading to a "technical debt bankruptcy" (Tom et al. 2013) where it is no longer feasible or desirable to build atop the existing code base.

These observations suggest the existence of a "sweet spot" for AI assistance in software development, expressed in the following result:

- **Result 1 (AI-assistance sweet spot)**. *While higher reliance on AI assistance in coding initially boosts software development productivity, it can backfire in the long run. There exists a "sweet spot" at which AI assistance improves overall productivity without triggering long-term revenue erosion or collapse.*

It is worth emphasizing that the negative effects of AI assistance would not typically be felt immediately. The trade-off between productivity and scalability is moot in the short run. Thus, if AI assistance is considered for a short term project with minimal expectations for code reuse, the firm is likely to observe net benefits as long as usage costs remain reasonable (Figure 3). For example, a startup developing a demo for a funding round would likely benefit from AI productivity boost without experiencing the long term effects.

- **Result 2 (Shorter horizons call for more AI-assistance)**. *Shorter horizons favor higher levels of AI assistance, because technical debt takes time to build up sufficiently for its negative effects to be felt.*

4.2 Effect of Developer Experience: Junior vs. Senior Developers

Recent IS research demonstrates that an individual's expertise substantially shapes the effectiveness of human–AI collaboration (Figge et al. 2025, Jussupow et al. 2021, Allen & Choudhury 2022, Wang et al. 2024). In software development, emerging empirical evidence suggests that AI assistance yields different performance effects for novice and experienced developers, with greater productivity gains observed among the former (Cui et al. 2024, Peng et al. 2023). These findings have sparked contrasting ideas for workforce management: one view suggests that firms could reduce reliance on costly senior developers by pairing junior developers with AI; another argues that firms might instead limit junior hiring and enhance the productivity of senior developers through AI augmentation. Motivated by these

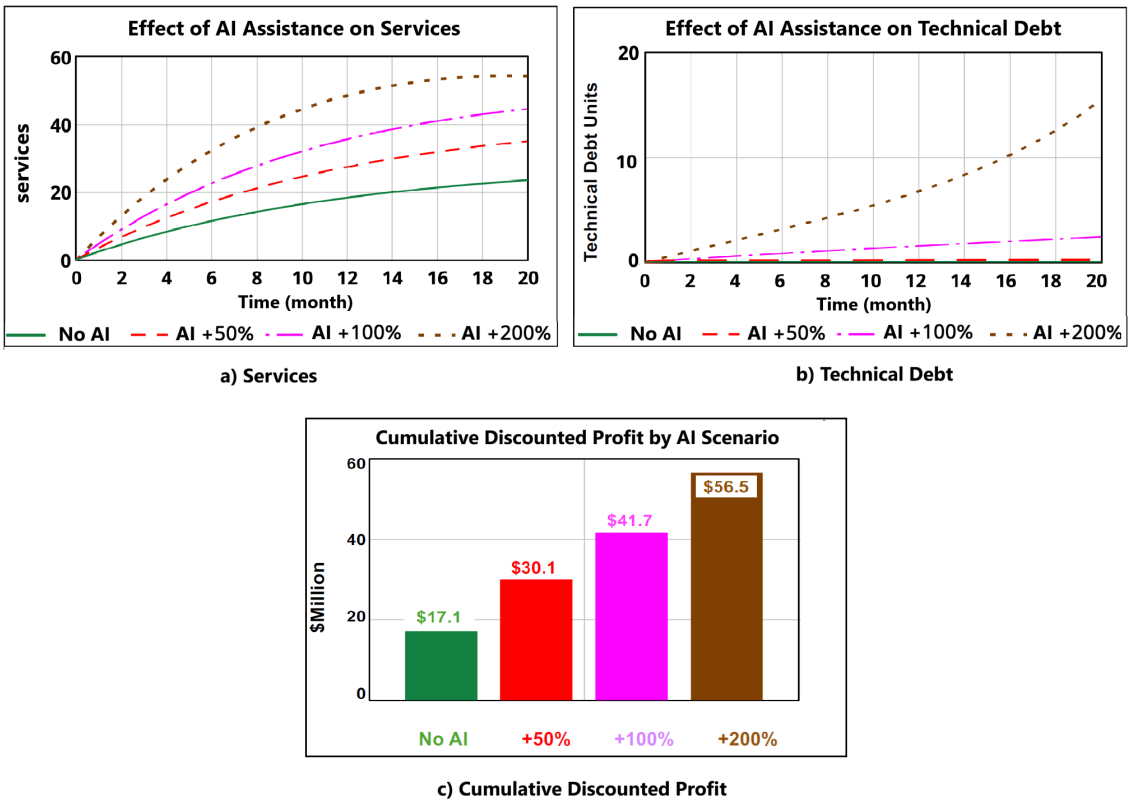


Figure 3: The Effect of AI Assistance in Short Horizons.

divergent strategies and the potential for expertise-driven performance disparities, we explore how developer expertise may affect the long term performance of AI-assisted software development.

Experienced developers ("senior") benefit from more extensive domain knowledge and tend to have a better sense of the "big picture." They are better able to identify architectural weaknesses early and prevent technical debt from propagating. Indeed, MacCormack et al. (2001) find that teams with generational experience are associated with higher software productivity and quality. Ramasubbu & Kemerer (2016) find that more experienced teams are associated with lower risk of system failure due to technical debt, implying that experienced developers have a better handle on technical debt management than novice developers. Accordingly, we assume that "junior" developers have a higher rate of technical debt generation than "senior" ones (i.e. γ_H is higher for junior developers).

Relying too much on AI-assistance may exacerbate this effect. In our interviews with software developers as well as blogs by industry veterans, concerns are raised about downstream effects of AI-coding, particularly in the hands of junior developers. One of our interviewees expressed this concern as: *"[With AI] a junior engineer can write as fast as a senior engineer, but they don't have the cognitive sense of what they're doing... or what problems they're*

causing... or even if it's a good idea to do what they're doing." Another engineer said "*One thing [from AI use] that's emerging is that [junior developers] don't know the context of what they're working on. We now don't get any recommendations from the juniors. Now it's the leads [technical managers] that are getting into the weeds and doing the debugging [because the juniors can't].*"

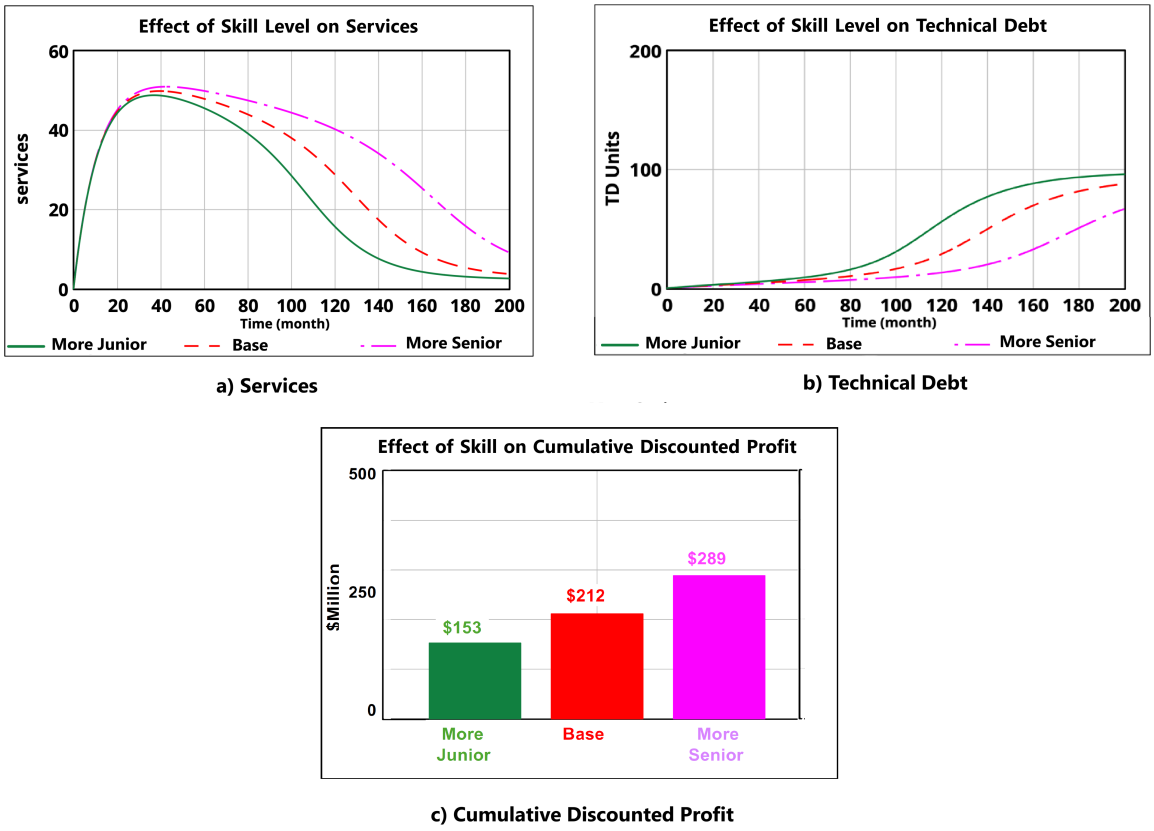


Figure 4: Effect of skill level when junior and senior programmers manage AI equally well. AI assistance for all three runs is +100 percent.

To study this, we first run scenarios that differentiate senior and junior developers by assuming that juniors generate more technical debt per new service created than in the base case and seniors less debt than in the base case. Meanwhile we hold constant their base service-development productivity without AI assistance, α_D, and their effectiveness in managing AI tools. In Figure 4, short-term performance is identical for both groups, but senior developers outperform juniors over time because juniors accumulate technical debt more quickly. As this debt builds, the codebase becomes increasingly burdened, slowing new development and raising maintenance costs. These dynamics produce a pronounced gap in long-run profitability.

The gap between seniors' and juniors' performance widens further if senior developers

are better at managing AI assistance (Figure 5). Specifically, when senior programmers are more effective at controlling the "AI tax" on technical debt—meaning that the amount of AI-generated debt per new service, γ_{AI}, is lower under their oversight than in the base case—the number of services remains at a healthy equilibrium even with substantial AI assistance because technical debt is kept in check. Juniors, however, are worse at managing the "AI tax" than in the base case, resulting in high technical debt and an early crash in long-term productivity.

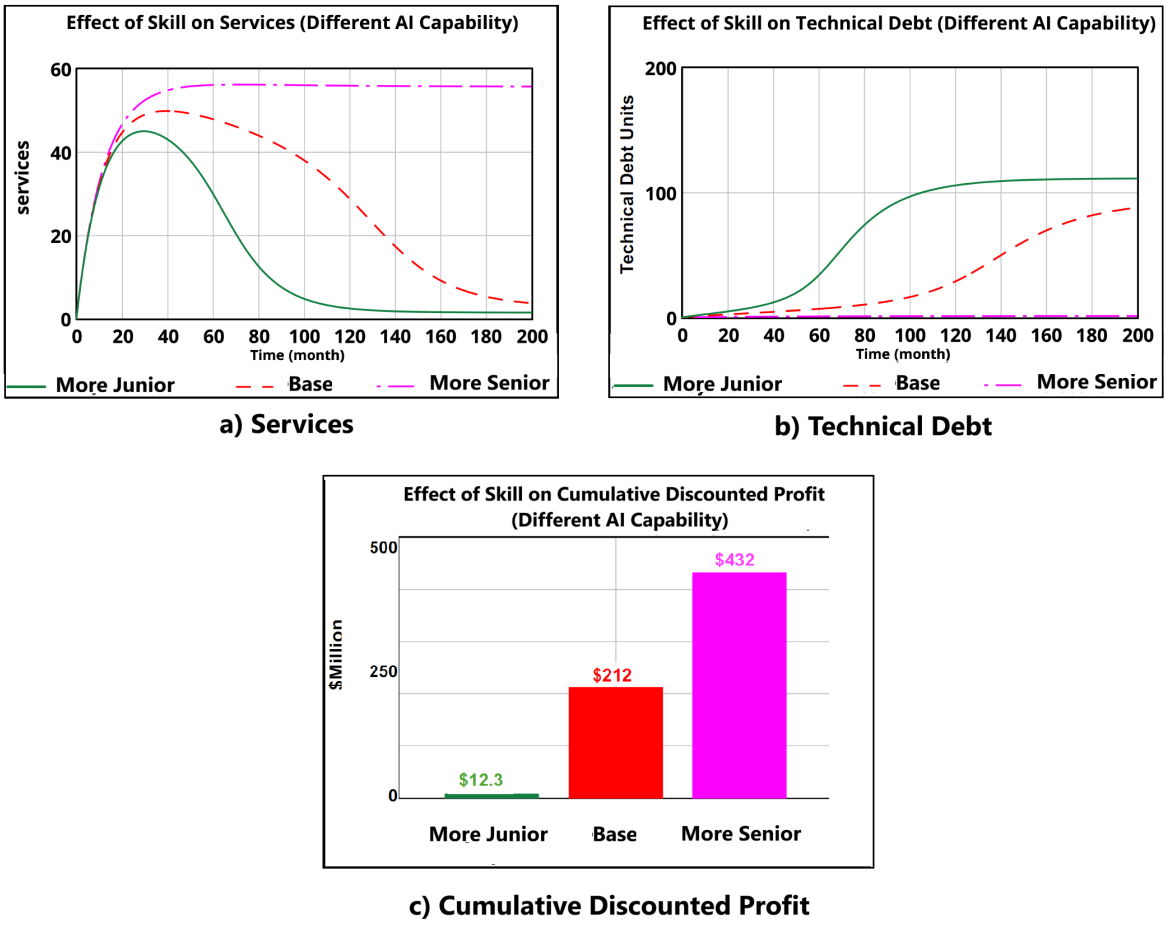


Figure 5: Effect of skill level when senior programmers manage AI better than junior programmers. AI assistance for all three runs is +100 percent.

Overall, Figures 4 and 5 show that boosting junior programmers' productivity with AI assistance can backfire in the long run. Any potential short-term savings from employing less expensive junior developers are likely outweighed by reduced profitability driven by rapidly rising maintenance burdens and the sustained decline in productivity caused by accumulating technical debt in the codebase. Hence, the adoption of AI tools may bias firms toward

employing senior developers rather than junior ones. Emerging labor-market evidence is consistent with this pattern: senior software-developer positions have been expanding in the United States, while junior-level roles have been declining (Brynjolfsson, Chandar & Chen 2025, Korn 2025).

- **Result 3 (AI assistance does not substitute for experience)**. *Reducing reliance on costly senior developers by pairing junior developers with AI can backfire in the long term. The savings in development cost can easily be erased by the compounding costs of maintenance.*

That said, while we highlight how senior developers are better equipped to manage AI-assistance in coding, our results should not be interpreted as a rationale for completely foregoing junior hires. Clearly, such a workforce strategy would cripple long term talent pipelines: Where would one find senior developers in the future if there are no juniors hired today? The key takeaway from our results is that an AI boost that is not backed by context knowledge and experience with the codebase can lead to unfavorable outcomes in the long run. This is especially problematic in "brownfield environments" as we explore next.

4.3 Effect of Programming Environment: Brownfield vs. Greenfield

"In the good old days, COBOL and CICS was most of it. Conditions were pretty controlled. Everything now has lots of different vendors and apps [all interfacing with each other]. If even one little thing updates, you may have a problem. (...) Everything's become brittle vs. 20-30 years ago. "

Whether a developer's new code has to interface with legacy software ("Brownfield") or not ("Greenfield") influences the accumulation of technical debt. In greenfield development environments, the accumulation of technical debt is relatively limited even with the "AI tax", since the project starts from a clean slate. Conversely, when development occurs atop legacy systems, the lack of architectural visibility of AI assisted coding can result in significant downstream issues.

We model a brownfield environment by assuming that there is intractable technical debt that cannot be retired without a significant overhaul. Thus, $TD(0) = TD^o > 0$. In other words, new services are built upon or interface with existing code that suffers from technical debt. As a result, unlike a greenfield environment, the negative impact of AI assistance is felt immediately through lower productivity and sharper rise in technical debt as new services are developed. We also assume that the initial technical debt TD^o does not obsolesce or get

cleaned up unlike the new technical debt. This is to capture the intractable nature of legacy systems, such as the sixty-year old COBOL code in banking systems that no one dares to update (Mims 2024).

Figure 6 shows the impact of AI assistance at different levels of legacy code interactions (represented by the amount of intractable technical debt, from "none" to "high"). We observe that in environments with high levels of legacy code interactions, fewer services are developed from the beginning of the analysis timeline since existing technical debt and complexity immediately lowers developer productivity, even with AI assistance. As technical debt compounds, the new development productivity falls further and eventually cannot catch up with the obsolescence rate, resulting in a decline in the number of relevant services. In contrast, in greenfield environments the firm can take advantage of AI assistance with limited negative repercussions.

- **Result 4 (AI-assisted coding more problematic in brownfield projects)**. *AI assistance in software development becomes more problematic when building atop or interacting with legacy software ("brownfield environments"). In these environments, too much reliance on AI assistance can lead to technical debt bankruptcy or revenue collapse.*

Finally the issues arising from interacting with legacy software also arise when interacting with legacy hardware (i.e. in embedded software development) or when a platform firm develops software that third-party apps build upon. We provide a more comprehensive interpretation of the implications for these environments in Section 5.

4.4 Strategic TD retirement as a guardrail

So far, our results have shown that in certain environments heavy reliance in AI-assistance can lead to ballooning maintenance costs and an eventual revenue collapse due to augmenting technical debt. This observation naturally brings about the question: What if the firm prioritizes TD retirement to mitigate these effects?

Firms notoriously allocate too little time on TD retirement mainly because the pressure to deliver products or features overshadows concerns about the long-term health of the codebase (McKinsey & Company 2020, 2023, Schelfaut & Shukla 2025). The benefits of debt reduction are delayed and hard to quantify, while the costs of postponing features are immediate and visible. Management also tends to underestimate the compounding nature of technical debt, leading to systematic bias toward short-term productivity at the expense of long-term maintainability. As one of our interviewees said: *"Management just doesn't see the exponentialness of it."*

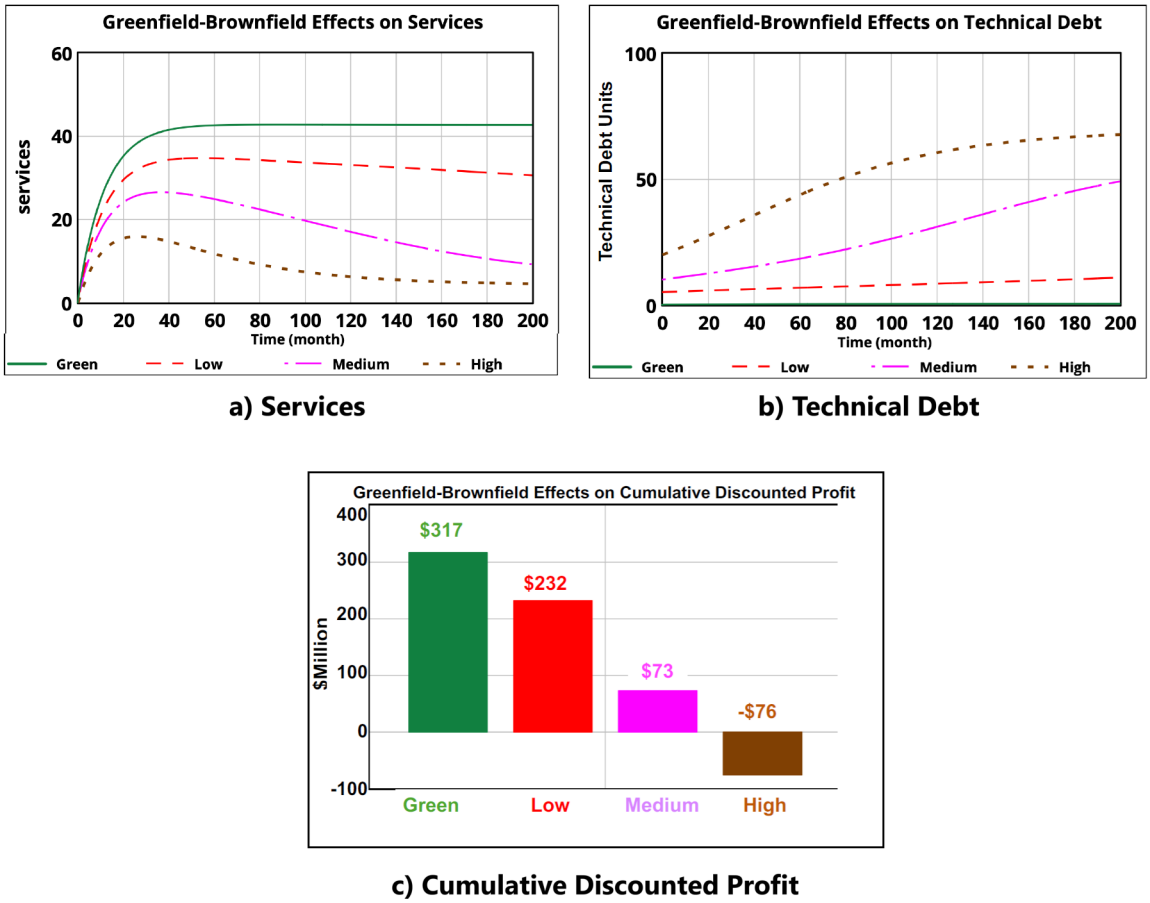


Figure 6: AI assistance is more problematic in brownfield environments (higher levels of interaction with legacy software). Notes: Green = greenfield environment, i.e. no interaction with legacy systems; low = low level of interaction with legacy systems, medium = medium level of interaction with legacy systems, high = high level of interaction with legacy systems.

While CIO's realize that technical debt is increasing, they allocate less than 20 percent of their tech budget for debt retirement (McKinsey & Company 2020). In fact, one of our interviewees admitted that they had to sneak in time for technical debt retirement since management rarely approved time for it: *"This [TD retirement] was never assigned from the top-down. We'd just sneak it in when doing maintenance very quietly and not talk about it. (. . .) No medal for TD retirement."*

In our base model, we assume 20% of total programmer time is allocated to TD retirement. Figure 7 illustrates how changes in this allocation affects the long term performance of AI-assisted software development. We consider allocations ranging from none to 20 %, 40%, and 60%. Note that increasing allocation to debt retirement means less time is allocated for service development, unless the firm hires additional programmers. This is why a very high

allocation to TD (such as 60 %) performs poorly in terms of profitability: With less than half of the programming time allocated to service development, the firm is not able to develop as many revenue generating services. However, zero allocation to TD retirement performs even worse in this case (AI assistance at 100% in all runs). With the accelerated TD accumulation due to AI assistance but no time allocated to retire TD, productivity falls quickly, which results in a revenue collapse. Together with increasing maintenance costs, cumulative discounted profit is hurt significantly. In contrast, when allocation to TD retirement is 40%, the firm is able to sustain a stable stream of services and keep technical debt manageable.

- **Result 5 (Strategic TD retirement as a guardrail)**. *Potential negative effects of AI-assisted coding may be reduced by dedicating a higher fraction of developer time to retiring technical debt.*

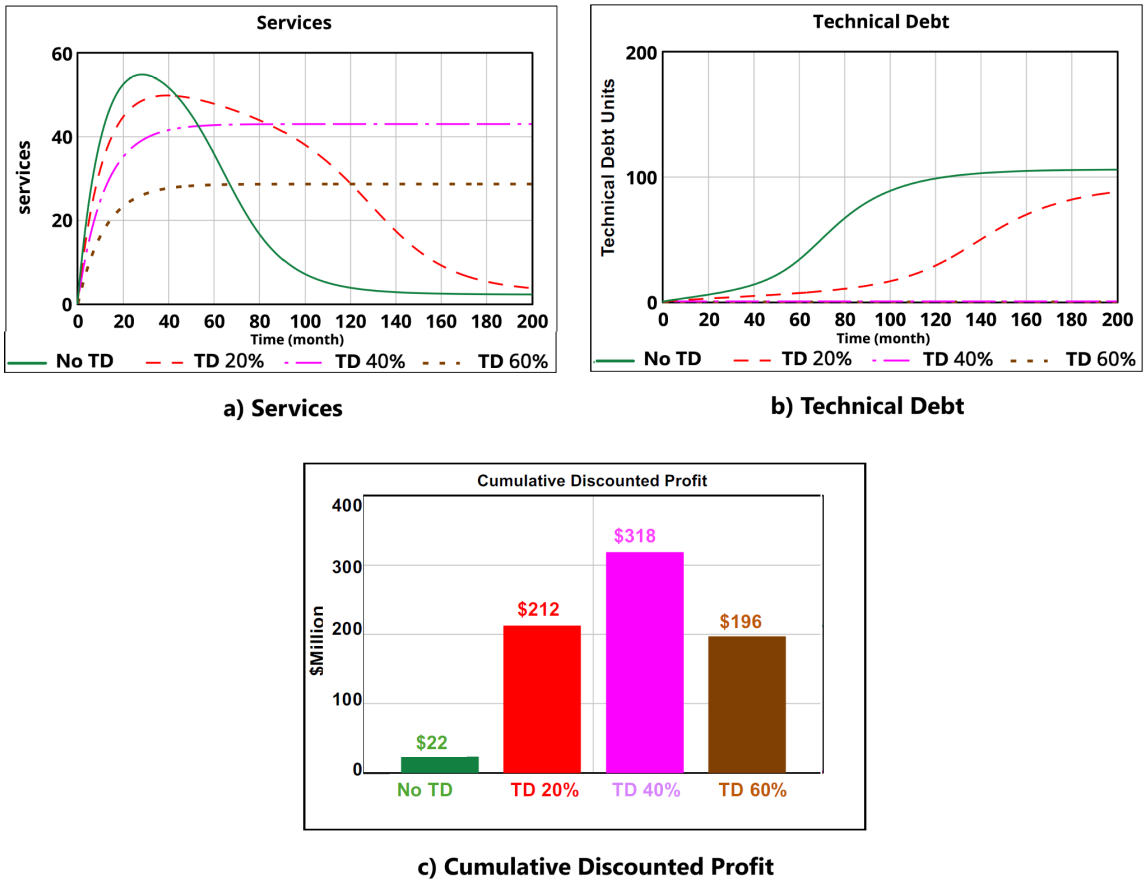


Figure 7: Effect of Increasing Technical Debt Personnel. Technical Debt Retirement Personnel Allocation = 0, 20, 40, 60 percent. AI assistance = +100 percent for all runs.

Finally, we note a potential flip side: While AI-assisted coding is likely to accelerate the growth of technical debt, it may also evolve into a tool for reducing it. Emerging

evidence suggests that AI coding assistants may support maintenance work by identifying duplicated code, improving documentation, and aiding refactoring and code review (Sivori 2024, Bousquette 2025, Dvivedi et al. 2024). In our model, the potential impact of such advancements can be captured by increasing α_{TD} (the productivity of TD retirement efforts). Our numerical analysis suggest that increased debt retirement productivity makes it easier to keep technical debt at bay, reducing the negative effects of AI-assisted coding.

4.5 Market Evolution Speed

In this section we consider how market evolution speed interacts with the AI assistance decision. In fast-evolving markets (such as digital platforms), products obsolesce more quickly. This means, *all else being equal*, time invested in product development has a lower rate of return, which leads to lower profitability. However, in software development, high market evolution speed may have one advantage: as services become obsolete, portions of the technical debt tied to them may also lose relevance. This helps contain the accumulation of technical debt before it escalates into an intractable and costly problem.

To understand the layered effect of market evolution speed, we vary service and TD obsolescence rates (τ_{obs}^{S} and τ_{obs}^{TD}). Figure 8 illustrates the resulting effect on service development, TD accumulation, and profit when programmers have substantial AI assistance (100 % in all runs). We observe that the same level of AI assistance leads to a much higher peak in the number of relevant services in a slow-evolving market compared to a fast-evolving one, because services in the latter become obsolete quickly. However, over time, technical debt accumulation (accelerated by the "AI tax") becomes much more problematic in slow-evolving markets, since debt does not easily become obsolete in these markets. As technical debt rises gradually and then suddenly, development productivity deteriorates, eventually triggering a collapse in the number of relevant services.

In contrast, fast-evolving markets do not accumulate technical debt to the same extent, allowing them to sustain a stable stream of relevant services throughout the horizon. As a result, despite having far fewer relevant services at the beginning, the discounted cumulative profit in these markets is higher than that of very slow-evolving markets in this case.

These observations suggest that in fast-evolving markets, higher levels of AI assistance may be sustainable, since technical debt accumulation is less of an issue.

- **Result 6 (Fast-evolving markets benefit from more AI-assistance)**. *Fast-evolving markets may benefit from higher levels of AI assistance, because technical debt accumulation is contained due to high obsolescence rate.*

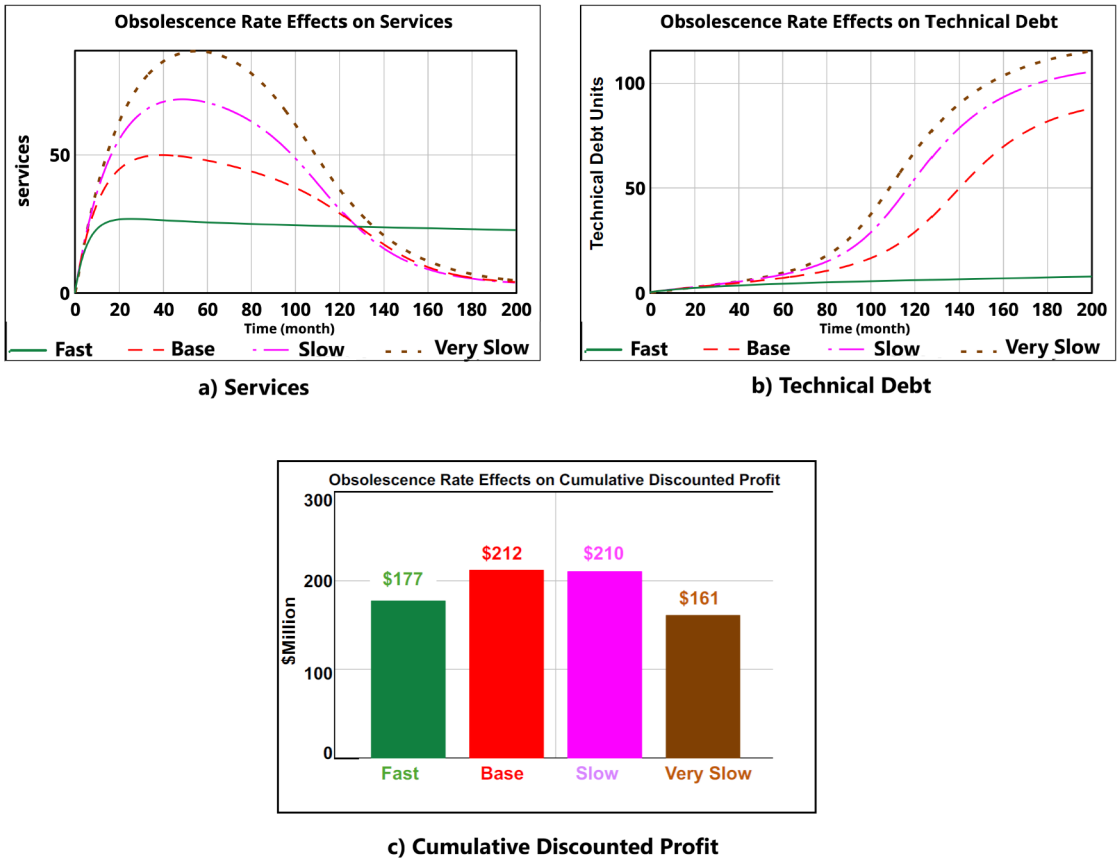


Figure 8: Effect of service and TD obsolescence rate on services, technical debt, and cumulative discounted profit. AI assistance = +100 percent for all runs. Fast evolving markets have lower TD for the same AI assistance level.

4.6 Optimal AI Assistance Strategy

In this section, we explore optimal AI assistance level under different programming environment conditions using the built-in optimizer in Vensim DSS. We consider a finite horizon decision making problem with multiple decision making points throughout the horizon. We extrapolate between these points to smooth the dynamic behavior. We run each optimization with multiple restarts to avoid getting stuck in local optima. In line with a common assumption in the optimal control literature (Holt et al. 1960, Anderson Jr 2001, Anderson et al. 2006, Anderson Jr et al. 2023), the firm's objective function involves a penalty cost to smooth the changes in the decision variables over time. Accordingly, the objective function is calculated as the difference between the cumulative profit and the cumulative penalty cost. The firm maximizes the final value of the objective function.

Figure 9 (a) shows how optimal AI-assistance level changes over time under different levels of legacy code interactions. First, consider the greenfield case (solid, green line) in

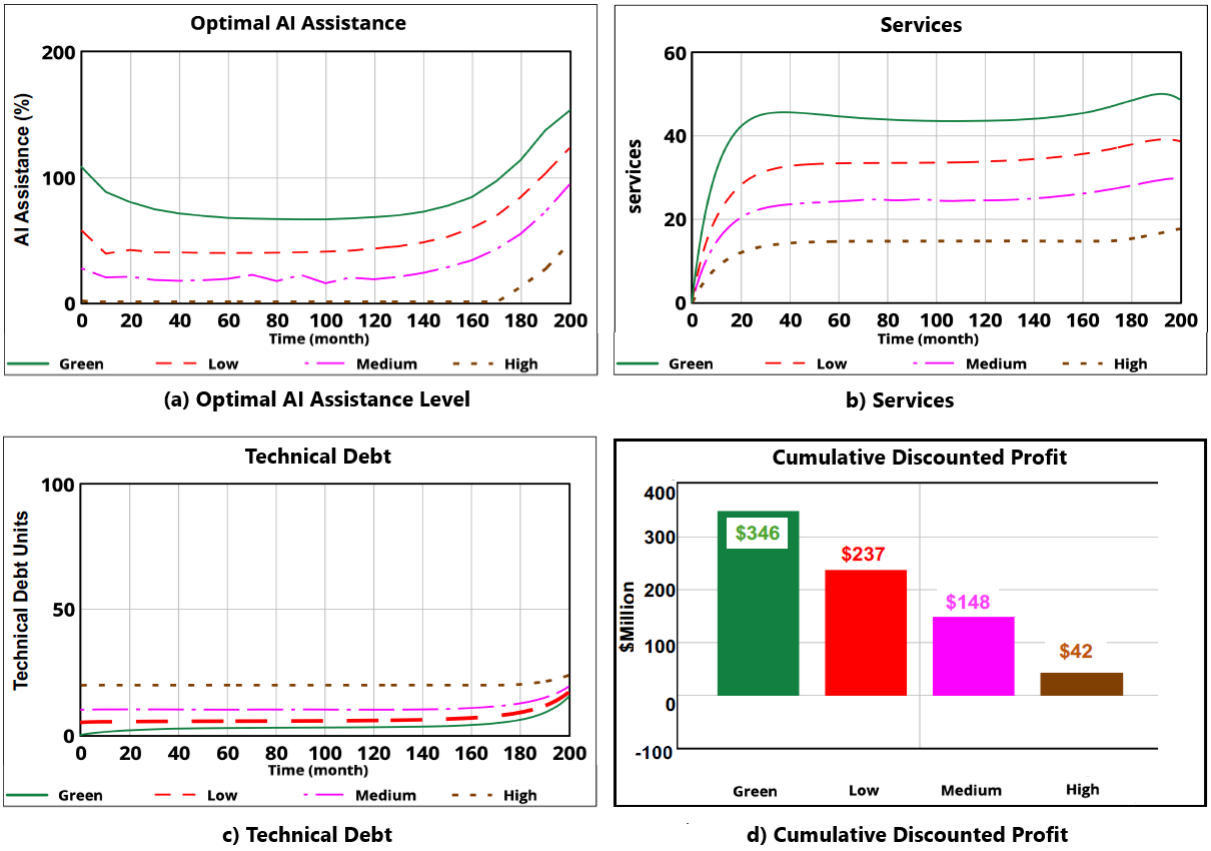


Figure 9: Optimal AI over time in different levels of brownfield environments

Notes: Green = greenfield environment, i.e. no interaction with legacy systems; low = low level of interaction with legacy systems, medium = medium level of interaction with legacy systems, high = high level of interaction with legacy systems.

which no legacy systems are present. At the start of the horizon—when the firm has not yet developed any services—it is optimal to deploy AI assistance aggressively, as no technical debt has accumulated. The resulting productivity boost enables the firm to accelerate service development and commence revenue generation earlier. As the AI-assisted codebase expands, however, technical debt accumulates alongside it. To keep this debt in check, the optimal level of AI usage declines after the initial surge and converges asymptotically toward a lower steady state. Near the end of the horizon, we observe a form of "profit-taking" behavior in which AI usage rises sharply, since the adverse consequences of the additional technical debt will materialize only after the horizon closes. In short, optimal AI usage exhibits a U-shaped trajectory over time, with a relatively flat trough. This pattern is robust across a wide range of parameter values.

We next examine the optimal level of AI assistance in environments with increasing interaction with legacy systems ("brownfields"). The overall behavioral pattern remains

	Dynamic	Static
Green	346	343
Low	237	234
Medium	148	146
High	42	41.8

Table 2: Cumulative Discounted Profit ($M) vs. Optimization Type

Notes. Profit for statically optimized AI only marginally higher than profit for dynamically optimized AI. Green refers to "greenfield" environment with zero legacy code interaction. Low, medium, and high refer to increasing levels of legacy code interactions in the programming environment.

broadly similar to the greenfield case; however, the level of AI assistance at any point in time decreases as legacy-system interaction intensifies – from low, to medium, to high – until, in the high-interaction case, very little AI assistance is used. Moreover, the initial surge in AI usage becomes smaller relative to the eventual equilibrium, because the pre-existing technical debt inherited from legacy systems is sufficiently large that it cannot be allowed to grow as rapidly as in the greenfield setting. For comparable reasons, the end-of-horizon "profit-taking" phase begins later.

At this point, it is useful to consider implications for real-world management of AI assistance for longer-term projects. In practice, the exact parameter values governing any real software system will not be known with sufficient precision to safely implement a fully optimal dynamic strategy. Observing that the optimal dynamic trajectory converges toward an equilibrium, we compare the optimal profit to the profit obtained under a constant level of AI assistance. Table 2 shows that using a constant level of AI assistance has only a modest effect on profit across different levels of legacy-system interaction.

Furthermore, the dynamically optimal strategy keeps technical debt as close as possible to the initial level inherited from the legacy codebase, thereby preventing debt from compounding and producing increasingly harmful effects. These observations suggest a practical managerial approach: begin AI assistance at a conservative level while closely monitoring indicators of technical debt such as sudden changes in the development rate or maintenance costs. AI assistance can then be increased gradually until the first signs of technical-debt pressure appear, at which point AI usage should be reduced to a safe level.

5 Discussion and Limitations

AI-assisted coding tools are rapidly gaining traction, and organizations are expecting substantial productivity improvements as these tools become integrated into development work-

flows (Li 2024, Amodei 2025). Indeed, this is a fast-moving space and we expect that some of the shortcomings we have pointed out, especially the inability of AI systems to "see the bigger picture," will improve over time. But for now, our interviews with software engineers and insights from the trade press suggest a more uneven and context-dependent reality. While experimental studies and "vibecoding" scenarios often report strong performance gains (Peng et al. 2023, Cui et al. 2024, Paradis et al. 2025, Bakal et al. 2025), outcomes are considerably more mixed in complex, long-horizon development environments (Becker et al. 2025, Medappa et al. 2025). Our results help explain this disconnect. We argue that the accumulation of technical debt plays a central role in reducing the long-term benefits of AI assistance, and demonstrate how this mechanism can reconcile the gap between short-term experimental results and real-world development outcomes. Together, these insights underscore the need for more nuanced guidance on when, where, and how AI-coding tools should be strategically deployed.

In particular, short-term, one-off projects in greenfield environments can often be executed safely by non-experts who rely heavily on AI tools ("vibecoding"), especially for rapid data analyses or minimum viable products intended to elicit early customer feedback. By contrast, long-lived projects require close attention to how AI-generated code interacts with its surrounding environment. We find that AI assistance should be reduced under conditions that heighten the risk of technical debt accumulation, such as substantial legacy-system interaction ("brownfields"), greater architectural complexity, limited capacity for debt retirement, slower rates of market change, and—most importantly—lower developer skill.

These factors provide an initial framework for understanding when AI assistance enhances long-term productivity and when it undermines it, but many additional drivers require deeper investigation. Key questions include: How does AI-assistance shift the balance between feature development and quality-assurance work such as code review and technical-debt retirement? How do organizational pressures—especially deadline-driven "rush work"—amplify technical debt accumulation by encouraging teams to skip essential refactoring or higher-level design? And how must team structures and collaboration patterns evolve in response?

These issues could become even more salient as agentic AI assumes a larger share of the development process. When AI agents autonomously generate substantial amounts of code, developers are likely to take on responsibilities that extend far beyond traditional coding. They effectively become tech leads and software architects, responsible for overseeing the accelerated flow of AI-produced code and managing the technical debt generated by a suite of agent "coders" acting on their behalf—agents that perhaps may also, in turn, invoke a second tier of their own additional agents. In this sense, developers may increasingly function as software "supply chain integrators," coordinating and validating code originating

from multiple AI "coders," and ensuring that these components integrate cleanly into the broader system architecture (Parker & Anderson 2002, Anderson et al. 2018). Much more research is needed to understand the implications of this shift for software processes, roles, and governance.

There are also important practical questions regarding which AI tools are best suited to which tasks. Interviewees consistently emphasized that AI assistance is currently most valuable for search-related activities, followed by generating test cases. (Notably, however, larger test suites introduce their own maintenance burdens: tests can themselves become fragile and require ongoing review.) Identifying the next set of "low-hanging-fruit" tasks for AI augmentation is an essential research direction, as is determining how software-engineering education and training should evolve to prepare developers to use these tools effectively.

Another important area for further study is identifying which aspects of brownfield legacy software most strongly influence decisions about when and how to use AI tools. Brownfield environments are often framed as interactions among corporate software systems within a single organization, but other contexts can pose even greater challenges. For example, in multi-sided platforms, hundreds or even thousands of third-party applications must interoperate with the platform's core software, often through convoluted and continually evolving interdependencies (Song et al. 2018, Agarwal & Kapoor 2023). In these settings, using AI tools to develop core platform code can be especially risky: technical debt introduced by AI-generated components may propagate broadly across the ecosystem, disrupting many dependent applications. We further speculate that these risks are amplified in business-to-business platforms, where greater customization on both sides of the market creates more complex and less predictable interactions than in the better-studied business-to-consumer platforms (Anderson, Lopez & Parker 2022). Platform owners therefore need to proceed cautiously when deploying AI in such environments.

A different set of challenges arises when using AI tools in embedded software environments, which are also brownfield settings because new code must interface with legacy hardware. An embedded control software engineer at an electric-vehicle manufacturer explained that his team writes code that interacts directly with specialized microprocessors for real-time control. Given the intricacies of this low-level interfacing, he was skeptical about the current usefulness of AI tools beyond supporting search-related activities. Two constraints make AI-generated code especially problematic for these engineers to use. First, their software must run extremely quickly—to the point that they have to hand-write their software in *assembly language* to avoid the code bloat introduced by higher-level languages such as C++ or Python. For the same reason, the bloat typical of AI-generated code (GitClear 2025, Doerrfeld 2025) is unacceptable; in fact, the team had already discontinued the use of

MATLAB-based auto-code generation because of bloat. Second, stringent functional-safety requirements impose further limits, and the interviewee doubted that current AI tools could reliably meet these safety constraints.

These insights underscore that AI-assisted software development is not a straightforward productivity enhancement problem, but rather a deeply contextual, system-level challenge shaped by architectural complexity, evolving developer roles, and hidden risks such as technical debt and cybersecurity vulnerabilities. We view this paper as one of the first attempts to capture these interdependencies and to lay the foundation for a more nuanced research and managerial agenda.

There are a number of limitations to our study. First, our simulation model has several normalizations made for simplicity. This allows us to explore broad strategies under different programming and market environments. However, the stylized nature of the model limits the ability to fully calibrate for a specific firm or industry. Future research could extend our framework to enable such calibration in order to develop tailored strategies.

Second, we have focused on accelerated technical debt accumulation as the primary mechanism for the mixed performance outcomes of AI-assisted software development. However, we have not explicitly modeled a related and increasingly critical concern: elevated cybersecurity risks (Haque et al. 2025, Ji et al. 2024). AI-generated code can unintentionally embed insecure patterns, rely on outdated libraries (i.e., "dependency drift"), or bypass routine threat modeling, increasing exposure to vulnerabilities that may not be immediately detectable but can lead to costly breaches later. The case of the startup Enrichlead illustrates these risks: its founder claimed 100% of the platform code was generated by an AI assistant (Cursor), yet within days the system was found to suffer "newbie-level" security flaws, forcing a shutdown (Kaminsky 2025). The recent cyberattack on Jaguar Land Rover underscores just how costly security lapses can become: the disruption to production resulted in a quarterly loss of nearly £485 million, with an estimated broader impact on the U.K. economy approaching £1.9 billion (Butt 2025, Leggett 2025). While we do not explicitly model cybersecurity risks, we note that higher technical debt (regardless of its source) is associated with higher cybersecurity risk (Ozkaya et al. 2021, BARR Advisory 2022). By modeling the negative impacts of technical debt on maintenance cost, we indirectly capture some of the broader cybersecurity implications for the firm's long-term performance. Future research could extend the model to explicitly link AI-assisted coding with cybersecurity risk exposure in order to offer specific mitigation strategies.

Third, in our simulations, the number of programmers remain constant; that is, the firm does not hire or fire developers. However, workforce implications of Generative AI have become a prominent concern, raising questions about how firms should shape future

talent strategies, skill composition, and organizational design in AI-assisted development environments. There is a clear need for further research across disciplines to examine how AI assistance may affect hiring and training practices, and long-term workforce planning.

Finally, throughout our analysis we normalized the usage cost of AI-coding tools to zero. This simplification allows us to focus on the underlying dynamics of AI-assisted software development without making assumptions about how the cost of AI assistance might evolve over time. There are mixed expectations regarding the future trajectory of AI costs. While marginal token and compute costs have been declining sharply (Appenzeller 2024), new generation reasoning models tend to generate substantially longer outputs, increasing overall token usage (Mims 2025). In addition, most AI providers currently operate at significant losses, and there is growing consensus that enterprise-grade AI coding tools will become more expensive as firms move toward sustainable pricing models. Future research can explicitly incorporate these uncertain cost trajectories to derive more robust best practices for AI-assisted software development.

6 Conclusion

Generative AI is reshaping software development by offering faster code generation, but its effects extend far beyond short-term productivity. In this paper, we use a system dynamics simulation model to examine the impact of Generative AI coding tools on the accumulation of technical debt and long-term system performance. Drawing on industry cases, interviews, and prior research, we study how outcomes vary depending on project type, developer expertise, market dynamism, and technical debt-management practices. Our findings provide guidance on when and how AI coding assistance should be used to balance faster development with system sustainability.

References

Accenture (2024), 'Build your tech and balance your debt', *Accenture* . Accessed: October 2025.
URL: *https://www.accenture.com/content/dam/accenture/final/accenture-com/document-3/Accenture-Build-Your-Tech-and-Manage-Your-Debt-2024.pdf*

Agarwal, S. & Kapoor, R. (2023), 'Value creation tradeoff in business ecosystems: Leveraging complementarities while managing interdependencies', *Organization Science* **34**(3), 1216–1242.

Allen, R. & Choudhury, P. (2022), 'Algorithm-augmented work and domain experience: The countervailing forces of ability and aversion', *Organization Science* **33**(1), 149–169.

Amodei, D. (2025), 'Anthropic ceo says that in 3 to 6 months, ai will be writing 90% of the code software developers were in charge of', *Business Insider* . Accessed: 2025-11-21.
URL: *https://www.businessinsider.com/anthropic-ceo-ai-90-percent-code-3-to-6-months-2025-3*

Anderson, E. G., Chandrasekaran, A., Davis-Blake, A. & Parker, G. G. (2018), 'Managing distributed product development projects: Integration strategies for time-zone and language barriers', *Information Systems Research* **29**(1), 42–69.

Anderson, E. G., Lopez, J. & Parker, G. G. (2022), 'Leveraging value creation to drive the growth of b2b platforms', *Production and Operations Management* **31**(12), 4501–4514.

Anderson, E. G., Morrice, D. J. & Lundeen, G. (2006), 'Stochastic optimal control for staffing and backlog policies in a two-stage customized service supply chain', *Production and Operations Management* **15**(2), 262–278.

Anderson, E., Keith, D. & Lopez, J. (2022), Opportunities for system dynamics research in operations management for public policy. Working Paper, University of Texas at Austin.

Anderson, E., Parker, G. & Tan, B. (2025), 'The hidden costs of coding with generative ai', *MIT Sloan Management Review* **67**(1), 12–14.

Anderson Jr, E. G. (2001), 'The nonstationary staff-planning problem with business cycle and learning effects', *Management Science* **47**(6), 817–832.

Anderson Jr, E. G., Parker, G. G. & Tan, B. (2023), 'Strategic investments for platform launch and ecosystem growth: A dynamic analysis', *Journal of Management Information Systems* **40**(3), 807–839.

Appenzeller, G. (2024), 'Welcome to llmflation – llm inference cost is going down fast'. Accessed on November 2025.
URL: *https://a16z.com/llmflation-llm-inference-cost/*

Aslan, M. (2024*a*), 'The hidden costs of ai coding assistants: Insights from a senior developer'. Accessed on February 2025.
URL: *https://blog.devgenius.io/the-hidden-costs-of-ai-coding-assistants-insights-from-a-senior-developer-76274fe6b345*

Aslan, M. (2024*b*), 'The hidden costs of ai coding assistants: Insights from a senior developer', Dev Genius (blog). Accessed July 30, 2025.
URL: *https://blog.devgenius.io/the-hidden-costs-of-ai-coding-assistants-insights-from-a-senior-developer-76274fe6b345*

Bakal, G., Dasdan, A., Katz, Y., Kaufman, M. & Levin, G. (2025), 'Experience with github copilot for developer productivity at zoominfo', *arXiv preprint arXiv:2501.13282* .

Banker, R., Liang, Y. & Ramasubbu, N. (2021), 'Technical debt and firm performance', *Management Science* **67**(5), 3174–3194.

Barlas, Y. (1996), 'Formal aspects of model validity and validation in system dynamics', *System Dynamics Review: The Journal of the System Dynamics Society* **12**(3), 183–210.

BARR Advisory (2022), The impact of technical debt on cybersecurity, Technical report. Industry report – BARR Advisory. Accessed from `https://www.barradvisory.com/wp-content/uploads/2022/05/The-Impact-of-Technical-Debt-on-Cybersecurity.pdf` on November 6, 2025.

Becker, J., Rush, N., Barnes, E. & Rein, D. (2025), 'Measuring the impact of early-2025 ai on experienced open-source developer productivity', *arXiv preprint arXiv:2507.09089* .

Bennett, K. H., Rajlich, V. T. & Wilde, N. (2002), Software evolution and the staged model of the software lifecycle, *in* 'Advances in computers', Vol. 56, Elsevier, pp. 1–54.

Berente, N., Gu, B., Recker, J. & Santhanam, R. (2021), 'Managing artificial intelligence.', *MIS quarterly* **45**(3).

Besker, T., Ghanbari, H., Martini, A. & Bosch, J. (2020), 'The influence of technical debt on software developer morale', *Journal of Systems and Software* **167**, 110586.

Besker, T., Martini, A. & Bosch, J. (2019), 'Software developer productivity loss due to technical debt—a replication and extension study examining developers' development work', *Journal of Systems and Software* **156**, 41–61.

Bousquette, I. (2025), 'How morgan stanley tackled one of coding's toughest problems', *The Wall Street Journal* .

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I. et al. (2010), Managing technical debt in software-reliant systems, *in* 'Proceedings of the FSE/SDP workshop on Future of software engineering research', pp. 47–52.

Brynjolfsson, E., Chandar, B. & Chen, R. (2025), Canaries in the coal mine? six facts about the recent employment effects of artificial intelligence. Stanford Digital Economy Lab Working Paper Available at `https://digitaleconomy.stanford.edu/wp-content/uploads/2025/08/Canaries_BrynjolfssonChandarChen.pdf`.

Brynjolfsson, E., Li, D. & Raymond, L. (2025), 'Generative ai at work', *The Quarterly Journal of Economics* **140**(2), 889–942.

Butt, J. (2025), 'Jaguar land rover's cyberattack holds an ominous lesson for british businesses'. Accessed on November 2025.
URL: *https://www.cnbc.com/2025/10/29/jaguar-land-rover-cyberattack-holds-ominous-lesson-for-british-firms.html*

Cámara, J., Troya, J., Burgueño, L. & Vallecillo, A. (2023), 'On the assessment of generative ai in modeling tasks: an experience report with chatgpt and uml', *Software and Systems Modeling* **22**(3), 781–793.

Chen, Z. & Chan, J. (2024), 'Large language model in creative work: The role of collaboration modality and user expertise', *Management Science* **70**(12), 9101–9117.

Coutinho, M., Marques, L., Santos, A., Dahia, M., França, C. & de Souza Santos, R. (2024), The role of generative ai in software development productivity: A pilot case study, *in* 'Proceedings of the 1st ACM International Conference on AI-Powered Software', pp. 131–138.

Cui, Z. K., Demirer, M., Jaffe, S., Musolff, L., Peng, S. & Salz, T. (2024), 'The effects of generative ai on high skilled work: Evidence from three field experiments with software developers', *Available at SSRN 4945566* .

Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C. & Jiang, Z. M. J. (2023), 'Github copilot ai pair programmer: Asset or liability?', *Journal of Systems and Software* **203**, 111734.

Dietvorst, B. J., Simmons, J. P. & Massey, C. (2015), 'Algorithm aversion: people erroneously avoid algorithms after seeing them err.', *Journal of experimental psychology: General* **144**(1), 114.

Doerrfeld, B. (2025), 'How ai generated code compounds technical debt'. Accessed in March 2025.
URL: *https://leaddev.com/technical-direction/how-ai-generated-code-accelerates-technical-debt*

Dvivedi, S. S., Vijay, V., Pujari, S. L. R., Lodh, S. & Kumar, D. (2024), A comparative analysis of large language models for code documentation generation, *in* 'Proceedings of the 1st ACM international conference on AI-powered software', pp. 65–73.

Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S. & Mockus, A. (2002), 'Does code decay? assessing the evidence from change management data', *IEEE transactions on software engineering* **27**(1), 1–12.

Esenduran, G., Gray, J. V. & Tan, B. (2022), 'A dynamic analysis of supply chain risk management and extended payment terms', *Production and Operations Management* **31**(3), 1394–1417.

Figge, P., Anderson, E. & Lewis, K. (2025), 'Ai-human learning systems: Investigating the strategic role of ai for organizational learning', *Strategic Organization* . Forthcoming.
URL: *https://doi.org/10.1177/14761270251385860*

Fügener, A., Grahl, J., Gupta, A. & Ketter, W. (2021), 'Will humans-in-the-loop become borgs? merits and pitfalls of working with ai', *Management Information Systems Quarterly (MISQ)-Vol* **45**.

Fügener, A., Grahl, J., Gupta, A. & Ketter, W. (2022), 'Cognitive challenges in human–artificial intelligence collaboration: Investigating the path toward productive delegation', *Information Systems Research* **33**(2), 678–696.

Ge, R., Zheng, Z., Tian, X. & Liao, L. (2021), 'Human–robot interaction: When investors adjust the usage of robo-advisors in peer-to-peer lending', *Information Systems Research* **32**(3), 774–785.

GitClear (2025), 'AI Copilot Code Quality: Evaluating 2024's Increased Defect Rate via Code Quality Metrics'. Accessed on April 2025.
URL: *https://www.gitclear.com/ai*$_a$*ssistant*$_c$*ode*$_q$*uality*$_2$025$_r$*esearch*

Gupta, R. (2025), 'The hidden cost of genai code: Technical debt in the age of ai assistance'. Accessed on April 2025.
URL: *https://medium.com/mr-dops/the-hidden-cost-of-genai-code-technical-debt-in-the-age-of-ai-assistance-d4b5be2d70de*

Haque, M. A., Siddique, S., Rahman, M. M., Hasan, A. R., Das, L. R., Kamal, M., Sujaee, K., Gupta, K. D. & George, R. (2025), Sok: Exploring hallucinations and security risks

in ai-assisted software development with insights for llm deployment, *in* '2025 Sixth International Conference on Intelligent Data Science Technologies and Applications (IDSTA)', IEEE, pp. 57–64.

Holt, C. C., Modigliani, F., Muth, J. & Simon, H. (1960), *Planning Production, Inventories, and Work Force.*, Prentice-Hall.

Ji, J., Jun, J., Wu, M. & Gelles, R. (2024), 'Cybersecurity risks of ai-generated code', *Center for Security and Emerging Technology, November* .

Jussupow, E., Spohrer, K., Heinzl, A. & Gawlitza, J. (2021), 'Augmenting medical diagnosis decisions? an investigation into physicians' decision-making process with artificial intelligence', *Information Systems Research* **32**(3), 713–735.

Kaminsky, S. (2025), 'The hidden dangers of ai coding'. Accessed on November 2025.
URL: *https://www.kaspersky.com/blog/vibe-coding-2025-risks/54584/*

Kapani, C. (2025), 'What leaders can learn from the millennium bug'. Accessed in November 2025.
URL: *https://leaddev.com/software-quality/what-leaders-learn-from-millennium*

Korn, M. (2025), 'There is now clearer evidence ai is wrecking young americans' job prospects', *The Wall Street Journal* . Accessed 2025-11-21.
URL: *https://www.wsj.com/economy/jobs/ai-entry-level-job-impact-5c687c84*

Kruchten, P., Nord, R. L. & Ozkaya, I. (2012), 'Technical debt: From metaphor to theory and practice', *Ieee software* **29**(6), 18–21.

Leggett, T. (2025), 'Jaguar land rover posts heavy loss after cyber-attack'. Accessed on November 2025.
URL: *https://www.bbc.com/news/articles/ckg1w255gy1o*

Leyden, J. (2025), 'Ai coding assistants amplify deeper cybersecurity risks', *CSO* .
URL: *https://www.csoonline.com/article/4062720/ai-coding-assistants-amplify-deeper-cybersecurity-risks.html*

Li, H. (2024), 'Ai coding tools mean there are no more junior software developers', *Business Insider* . Accessed: 2025-11-21.
URL: *https://www.businessinsider.com/ai-coding-tools-junior-software-developers-microsoft-github-copilot-chatgpt-2024-10*

Li, Z., Avgeriou, P. & Liang, P. (2015), 'A systematic mapping study on technical debt and its management', *Journal of systems and software* **101**, 193–220.

Luo, X., Qin, M. S., Fang, Z. & Qu, Z. (2021), 'Artificial intelligence coaches for sales agents: Caveats and solutions', *Journal of Marketing* **85**(2), 14–32.

MacCormack, A., Verganti, R. & Iansiti, M. (2001), 'Developing products on "internet time": The anatomy of a flexible development process', *Management science* **47**(1), 133–150.

Marks, G. (2025), 'Business tech news: Zuckerberg says ai will replace mid-level engineers soon', *Forbes* .
URL: *https://www.forbes.com/sites/quickerbettertech/2025/01/26/business-tech-news-zuckerberg-says-ai-will-replace-mid-level-engineers-soon/*

McKinsey & Company (2020), 'Tech debt: Reclaiming tech equity', *McKinsey* . Accessed: March 12, 2025.
URL: *https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/tech-debt-reclaiming-tech-equity*

McKinsey & Company (2023), 'Breaking technical debt's vicious cycle to modernize your business', *McKinsey* . Accessed: March 12, 2025.
URL: *https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/breaking-technical-debts-vicious-cycle-to-modernize-your-business*

Medappa, P. K., Tunc, M. M., Vroegindeweij, M., Fransoo, J. C. et al. (2025), 'Ai-assisted programming may decrease the productivity of experienced developers by increasing maintenance burden', *arXiv preprint arXiv:2510.10165* .

Mims, C. (2024), 'The invisible $1.52 trillion problem: Clunky old software', *The Wall Street Journal* .

Mims, C. (2025), 'Cutting-edge ai was supposed to get cheaper. it's more expensive than ever', *The Wall Street Journal* .

Osmani, A. (2024), 'The 70% problem: Hard truths about ai-assisted coding', Substack blog, "Elevate". Accessed May 2025.
URL: *https://addyo.substack.com/p/the-70-problem-hard-truths-about*

Ozkaya, I., Nord, R. & Woody, C. (2021), Examples of technical debt's cybersecurity impact, Technical report. Report by Carnegie Melon University Software Engineering Institute.

Accessed from https://apps.dtic.mil/sti/trecms/pdf/AD1144728.pdf on November 6, 2025.

Paradis, E., Grey, K., Madison, Q., Nam, D., Macvean, A., Meimand, V., Zhang, N., Ferrari-Church, B. & Chandra, S. (2025), How much does ai impact development speed? an enterprise-based randomized controlled trial, *in* '2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)', IEEE, pp. 618–629.

Parker, G. G. & Anderson, E. G. (2002), 'From buyer to integrator: The transformation of the supply-chain manager in the vertically disintegrating firm', *Production and operations management* **11**(1), 75–91.

Peng, S., Kalliamvakou, E., Cihon, P. & Demirer, M. (2023), 'The impact of ai on developer productivity: Evidence from github copilot', *arXiv preprint* .

Queiroz, M., Anand, A. & Baird, A. (2024), 'Manager appraisal of artificial intelligence investments', *Journal of Management Information Systems* **41**(3), 682–707.

Rai, A., Constantinides, P. & Sarker, S. (2019), 'Next generation digital platforms: Toward human-ai hybrids', *MIS quarterly* **43**(1).

Ramasubbu, N. & Kemerer, C. F. (2016), 'Technical debt and the reliability of enterprise software systems: A competing risks analysis', *Management Science* **62**(5), 1487–1510.

Ramasubbu, N. & Kemerer, C. F. (2021), 'Controlling technical debt remediation in outsourced enterprise systems maintenance: An empirical analysis', *Journal of Management Information Systems* **38**(1), 4–28.

Revilla, E., Saenz, M. J., Seifert, M. & Ma, Y. (2023), 'Human–artificial intelligence collaboration in prediction: A field experiment in the retail industry', *Journal of Management Information Systems* **40**(4), 1071–1098.

Rinta-Kahila, T., Penttinen, E. & Lyytinen, K. (2023), 'Getting trapped in technical debt: Sociotechnical analysis of a legacy system's replacement', *Mis Quarterly* **47**(1).

Rolland, K. H., Mathiassen, L. & Rai, A. (2018), 'Managing digital platforms in user organizations: The interactions between digital options and digital debt', *Information systems research* **29**(2), 419–443.

Russo, D. (2024), 'Navigating the complexity of generative ai adoption in software engineering', *ACM Transactions on Software Engineering and Methodology* **33**(5), 1–50.

Schelfaut, K. & Shukla, P. P. (2025), 'How to manage tech debt in the ai era', *MIT Sloan Management Review (Online)* pp. 1–5.

Sivori, A. (2024), 'Will generative ai increase salesforce technical debt?'. Accessed on April 2025.
URL: *https://www.salesforceben.com/will-generative-ai-increase-salesforce-technical-debt/*

Song, P., Xue, L., Rai, A. & Zhang, C. (2018), 'The ecosystem of software platform', *Mis Quarterly* **42**(1), 121–A6.

Sterman, J. D. (2000), *Business dynamics: systems thinking and modeling for a complex world*, number HD30. 2 S7835 2000.

Sterman, J., Oliva, R., Linderman, K. W. & Bendoly, E. (2015), 'System dynamics perspectives and modeling opportunities for research in operations management', *Journal of Operations Management* **39**, 40.

Stiefel, A. (2025), 'The most common security vulnerabilities in ai-generated code', *Endor Labs* .
URL: *https://www.endorlabs.com/learn/the-most-common-security-vulnerabilities-in-ai-generated-code*

Tiwana, A. & Safadi, H. (2024), 'Atrophy in aging systems: Evidence, dynamics, and antidote', *Information Systems Research* **35**(1), 66–86.

Tom, E., Aurum, A. & Vidgen, R. (2013), 'An exploration of technical debt', *Journal of Systems and Software* **86**(6), 1498–1516.

US Department of Transportation (2023), 'DOT penalizes southwest airlines $140 million for 2022 holiday meltdown'. Accessed in May 2025.
URL: *https://www.transportation.gov/briefing-room/dot-penalizes-southwest-airlines-140-million-2022-holiday-meltdown*

Vogel, S. (2024), 'Crowdstrike outage hits us hospitals'. Accessed in May 2025.
URL: *https://www.healthcaredive.com/news/crowdstrike-outage-hits-us-hospitals/721887/*

Wang, W., Gao, G. & Agarwal, R. (2024), 'Friend or foe? teaming between artificial intelligence and workers with variation in experience', *Management Science* **70**(9), 5753–5775.

Wired (2025), 'Vibe coding is coming for engineering jobs', *Wired* . Accessed: 2025-11-21. **URL:** *https://www.wired.com/story/vibe-coding-engineering-apocalypse*

Woodard, C. J., Ramasubbu, N., Tschang, F. T. & Sambamurthy, V. (2013), 'Design capital and design moves: The logic of digital business strategy', *MIS Quarterly* **37**(2), 537–564.

Zanzotto, F. M. (2019), 'Human-in-the-loop artificial intelligence', *Journal of Artificial Intelligence Research* **64**, 243–252.