Impact of Generative Artificial Intelligence on Agile Software Development: A System Dynamics Approach

PERMISSION TO BORROW

In presenting this dissertation as a partial fulfillment of the requirements for an advanced degree from Georgia State University, I agree that the Library of the University shall make it available for inspection and circulation in accordance with its regulations governing materials of this type. I agree that permission to quote from, copy from, or publish this dissertation may be granted by the author or, in her absence, the professor under whose direction it was written or, in his absence, by the Dean of the Robinson College of Business. Such quoting, copying, or publishing must be solely for scholarly purposes and must not involve potential financial gain. It is understood that any copying from or publication of this dissertation that involves potential gain will not be allowed without written permission of the author.

Madhu Kumar Kota

NOTICE TO BORROWERS

All dissertations deposited in the Georgia State University Library must be used only in accordance with the stipulations prescribed by the author in the preceding statement.

The author of this dissertation is:
Madhu Kumar Kota
J. Mack Robinson College of Business
Georgia State University
Atlanta, GA 30302-4015

The director of this dissertation is:

Balasubramaniam Ramesh
J. Mack Robinson College of Business
Georgia State University
Atlanta, GA 30302-4015

Impact of Generative Artificial Intelligence on Agile Software Development:

A System Dynamics Approach

by

Madhu Kumar Kota

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree

Of

Executive Doctorate in Business

In the Robinson College of Business

Of

Georgia State University

GEORGIA STATE UNIVERSITY

ROBINSON COLLEGE OF BUSINESS

2025

ACCEPTANCE

This dissertation was prepared under the direction of the *MADHU KUMAR KOTA* Dissertation Committee. It has been approved and accepted by all members of that committee, and it has been accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Business Administration in the J. Mack Robinson College of Business of Georgia State University.

Richard Phillips, Dean

DISSERTATION COMMITTEE

Dr. Balasubramaniam Ramesh (Chair)
Dr. Lan Cao
Dr. Likoebe Maruping

ACKNOWLEDGEMENTS

I sincerely thank my advisor, Professor Balasubramaniam Ramesh, for his unwavering support, insightful guidance, and continuous encouragement. His foresight in directing me toward research in the emerging field of generative artificial intelligence and providing the necessary resources was instrumental to my scholarly development and the successful completion of this dissertation.

I extend my heartfelt thanks to Professor Lan Cao for serving on my dissertation committee. I am incredibly fortunate to have had her mentorship. She generously devoted countless hours to help me learn, design, and develop a sophisticated system dynamics simulation model based on her pioneering research.

I also sincerely thank Professor Likoebe Maruping for serving on my dissertation committee and providing valuable feedback. I am grateful to Professor Lars Mathiassen, Professor Charles Dhanaraj and Dr. Louis Grabowski for their steadfast support and encouragement throughout the program. Special appreciation goes to my fellow student, Kartikeya Negi, for his collaboration and partnership in our research efforts.

I am deeply indebted to my wife, Prasoona, whose unwavering support, patience, and encouragement made this journey possible. I am also grateful to my two wonderful sons, Druva and Sasi, whose belief in me gave me strength and motivation. As they pursue their own aspirations, I hope they carry with them the belief that no dream is too big and no goal is beyond reach.

This dissertation is dedicated to my brother-in-law, Dr. C. Poornachand (1955–2023), whose early encouragement and belief in the power of education inspired me to pursue higher learning. I dearly wish he were here to witness this day as I receive my doctoral degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Impact of Generative Artificial Intelligence on Agile Software Development:
A System Dynamics Approach

by

Madhu Kumar Kota

May 2025

Chair: Dr. Balasubramaniam Ramesh

Major Academic Unit: Doctor of Business Administration Program

Generative Artificial Intelligence (GenAI) is rapidly transforming software engineering, particularly in agile software development. It promises unprecedented productivity gains for software developers, yet its use in agile software development is unexplored. Existing research primarily examines GenAI's short-term effects on developer productivity but lacks a holistic understanding of its long-term implications on agile teams and projects. To address this gap, my study employs system dynamics modeling approach to analyze the dynamic interactions and feedback loops introduced by GenAI in agile environments. Given the collaborative and iteration-driven nature of agile environments, using GenAI leads to inevitable unintended consequences such as tech debt and defect accumulation, which I studied through the lens of the Technical Debt framework. My dissertation employs a system dynamics (SD) approach to examine how GenAI-driven coding, testing, and documentation affect agile project outcomes such as productivity and software quality over time. Building on a validated agile SD model and guided by interviews with software development professionals and real-world project data, it incorporates how GenAI use induces reinforcing and balancing feedback loops due to rapid code generation and refinement, complexity and defect buildup, and refactoring activities. The findings indicate that while GenAI initially boosts the coding productivity of individual developers, insufficient refinement of generated code can degrade the quality of the software over time. Moreover, it is found that sufficient refinement of generated code and the use of GenAI for testing and documentation can offset these drawbacks, thereby sustaining productivity and quality over the course of the project. This dissertation provides an SD model that contributes to the literature on GenAI's effects on software development and guides individuals and organizations to effectively incorporate GenAI in their agile software projects.

INDEX WORDS: Generative Artificial Intelligence (GenAI), Agile Software Development, System Dynamics Modeling, Technical Debt, Development Productivity, Software Quality, Code Refinement, Refactoring.

I INTRODUCTION

In November 2022, a Generative Artificial Intelligence (GenAI) application named ChatGPT was made publicly available by OpenAI, an US-based AI research organization. ChatGPT quickly gained a lot of traction due to its ability to understand and respond in natural language (Ooi et al., 2023). The promise of productivity and creativity improvements delivered by GenAI, such as in large language models (LLMs), has taken the world by storm (Ozkaya, 2023). It has the potential to change the software development profession more than any other recent technology (Ebert and Louridas, 2023).

With the introduction of GenAI, software development processes and tools have started to transform significantly (Ulfsnes et al., 2024). GenAI tools are quickly adopted in software development organizations for both technical and managerial activities. By leveraging GenAI, software organizations can automate traditional processes and improve productivity. Indeed, recent studies have catalogued numerous GenAI-driven tools automating tasks ranging from code generation to testing, underscoring GenAI's broad influence across the software lifecycle (Bandi et al., 2023). GenAI has the potential to impact various software development tasks, including requirements elicitation, user experience design, software development, coding, testing, and delivery.

Recent studies have shown that GenAI-enabled software development tools can greatly boost developer productivity by simplifying and automating various coding tasks (Peng et al., 2023). However, the use of generated code is not without challenges; it can lead to increased complexity and reduced software quality, as generated code, while syntactically correct, may be difficult to interpret and maintain (Ebert & Louridas, 2023). Existing research has largely overlooked these short and long-term effects on software development.

Although many recent publications have published studies on the use of GenAI in software engineering practice, empirical evidence on the current state of its development, adoption, benefits, and challenges is limited (Nguyen-Duc et al., 2023). There is no clear understanding of the impact of GenAI tools in software engineering practice. The broader implications of GenAI on overall software development practices remain largely unexplored. Moreover, there is a gap in understanding how GenAI tools influence project and team-level outcomes, particularly in the context of agile software development projects that are characterized by dynamic environments. Therefore, a deeper understanding of how GenAI-enabled software development affects agile practices to influence project outcomes over time is needed. Answering this research gap will shed light on the real-world implications of using GenAI, especially its impact on productivity and software quality in agile software development projects. Therefore, I pose the following research question:

How does generative artificial intelligence impact agile software development outcomes (namely, development productivity and software quality) dynamically over time?

This study defines development productivity as the number of development tasks a developer completes in a given time, measured as story points per person per day. Software quality, as used in this study, refers to the structural and functional soundness of the codebase, including its clarity, consistency, maintainability, and defect-proneness. It captures how well the code supports reliable execution, ease of future modification, and alignment with development standards - factors that are directly affected by code complexity, error rates, and adherence to clean coding practices. Given the scarcity of longitudinal data on the effectiveness of GenAI in real-world software development projects, prior research has primarily taken a static, developer-level approach to assess GenAI's impact on productivity at the level of software development tasks (such as coding, test case generation, etc.). However, to truly understand the impact of GenAI on agile software projects, a more comprehensive approach is necessary. I employ system dynamics modeling (Coyle, 1997), a methodology particularly well-suited to capturing the complex interactions and feedback loops inherent in real-world systems. System dynamics has been widely used to model and analyze the dynamic behavior of software development practices (Abdel-Hamid, 1984), including those in agile environments (Cao et al., 2010). This approach allows for a comprehensive examination of how GenAI influences agile practices, offering valuable insights into their dynamic impact on the productivity of agile software development.

Building on this approach, I develop a system dynamics model that integrates findings from existing literature to capture the impact of GenAI on agile projects. To develop and validate the proposed model, I collected both quantitative and qualitative data from software professionals working on real-world agile projects. Qualitative data from interviews is used to develop and validate the dynamic hypotheses and feedback mechanisms within the model, while quantitative data captures the strength of these validated relationships. I then use system dynamics simulations with the validated model to explore the short and long-term impacts of GenAI on agile software development projects.

Through this study, I aim to provide a deeper understanding of how GenAI influences agile software development and offer valuable insights and guidance for teams and organizations seeking to improve project performance using GenAI tools.

II Background

While the promise of artificial intelligence in code generation has been recognized for quite some time, much progress in this research area is picked up only recently since the release of ChatGPT in 2022. Though some studies have explored the application of generative pre-training transformer (GPT) tools for code generation, GenAI was not the focus for research in Software Engineering until 2020 (Pakalapati et al., 2023). The utilization of GenAI tools across all phases of software development is not thoroughly explored (Rajbhoj et al., 2024).

The recent improvements in the performance of LLMs and release of GenAI tools such as GitHub Copilot and ChatGPT, research interest has spiked across several disciplines (Nguyen-Duc et al., 2023). Several papers are recently published on self-archiving repositories such as arXiv5 and PaperWithCode. Though there aren't many studies published in peer-reviewed academic journals, many reports on the use of GenAI in software development have appeared in practitioner journals and conferences.

II.1 Literature Review

A literature search is conducted in two areas of concern: Generative Artificial Intelligence and Agile Software Development. To search for the literature related to these two areas, I selected six search engines (Google Scholar, arXiv, Elsevier Science Direct, IEEE Xplore Digital Library, ACM Digital Library, and Elicit). These search engines help me to find research papers published in journals, archives, conferences, and workshops, and provide access to preprints (Huang et al., 2024).

I grouped the existing literature into three main themes (i) literature on the future of software development using GenAI detailing use cases, challenges, and limitations, (ii) literature on developer experience of GenAI that report information such as productivity metrics, code quality and usability of LLM tools, and (iii) literature on adoption and acceptance of GenAI with a focus on trust, beliefs and behaviors of software developers.

Primarily the literature published in academic journals is exploratory and does not offer a comprehensive assessment of the impacts of GenAI on the interdependencies and complexities involved in agile software development. For instance, a recent study on integrating GenAI into agile methods highlighted potential benefits but provided no empirical evaluation of project-level outcomes (Bahi et al., 2024). Thus, a clear gap exists in empirically examining how GenAI affects the dynamic interplay of factors in agile projects. Ulfsnes (Ulfsnes et al., 2024) describes six use cases of GenAI in software development, Nguyen-Duc (Nguyen-Duc et al., 2023) describes eleven use cases, Huang (Huang et al., 2024) studied seven use cases, Kokol (Kokol, 2024) discussed five use cases, and Sauvola (Sauvola et al., 2024) four use cases. Some of the GenAI use cases and corresponding publications are listed in Table 1. Among them are a few common themes and use cases that are of interest to my study. The common use cases of GenAI for software development are in the following phases in software development: (1) requirements engineering, (2) design specifications, (3) software implementation and coding, (4) quality assurance and testing, and (5) software maintenance and evolution. Though these studies describe the use of GenAI clearly, the assessment of its impact is not measured and articulated well. My research will fill this gap by investigating how GenAI impacts short-term and long-term outcomes such as productivity and quality in these areas.

The use of GenAI has expanded beyond code generation, to all aspects of the software development lifecycle (Sauvola et al., 2024). For example, GenAI tools like CodeRabbit can automate code reviews by identifying errors and enforcing coding standards, thus streamlining the review process. Tools like TestGPT can create comprehensive test-case suites, requiring minimal manual input (Hnatushenko & Pavlenko, 2024). Tools such as GitHub Copilot can also aid in documentation, thus improving code maintainability (Dhyani et al., 2024). Additionally, automated refactoring and code optimization can enhance code readability and performance.

II.2 Generative AI in Software Development

Generative Artificial Intelligence (GenAI) is a subfield of artificial intelligence capable of generating new content including text, image, audio or video using deep learning models, in response to prompts (Nguyen-Duc et al., 2023). These models can learn the patterns and structure from training data and generate new content that is similar to the content used in training (Bandi et al., 2023).

Text generation using LLMs for requirements elicitation, design specifications, coding, and testing are of primary interest in this study. Though automated code generation is not new, recent advances in LLMs such as ChatGPT have led to a breakthrough in automated code generation (Vaithilingam et al., 2022). Generative AI tools like ChatGPT, GitHub Copilot and Google's Gemini are fluent in programming languages like Java, Python, JavaScript and C++, and can generate code for given prompts. Evidence from existing (laboratory) experiments generally suggests a significant improvement in the productivity of software developers who are using GenAI tools (Cui et al., 2024).

Table 1. GenAI use-cases in software development

GenAI Use-Case	Description	Relevant paper
Requirements Engineering	Leveraging GenAI to gather, analyze, and refine software requirements. GenAI can aid product owners and customers in creating detailed software requirements from high-level goals	White et al., 2024; Arora et al., 2024; Cheng et al., 2024
Design Specifications	Utilizing GenAI to automate the generation and refinement software design and architectural specifications.	Sauvola et al., 2024; Ebert & Louridas, 2023
Coding	Applying GenAI to automate code generation, code completion, and code optimization tasks with the help of natural language prompts.	Nguyen-Duc et al., 2023; Huang et al., 2024
Quality Assurance and Testing	Using GenAI to generate and validate test cases, assist in defect prediction, and refactoring.	Hnatushenko & Pavlenko, 2024; Bajaj & Samal, 2024
Software Maintenance and Evolution	Assisting in maintenance tasks like code updates, code summarization and writing documentation, patch generation, and legacy management.	Nguyen-Duc et al., 2023; Rajbhoj et al., 2024

Code-completion systems in the integrated development environment (IDE) that suggest code snippets to developers while they are coding are the most commonly used GenAI tools (Vaithilingam et al., 2022). To generate snippets of code, they use a large language model that predicts what the user might type next and present code suggestions (the completion). Developers can also ask the tool within the context of what they are currently working on to generate code snippets (the prompt). The benefits of generating software code are extremely high, however, assessing the impact of code generation is a challenge (Vaithilingam et al., 2022).

Recent studies show that using GenAI tools can significantly enhance developer productivity, particularly for repetitive tasks and during the initial stages of software development, where productivity gains of 20–50% have been observed (Peng et al., 2023). Studies have also highlighted the reduction in overall development costs from using GenAI tools (Chui et al., 2023). However, prior literature also highlights potential negative consequences of using GenAI tools such as an increase in the complexity of the codebase from overusing generated code (Idrisov & Shippe, 2024), and an increase in the dependence on GenAI tools over time (Lim et al., 2023). The dynamic interplay between such positive and negative forces can have undiscovered consequences for the long-term impact on software projects, thus requiring further investigation.

GenAI is increasingly used in the following software development tasks. (Huang et al., 2024):

Requirements engineering: Once high-level requirements are understood, a product owner can generate detailed requirements using appropriate prompts. This helps product owners to discover the requirements of a software system interactively and capture any missing details. The generated output combines the initial high-level requirements fed by the product owner and any new or missing requirements it discovered to fill gaps in requirements. This helps in ensuring the completeness of software requirements and accelerates the requirements engineering process (White J et al., 2024).

Code generation: In the software development process, GenAI tools are commonly used for code generation by software developers. The intent is to generate software code using requirements documented in natural language. This provides multiple advantages, including automating and accelerating development tasks. GenAI driven code generation can help developers produce functional code quickly; however, it may still require review and refinement to meet project standards and to handle edge cases. (Nguyen-Duc et al., 2023).

Code summarization: During software development and maintenance phases, developers spend a portion of time to understand and comprehend existing codebase (Xia et al., 2017). Code summarization is the process of generating a natural language description of code in a software system. It involves comprehending the codebase and generating descriptions directly from source code. Code summarization enhances developers' understanding of the code, thereby improving developer productivity. Developers often fail to provide clear comments in their code, due to tight project schedules and heavy workloads. GenAI tools can help automatically generate and inject comments in code, thus making code summarization effective (Huang et al., 2024).

Test case generation: GenAI tools can be used to automatically create test cases and testing scripts based on software requirements or existing code. By doing so, they reduce the manual effort required in writing tests and help in identifying edge cases that developers might overlook. Automating test generation can improve the robustness of the software by ensuring a wider coverage of scenarios (Bajaj & Samal, 2023). However, concerns have been raised regarding the possible drawbacks of these tools, including higher complexity of generated code compared to human-written code (Idrisov & Schlippe, 2024). In other words, large volumes of generated code can increase the overall complexity of the codebase (Idrisov & Schlippe, 2024), decreasing its overall quality (Imai, 2022). Generated code may not always adhere to project-specific coding standards, provide uniformity, and guarantee correctness (Nguyen-Duc et al., 2023).

Patch generation: In software development, code patches are small sets of code changes or fixes that resolve bugs or vulnerabilities in software. Traditionally, software patches are created by developers by identifying the root cause of a bug and then writing code to correct it. Generative

AI automates parts of this process by analyzing the code, identifying problematic sections, suggesting fixes, and generating a patch with the corrected code. GenAI models used for patch generation include OpenAI’s Codex, DeepMind's AlphaCode, and Google's CodeBERT. These models are trained on large codebases and can be fine-tuned on specific languages or types of code to improve their performance in specific contexts.

Code optimization: Software code optimization is the process of improving the runtime behavior of a program by modifying the source code to work more efficiently, without changing its functionality. This can involve reducing execution time, memory usage, or other resource consumption, making the code faster, more scalable, and better suited for its operating environment. GenAI can assist with code optimization by automating parts of the process, providing insights into performance improvements, and suggesting optimized code. A few ways GenAI could help optimize code with functionalities like automated code refactoring, algorithm selection, code profiling and bottleneck identification, and automatic test generation, etc.

Code translation: Code translation is the process of converting software code written in one programming language to a different programming language without changing its functional behavior. This is often done to make the code compatible with different platforms, improve maintainability, upgrade or migrate to newer technologies, or leverage performance benefits. Code translation can range from relatively straightforward syntax changes to more complex structural adaptations, depending on the similarities and differences between the languages. GenAI models such as OpenAI's Codex, DeepMind’s AlphaCode, and other LLMs fine-tuned on codebases can facilitate code translation in several ways, including automated code conversion, cross-language API and syntax mapping, error detection and debugging. Code translation is a challenging task as syntax differences across programming languages can lead to inconsistent program outcomes.

GenAI can make code translation faster, efficient, and more consistent. It helps developers overcome the challenges associated with language-specific syntaxes, libraries, and idioms, enabling smoother upgrades, migrations and modernizations across platforms (Weisz et al., 2022).

II.3 Agile Software Development

Agile software development is an iterative approach to creating software that emphasizes collaboration, adaptability, and rapid delivery of working software products (Bahi et al., 2024). Agile methodology provides structured frameworks for planning, executing, and continuously improving the software delivery process. Agile development methods have gained substantial popularity for software development in nearly all domains.

The Agile approach contrasts with traditional "waterfall" development, which involves a linear sequence of requirements gathering, design, development, testing, and deployment phases. Agile is better suited for projects with frequently and rapidly changing requirements. In agile methodology, documentation and plans are deliberately kept to a minimum to focus the development effort on delivering working software. The agile development methodology has been a great interest to practitioners across multiple fields, including the information systems development. Agility is closely associated with flexibility, collaboration, leanness, and continuous delivery (Conboy, 2009). Table 2 presents an overview of the key agile practices that form the foundation upon which agile frameworks like Scrum and XP are built.

Table 2. A description of key agile practices (adapted from Glaiel et al., 2014)

Agile Practice	Description	Citation
Story/feature driven development	System is broken down into client-valued "features" or "stories," allowing early deployment, demonstrations, and feedback.	Cusumano & Smith, 1995
Iterative and incremental development	Development happens in short, iterative cycles, with support for changing requirements of the client.	Larman & Basili, 2003
Adaptive decision-making	Autonomy for the teams in determining the methods for achieving project objectives. Small and frequent decisions and process improvements based on assessment of prior outcomes.	Maruping et al., 2009; Meso & Jain, 2006
Refactoring	Code is regularly restructured to maintain quality, reduce technical debt, and support scalable and maintainable software architecture.	Cao et al., 2010; Fojtik, 2011
Continuous Testing and Code Integration	Newly written code is continuously tested, allowing for early detection of issues, and is integrated into the software codebase frequently.	Cusumano & Smith, 1995
Dynamic Teams	Collaborative practices like pair programming and daily stand-ups are implemented to enhance team agility and productivity.	Wood et al., 2013

High Customer Involvement	Active customer participation is continuous, allowing for evolving requirements and aligning development with user expectations.	Beck et al., 2001; Cao et al., 2010

The agile software development context, with its iterative cycles, rapid delivery, incremental value creation (multiple releases), and refactoring activities provides a particularly interesting landscape to study the impact of GenAI. On one hand, Agile's emphasis on quick delivery of working code and feedback cycles can amplify the immediate productivity gains that GenAI offers. On the other hand, the same intense focus on rapid delivery combined with lightweight documentation and frequent changes can lead to unforeseen consequences. While GenAI helps speeding up the time to market it also drives reallocation of development effort from new feature development to rework and refactoring of generated code.

II.4 System Dynamics for Modeling Information Systems

System dynamics (SD) is a robust method for modeling and analyzing the interactions that evolve within complex systems over time (Coyle, 1997). It has been extensively applied to understand the dynamics of software development practices. Early foundational work used system dynamics to model the software development process following the waterfall model (Abdel-Hamid, 1984). Building on this foundation, subsequent research expanded the application of system dynamics to agile software development, integrating various subsystems such as human resource management and change management into a cohesive framework (Cao et al., 2010). This model captured the intricate interplay between human and technical components within real-world agile software development projects, providing insights into how these dynamics influence outcomes such as team productivity, software quality, and development costs.

SD is effective for modeling software development since it comprises of complex interdependencies between components and feedback characteristics. Over the years, SD has been used to model the various facets of software development, such as requirements management (Höst et al., 2001) and software testing (Sahaf et al., 2014) as it provides a comprehensive modeling framework to capture both time-delayed feedback and complex causal relationships present in these systems.

Table 3. IS research incorporating SD methodology (adapted from Fang et al., 2018)

Research Area	**Research Topic**	**Focus of Study**	**Relevant paper**
IS development	Software development - quality assurance (QA) activities	Impact of QA activities on the project cost and schedule	Abdel-Hamid, 1989
	Software development	Impacts of software development management decisions over time. Collaborative requirements Engineering	Rodrigues & Williams, 1997; Stallinger & Grünbacher 2001
	IS development	Social and organizational dynamics that cause the success/failure of ISD	Luna-Reyes et al., 2005
	Agile software development	Dynamic impact of agile practices on software development outcomes	Cao et al., 2010
IS organization and strategy	Data warehouses (DW)	Impact of management policies on DW development	Quaddus & Intrapirot, 2001

	ERP systems	Dynamic impact of critical factors on ERP success and failure	Akkermans & van Helden, 2002
	Knowledge management systems	Reasons for KMS initiative failures	Gallivan et al., 2003
	New technology - CT scanner	Impact of implementing new technology on occupational disruption and delay in realizing benefits	Black et al., 2004
	Knowledge systems	Dynamic challenges organizations face in harnessing knowledge	Garud & Kumaraswamy, 2005
	E-commerce	Impact of e-commerce resource endowment on firm performance over time	Fang et al., 2018
Economics of IS and IT	Network service provision	Impact of business processes on provisioning network services.	Dutta, 2001
	IT offshoring	Mechanics impacting growth in IT offshoring	Dutta and Roy, 2005

SD is particularly appropriate for modeling agile software development due to its capability to capture the complex and dynamic interactions inherent in agile practices (Cao et al., 2010). Agile software development involves numerous interconnected practices that influence each other, such as continuous customer feedback, iterative progress, and adaptive planning. These interactions often involve feedback loops, time delays, and non-linear cause-and-effect relationships that are difficult to represent using static models. SD, with its emphasis on feedback systems and dynamic behaviors, provides a robust framework for modeling these agile interactions, allowing for a more holistic analysis of how individual practices affect the overall development process and project outcomes (Cao et al., 2010).

The SD model developed by Cao (Cao et al., 2010) integrates key agile practices such as customer involvement, change management, agile planning and control, refactoring, and pair programming into a cohesive system. The model is grounded in primary data collected from organizations employing agile methods (Scrum and XP) and captures the interplay between these practices and their impact on project performance. For example, the model captures how continuous customer involvement fosters rapid feedback and adjustments, which in turn influence development productivity and software quality. The change management subsystem of the model reflects how adaptive and corrective changes are dynamically managed and their effects on project scope, cost, and schedule. This work also presents insights into the behavior and impact of agile practices, such as refactoring and pair programming, on agile projects. The authors show that refactoring plays a crucial role in quality assurance (QA) and affects the cost of implementing change and overall project productivity. Delays in refactoring are shown to significantly increase costs and slow development progress, emphasizing the need for timely code maintenance. Additionally, the model demonstrates that pair programming, while more resource-intensive, can lead to higher code quality and reduced rework, ultimately optimizing task completion rates. These findings highlight how the SD model can serve as a simulator to test and refine strategies for effective agile software project management.

III Theoretical Framework

GenAI tools can significantly enhance developer productivity during the initial stages of software development (Peng et al., 2023) and reduce overall development costs (Chui et al., 2023). However, concerns have been raised regarding the possible drawbacks of these tools, including higher complexity of generated code compared to human-written code (Idrisov & Shippe, 2024). In other words, large volumes of generated code increase the overall complexity of code (Idrisov & Schlippe, 2024), decreasing its overall quality (Imai, 2022). Generated code may not always adhere to project-specific coding standards, patterns or best practices. When accumulated into large volumes over time, generated code lacks a comprehensive documentation or comments explaining the rationale behind certain implementations making it difficult for developers to understand and modify the codebase, especially for long-term maintenance. Thus, the short-term benefits of GenAI could lead to long-term implications, accruing debt that needs to be addressed in the future.

The complex interplay between the positive and negative effects of generated code over time leads to unpredictable outcomes for the overall quality and productivity of the agile software project, which I study through the lens of Technical Debt (TD). While individual pieces of generated code may have relatively low complexity and quality issues, the accumulation of a large volume of generated code over time leads to significant accumulation of technical debt.

III.1 The Concept of Technical Debt

Technical debt (TD) is a metaphor introduced by Ward Cunningham (Cunningham, 1992) to describe the long-term costs incurred from taking shortcuts in software development to meet immediate goals (Curtis et al., 2012). This concept highlights that while these compromises can accelerate development in the short term, they accumulate “debt” that must be repaid through future maintenance and refactoring, leading to increased costs, decreased maintainability, and difficulty in modernizing the software (Tom et al., 2013). TD extends beyond just code to include technology and architectural decisions, design compromises, and insufficient documentation (Li et al., 2015). It is both a tactical and strategic construct: Taking on TD might provide short-term benefits but poses significant risks if not managed properly, creating long-term detrimental effects on software products (Brown et al., 2010).

TD is accumulated in various forms in software development. Intentional TD is incurred through tactical decisions to speed up immediate development, with full awareness that the issues will need to be addressed later. Unintentional TD, on the other hand, results from poor oversight, lack of knowledge, or the accumulation of coding errors and growing code complexity (Li et al., 2015). From a software development perspective, TD can be classified into different types, including code debt, design debt, and architectural debt, with code decay and architectural deterioration often cited as significant reasons for incurring TD (Tom et al., 2013). These classifications help in identifying and managing TD across different phases of the software lifecycle, facilitating better decision-making and prioritization of debt repayment.

III.2 Theoretical Framework of Technical Debt

Tom et al., (2013) proposed a comprehensive theoretical framework to better understand the multi-faceted nature of TD. This framework (Figure 1) integrates dimensions of TD, its attributes, precedents, and potential outcomes. It conceptualizes technical debt as a practical model that can aid in identifying, tracking, and managing TD in software development. This framework categorizes TD into four types:

1. **Strategic Debt:** TD is proactively incurred to gain competitive advantages, such as faster market entry, with the understanding that it will need to be addressed in the future.
2. **Tactical Debt:** Short-term trade-offs made under schedule pressure to meet deadlines or critical milestones.

3. **Incremental Debt:** Small unnoticed issues like poor naming conventions, non-compliance with coding standards or lack of documentation that accumulate over time and become difficult to manage as software grows.
4. **Inadvertent Debt:** Debt accumulating from lack of oversight or knowledge when developers make suboptimal decisions without realizing the long-term implications, leading to future technical complications.

The framework sheds light on attributes of TD such as monetary cost, difficulty to quantify, “interest” (the ongoing effort needed to deal with TD) and principal (the effort to pay it off), and the potential for debt amnesty or even “debt bankruptcy”. Precedents are factors that lead to technical debt, such as prioritization of speed under resource constraints, project time pressures, or lack of comprehensive design reviews. Outcomes of technical debt include impacts on team morale, productivity, product quality, and project risk.

Most importantly, the outcomes illustrate how TD affects quality and productivity over time. In the short term, accruing technical debt can boost productivity by allowing faster delivery, but often at the cost of reduced quality. Over the long term, unresolved TD negatively impacts productivity as teams face slower development due to accumulation of complex, difficult-to-maintain code and increased defects.

Agile software development emphasizes rapid delivery and frequent releases, Agile practices' iterative and customer value-focused nature sometimes leads to trade-offs, such as reducing focus on code optimization and peer review in favor of rapid delivery, contributing to the buildup of TD (Li et al., 2015). While GenAI helps increase productivity and rapid delivery initially, generating large volumes of code over time contributes to code complexity and poor quality, resulting in accumulated TD. Managing TD in agile software development often requires making difficult decisions, such as incorporating frequent code reviews and refactoring (which may slow product delivery) to keep debt in check (Behutiye et al., 2017). If TD, accumulated through GenAI use is not managed early throughout development, it can degrade software quality, reduce productivity, and increase maintenance effort and costs over time (Tom et al., 2013). In a scenario with initially low complexity and few quality issues, the accumulation of a large volume of GenAI-generated code over time can still lead to significant growth in technical debt.

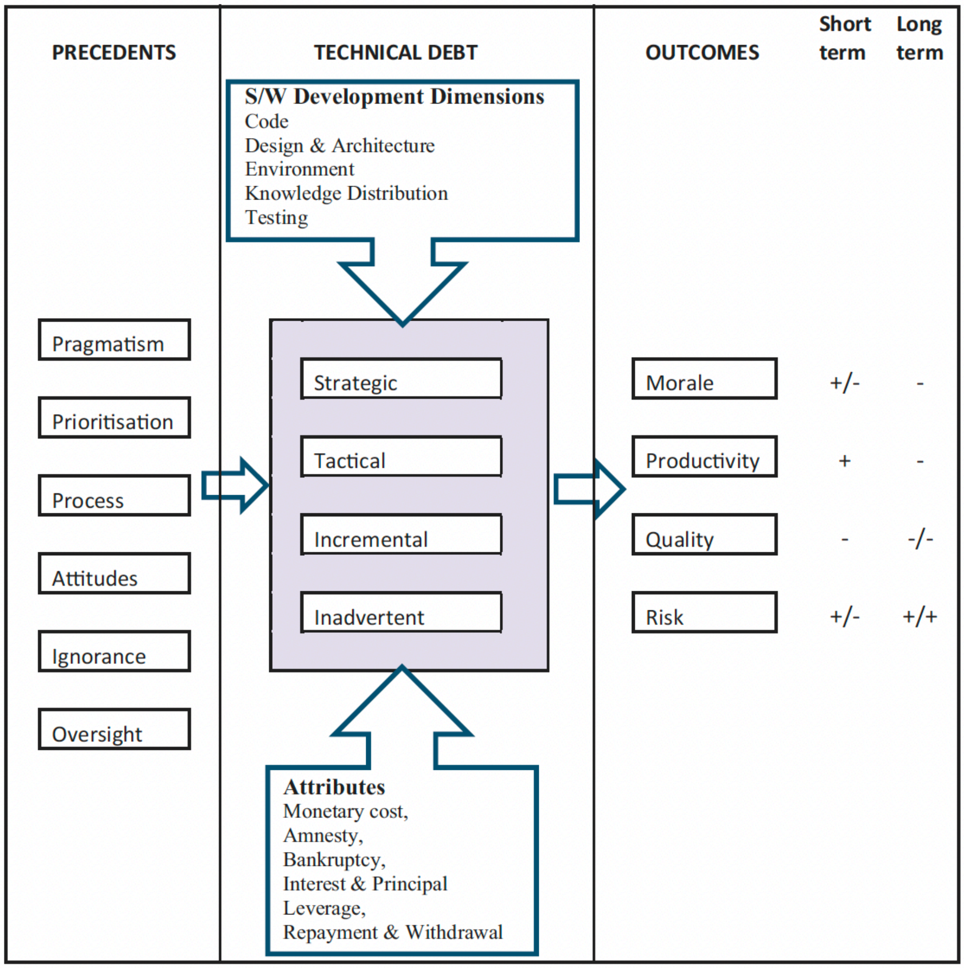

Figure 1. Theoretical framework of technical debt (Tom et al., 2013)

This research focuses on studying how the short-term productivity gains from GenAI contribute to the accumulation of technical debt over time due to poor quality, resulting in reduced productivity in the long term. GenAI influences multiple dimensions of the software development process, including code, design and architecture, development environment, knowledge distribution, and testing, which can result in the accumulation of strategic, tactical, incremental, and inadvertent TD, thereby impacting productivity, quality, team morale, and risk outcomes. I study GenAI use in agile development through the lens of the TD framework (Figure 1). However, my study focuses only on productivity and quality.

IV Methodology

IV.1 System Dynamics Modeling

System dynamics is a methodology and modeling technique used to understand and analyze complex systems over time. It was developed by Jay Forrester at MIT in the 1950s and has since been applied to various fields, including business, economics, environmental studies, and social sciences (Ford & Sterman, 1998). Key Concepts of system dynamics include:

Feedback Loops: System dynamics focuses on the interactions between inter-related components of a system, particularly causal feedback processes. A feedback loop is a closed sequence of causes and effects, a closed flow of actions and information (Abdel-Hamid, 1984). These loops can be either reinforcing (R), which lead to an exponential change in the outcome, or balancing (B), which lead to a stabilizing change in the outcome.

Stocks and Flows: The model represents system elements as stocks (accumulations) and flows (rates of change).

Time Delays: System dynamics accounts for the delays between actions and their effects within a system.

Non-linearity: The approach recognizes that relationships between variables in complex systems are often non-linear.

System dynamics uses causal feedback loops, stocks and flows, and time delays to capture the causality that exists among the elements of a system over time (Randers, 1997; Sterman, 2000). Feedback loops capture circular causality, and stocks and flow elements capture potential delays in the system over time (Fang et al., 2018).

IV.2 Structure of System Dynamics Models

The primary objective of SD models is to study how feedback structures within a system drive its behavior, enabling the identification of leverage points within the system that can influence desired outcomes (Kampmann & Oliva, 2008). SD assumes that the system being modeled has a causal structure with feedback loops (Richardson, 2011). An SD model's structure comprises of various entities of the system and relationships between them that are causal rather than correlational (Bala et al., 2017). Causal feedback loops help capture the system's recursive behavior over time by identifying relationships between system entities that form a closed causal loop. A causal loop diagram is a simple way to depict the structure of the SD model, and shows major components of the system, the causal relationships between them, and the feedback loops that characterize the system's dynamic behavior. Figure 2 shows an example of a causal feedback loop of and SD model of new product adoption. Reinforcing loops (R) in an SD model's description represent positive feedback mechanisms that amplify changes and drive exponential growth or decline, whereas balancing loops (B) represent negative feedback mechanisms that work to stabilize the system and maintain equilibrium.

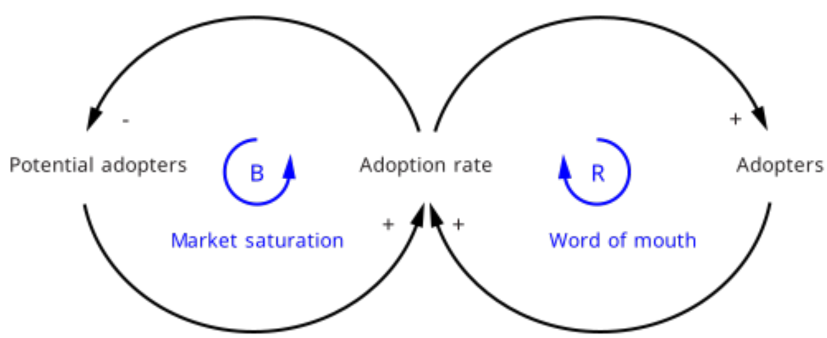

Figure 2. Example - Feedback loop of new product adoption

Feedback loop diagrams (Figure 2) aid in visualizing a system's structure and behavior at a high level of abstraction. To model and analyze the system more quantitatively, a stock-and-flow description of an SD model is used (Figure 3). Stocks and flows represent accumulations of various entities and their rate of change and introduce path dependence and inertia to describe the changes in system components (Forrester, 1968). Delays in inflows and outflows are an important characteristic of a stock-and-flow description of an SD model (Forrester, 1971). Stock-and-flow depictions are usually built and simulated using computer software. Table 4 describes the elements of stock-and-flow depictions of SD models, and Figure 3 gives an example of a stock-and-flow description of an SD model for new product adoption.

Stock-and-flow descriptions of SD models also rely on mathematical equations to define each component's behavior (Sterman, 2000). These equations connect stocks, flows, auxiliaries, and constants, allowing system behavior to be examined through simulation. SD models depict systems in continuous time. This approach enables the analysis of how accumulations impact feedback mechanisms continuously over time. However, discrete time steps are used for computational simulations run on the SD models (Sterman, 2000; Richardson, 2011).

Table 4. Elements of stock-and-flow descriptions of SD models

SD Model Component	Description
Stock	Represent the accumulation of entities within the system, such as resource stock, population count, or financial reserves, etc. These levels change over time due to the net impact of inflows and outflows.
Flow Rate	Indicate the movement of resources or information between levels and are influenced by decision-making processes and external variables.
Information Channel	Connections that facilitate communication between different parts of the system, ensuring that decision functions are informed by the current state of levels and feedback mechanisms. These connections form the basis for the reactive and adaptive behavior of SD models.
Auxiliary Variables	Variables that influence the flows or stocks in the model without accumulating over time. These inputs often include constants or external variables that impact the system's behavior.
Source/sink	Elements that act as origins or endpoints of stocks in a system, representing where entities such as resources or information enter (source) or leave (sink) the system.

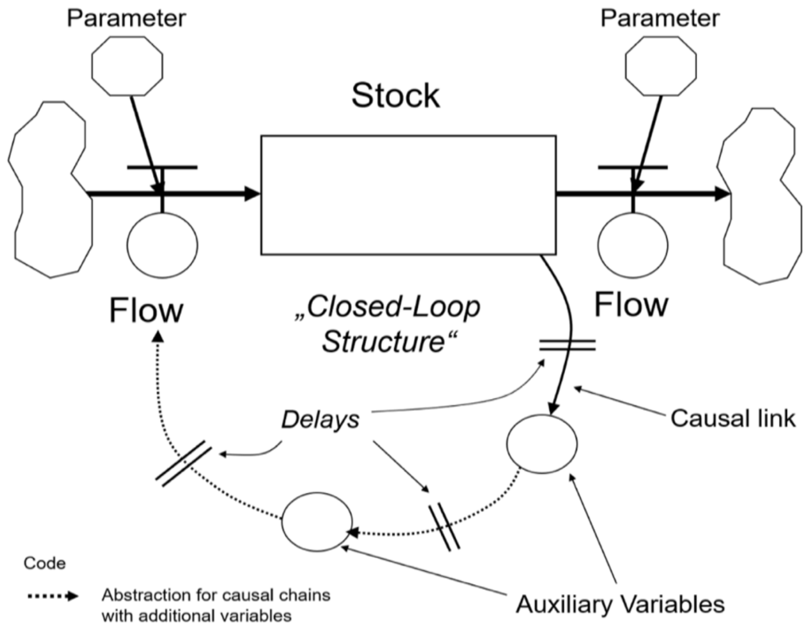

Figure 3. Example stock-and-flow diagram with components (Schoenenberger et al., 2021)

Simulations run on SD models allow researchers to observe how the system behaves under different scenarios and to conduct sensitivity analyses on key parameters (Sterman, 2000). Model behavior can be validated by comparing simulation outputs with real-world data or known patterns from literature.

IV.3 Appropriateness of SD to Study Impact of GenAI on Agile Software Development

Agile practices like refactoring, unit testing, cyclical integration of software, customer involvement, and continuous updates to the software are interlinked, creating a complex sequence of cause-and-effect relationships with feedback cycles. SD effectively captures these dynamics, as demonstrated in prior software process modeling research (Abdel-Hamid, 1989), making it an appropriate method for studying the dynamic interactions between GenAI tools and agile software development practices that affect project outcomes like development productivity and software quality.

The effectiveness of the system dynamics methodology in modeling software development practices is well established (Rodrigues & Williams, 1997). Extending this research, I use it for capturing the dynamic impact of GenAI on the productivity of agile software projects. This research is among the first to apply SD to study the impact of GenAI tools on agile software development practice.

IV.4 Research Approach

This study follows the SD modeling method that comprises the stages of dynamic hypothesis development, initial model development and refinement based on real-world project data and interviews, and model validation through independent data collection and analysis after model development (Cao et al., 2010).

Synthesizing knowledge from published literature on agile software development dynamics and the impact of GenAI tools on software development outcomes, I developed a dynamic hypothesis (Section 5) and a partially developed an initial model (Section 6). Interviews with software development professionals in agile software development projects are conducted to further help build the initial model and refine it. Project-related metrics collected using the project management tool Atlassian JIRA from a separate agile project are used for behavioral validations.

IV.4.1 *Problem Definition: Dynamic Hypothesis*

A dynamic hypothesis represents a working theory on a system or a part of it reflected in the model based on real-world data and knowledge (Sterman, 2000). The hypothesis is both dynamic and provisional:

1. **Dynamic** because it explains the feedback-driven nature of the system, and
2. **Provisional** because it may evolve with further insights from data collection

Section 5 details the dynamic hypothesis that guides the development of the SD model, capturing the impact of GenAI tools on software quality and development productivity in agile projects. The dynamic hypothesis is proposed based on the theoretical lens of technical debt (presented in Section 3).

IV.4.2 *Initial Model Development*

The dynamic hypothesis reflects the abstract description of the system of interest. Developing the detailed SD model involves creating a stock-and flow representation of the system components, along with the corresponding equations that define the behavior of these components. Initial model development is accomplished through two approaches, and their combination. These two approaches are:

1. **Utilizing relevant well-established literature** to develop the stocks, flows, causal and dynamic relationships, and parameter values, for building the SD model, as exemplified by Fang et al., (2018)'s work that uses extant literature on the impact of E-commerce resource endowments to create a system dynamic model of the impact of e-commerce endowments on firm performance.
2. **Collecting real-world data** from interviews with relevant stakeholders and secondary sources, as exemplified by Cao et al., (2010) that used interview data from software development professionals to build an SD model of agile software development.

To aid initial model development, I leveraged an existing well-established SD model of agile software development (Cao et al., 2010), as a baseline. This baseline model is the most comprehensive and well-cited SD model of agile software development to date. Next, I incorporated the dynamic hypothesis (described in detail in Section 5) and specific empirical findings from prior literature to develop the SD model components (summarized in Table 4), building the initial version of the model, while retaining many of the baseline model's core variables and relationships. I retained a large portion of the baseline SD model because it is well established and already incorporates key agile practices such as customer involvement, pair programming, and refactoring, and has been empirically validated with real-world data. My interest is specifically to model how GenAI tools impact agile software development practices. The initial model is described in Section 6.

To complete the development of the initial version of the model comprising the relationships between GenAI tools and agile variables, interviews were conducted with software development professionals from a medium-size software company. Analysis of the qualitative and quantitative data from the interviews contributed to constructing accurate structures and components for the initial SD model.

Data collection for initial model development and model refinement occurred in two separate phases, through semi-structured interviews conducted virtually. Participants were solicited through social and professional networks. A convenience sampling method was adopted to recruit software professionals who are currently working in agile software development projects and are using GenAI tools. All participants were pre-screened to ensure that they fit the study criteria. Each interview lasted about 45-60 minutes and was voice recorded with participants' consent. The voice recordings were transcribed. Data collection and analysis were done iteratively. The interviews consisted of about 10 open-ended questions that were approved by GSU's IRB. The transcripts of the interviews and interview notes were analyzed and coded using NVivo software (https://lumivero.com/products/nvivo/). An initial coding scheme is developed based on a review of the literature and the initial model, and it was updated based on data analysis (Charmaz, 2014).

I interviewed 18 software professionals from multiple companies, the details of which are as follows:

Participants: 18 software development professionals working in agile software development projects and actively using GenAI tools.

Companies: 4 technology companies engaged in agile software development.

Projects: Agile software development projects (3–9-month duration), team size of 4-8 members, running 2-week sprints.

Types of data collected: Metrics on the percentage of code generated using GenAI tools, measures of development productivity, code complexity, code quality, code refinement effort, efforts to fix code complexity issues, overall software quality, refactoring effort, testing effort, and documentation effort.

IV.4.3 *Model Refinement*

In the model refinement phase, further interviews were conducted in multiple agile software development organizations to refine the initial model. In this phase, I gathered additional data from software professionals across our focal organizations, with interviews involving two to five developers per organization. This resulted in a more generalizable SD model that reflects the real-world impact of GenAI on agile practices more accurately.

Table 5. Interviewee Titles and Experience

Phase	Interview #	Interviewer Title	Experience	GenAI Use
Initial Model Development	P-001	Software Developer	6 Years	1.5 Years
	P-002	Software Developer	6 Years	1 Year
	P-003	Software Developer	6 Years	1 Year
	P-004	(Jr.) Data Science Developer	1 Year	8 months
	P-005	(Jr.) Software Developer	1 Year	2 Years
	P-006	Sr. Software Developer	15 Years	1 Year
Model Refinement	P-007	Software Developer	2 Years	1 Year
	P-008	Software Architect	15 Years	1.5 Years
	P-009	Sr. Software Architect	16 Years	1 Year
	P-010	Chief Architect	25 Years	1 Year
	P-011	Sr. Software Developer	8.5 Years	1 Year
	P-012	Software Developer	2.5 Years	1 Year
	P-013	(Jr.) Software Developer	1.5 Years	1 Year
	P-014	VP Software Engineering	20 Years	2 Years
	P-015	Principal Software Developer	20 Years	5 Years
	P-016	Data Science Developer	5 Years	8 months
	P-017	Data Science Developer	5 Years	2 Years
	P-018	Principal Software Consultant	20 Years	2 Years

IV.4.4 *Model Validation*

Model validation is a crucial step in model-based methodologies, performed after developing the model but before engaging in behavior analysis or proposing /policy recommendations. Validation ensures that the developed SD model accurately represents the real-world system and gives reliable results. SD employs three categories of validation tests:

direct structure tests, structure-oriented behavior tests, and behavior pattern tests (Barlas, 1996). Table 6 details the validation tests, and their corresponding procedures recommended in the SD methodology literature.

These tests were conducted to validate the SD model developed in this research. Some structural validation tests required data collection and analysis through interviews with software professionals at agile software organizations. Behavior-reproduction tests (Forrester, 1980) are conducted to compare the simulated results of the model with project behavior data (time-series metrics of project productivity and quality) that is obtained from an independent real-world agile software development project.

Table 6. SD model validation tests (adapted from Fang et al., 2018)

Test Category	Test	Purpose	Procedure
Direct Structural Tests (Forrester and Senge, 1980; Richardson and Pugh, 1981)	Structure Assessment	Ensures that the model structure aligns with the descriptive knowledge pertinent to the system.	Examine causal loops, stock and flow diagrams, and model equations. Conduct interviews, workshops to gather expert opinions, use archival data, and directly observe system processes.
	Parameter Assessment	Verifies that parameter values align with descriptive and numerical insights about the system.	Apply statistical methods to estimate parameters, or judgmental methods based on interviews, expert input, and firsthand experience.
	Boundary Adequacy	Confirms that key concepts essential to the problem are included within the model's boundaries.	Use boundary charts, variable lists, or causal loops to clearly present endogenous and exogenous variables for review.

	Dimension Consistency	Checks that each model equation is dimensionally consistent without involving parameters that lack real-world relevance.	Employ dimensional analysis tools and inspect model equations for problematic parameters.
Structure-Oriented Behavior Test (Barlas, 1996; Forrester and Senge, 1980; Sterman, 2000)	Extreme Conditions	Validates that the model behaves logically under extreme policy scenarios.	Assess the model’s response to extreme values for each input individually and in combination.
	Sensitivity Test	Evaluates how significantly numerical values and behaviors shift in response to variations.	Test the model’s response to a range of numeric values and check if the generated behavior aligns with the actual system.
	Integration Error	Ensures results remain stable across different time steps or numerical integration methods.	Reduce the time step by half and assess behavioral changes; also test alternative integration methods for consistency.
Behavior Reproduction Test (Forrester, 1980)	Behavior Reproduction	Ensures that the model accurately replicates the system’s key behaviors.	Qualitatively compare model output with real data by examining behavior modes, variable shapes, and relative amplitudes. Calculate statistical measures to assess the alignment between model and data.

Validation ensures that the developed SD model can simulate outcomes that are closely aligned with the real-world system and produce reliable results. Structural validation of the SD model involved final interviews with developers and managers at agile software development organizations. Behavior-reproduction tests (Forrester, 1980) validate the model by comparing the simulated results of the model with project results collected from agile software development organizations.

After the model is structurally validated, I conducted behavior validation using further interview data and development metrics including productivity and quality data from an independent real-world project to compare and validate with the simulation results.

V Dynamic Hypothesis

V.1 Overview

To build the SD model that evaluates the impact of GenAI use on development productivity and software quality in agile projects, I start by formulating a dynamic hypothesis. The dynamic hypothesis, represented as a causal feedback loop (Figure 4), is based on the theoretical framework of technical debt discussed in Section 3. The dynamic hypothesis outlines how specific GenAI use cases impact software code complexity, quality, and productivity in the short and long term in agile projects. I bound the scope of the dynamic hypothesis before building it step-by-step.

The dynamic hypothesis primarily focuses on code complexity or code debt, a type of TD that refers to suboptimal code that accelerates initial development but hampers future maintenance. While other types of TD like architectural debt and design debt are important, I focus on code debt for two reasons. First, it is the most frequently studied type of TD that directly impacts development productivity and software quality in agile projects (Li et al., 2015; Behutiye et al., 2017). Second, recent research indicates that coding is the predominant use case for GenAI tools in software development (Ebert & Louridas, 2023). A large volume of AI-generated code can lead to more complexity than human-written code, potentially contributing to code debt (Idrisov & Schlippe, 2024).

Although popular GenAI tools like GitHub Copilot and OpenAI Codex can aid agile teams in various software development-related activities, I focus only on those use cases of GenAI tools that are used in the major software production activities of coding, Quality Assurance (software testing), and refactoring. These use cases directly impact software quality and development productivity, which are the outcomes of interest in my research question.

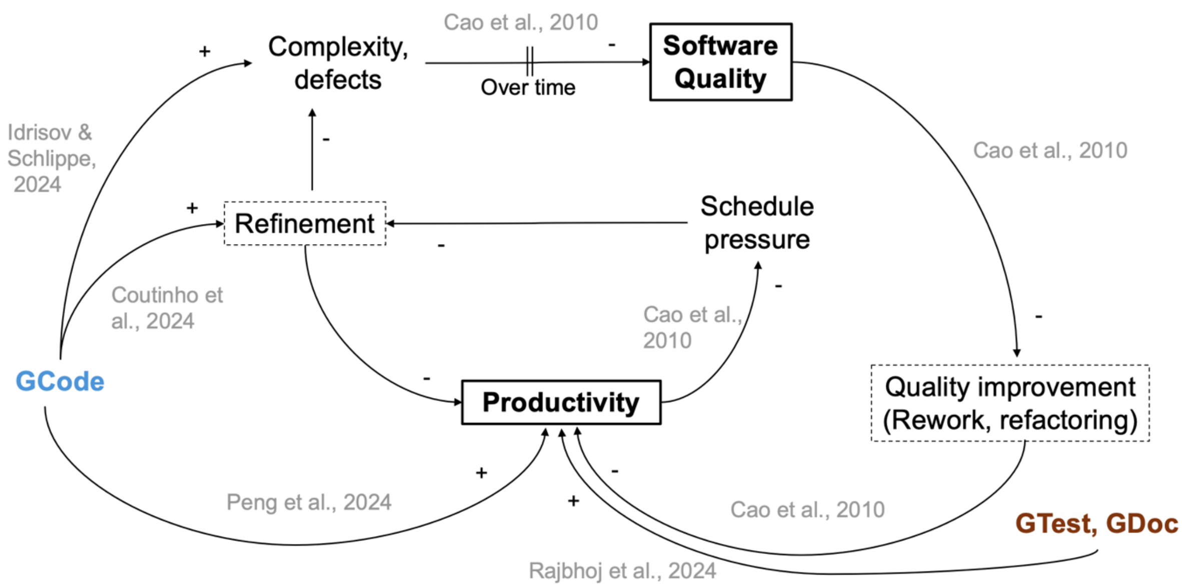

Figure 4. Dynamic hypothesis of GenAI's impact on software quality and productivity.

V.2 Feedback Loops

The combined effect of both reinforcing and balancing loops is shown in Figure 5 and described below.

Balancing feedback loop: Balancing feedback loops are processes that counteract change and tend to stabilize systems. Even though GenAI boosts development productivity, developers need to spend additional effort to refine the generated code in order to improve its quality. This added rework effort results in diminished net impact of GenAI on development productivity. As productivity goes down due to time spent on refinement, schedule pressure increases, forcing the team to bypass refinement activities in the next cycle in order to meet deadlines. Thus, refinement effort balances out the productivity changes in the system through schedule pressure. Therefore, refinement acts as the balancing factor in the feedback loop as shown in Figure 5. This scenario aligns with prior observations that under time pressure, teams tend to postpone quality-improvement activities to meet delivery dates (Austin, 2001).

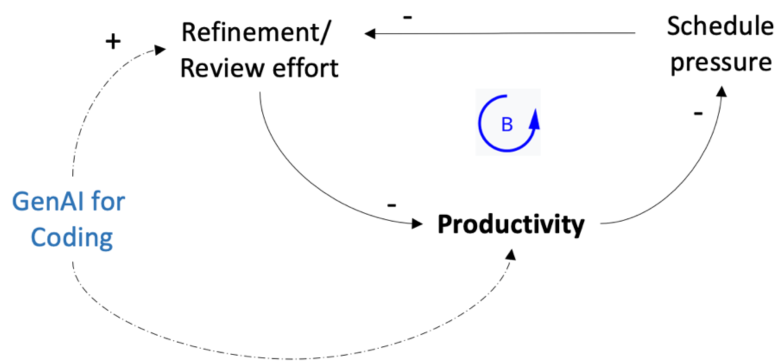

Figure 5. Balancing feedback loop

Reinforcing feedback Loop: Reinforcing feedback loops are processes in which an initial change in a system leads to additional changes that amplify the original effect. As GenAI increases software development productivity, it results in a reduction of schedule pressure. Reduced schedule pressure gives the development team the ability to allocate sufficient time to work on (generated) code refining activities, resulting in reduced code complexity. As the code becomes less complex and easier to understand, the software quality improves, which in turn reduces the effort needed for quality improvement activities, thereby further improving development productivity. However, if the team spends less effort on refinement, code complexity accumulates, reducing software quality, and requiring higher effort on quality improvement activities such as refactoring and rework, which then reduces development productivity. In this way, the refinement effort of developers reinforces the productivity changes in the system by counteracting the needed quality improvement effort, as shown in Figure 6. Allocating sufficient time for continuous refinement and testing thus reinforces positive outcomes, echoing the importance of refactoring in sustaining long-term agile performance (Fojtik, 2011).

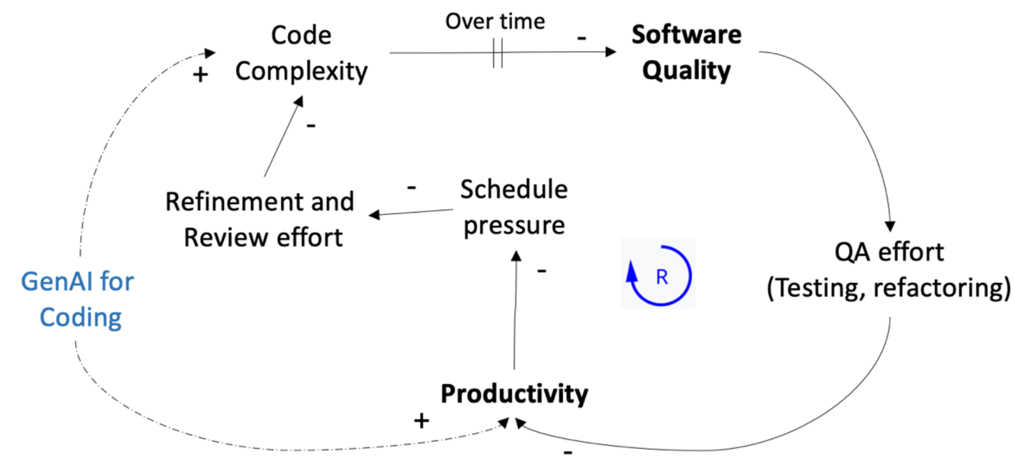

Figure 6. Reinforcing feedback loop

Using GenAI for coding can boost immediate productivity at the cost of increased code complexity and code debt, which may decrease software quality and development productivity in the long run due to more effort being needed in code refinement, review, and QA activities. However, using GenAI tools in QA activities such as automated test case generation and refactoring can reduce the human effort needed for these tasks, improving overall software quality and productivity. Figure 5 illustrates this dynamic hypothesis by showing how GenAI usage in different aspects of software development influences software quality and development productivity. The variables “code refinement and review effort” and “software quality improvement effort” capture the effort needed to reduce code complexity and to fix degraded software quality respectively.

Impact of GenAI on coding productivity: GenAI tools have the potential to boost coding productivity as they help developers write code faster. In one study, a productivity gain of 55% was reported from using GitHub Copilot for a coding task (Cui et al., 2024). While such immediate productivity gains from using GenAI are evident in simple coding tasks, their applicability to the broader, complex landscape of agile projects, characterized by long-term, interdependent, and feedback-driven activities, remains uncertain. It is important to note here that GenAI usage for coding is modeled as a continuous variable, and the immediate productivity gains realized will depend on the proportion of code written with the help of GenAI tools. Similarly, a recent field study found generally positive effects of GenAI on individual developer productivity but emphasized unresolved challenges in more complex, team-based scenarios (Coutinho et al., 2024).

Impact of GenAI on code complexity: Although GenAI can enhance coding productivity, it may lead to increased code complexity. Research indicates that AI-generated code often exhibits higher cyclomatic complexity and resource usage, especially in complex tasks (Idrisov & Schlippe, 2024). Limitations in context length and project-related information available to popular tools like GitHub Copilot can contribute to the generation of code that is syntactically correct yet suboptimal, potentially increasing code complexity over time.

Managing code complexity in the short term: In agile projects a part of the development effort is allocated to code refinement and peer review immediately after coding, to minimize coding errors, control code complexity, and reduce the risk of accruing code debt (Cao et al., 2010). It is logical to assume that the more code generated by GenAI tools, the more code refinement and peer review effort is needed to reduce code complexity.

Unchecked code complexity's impact on software quality: Sustained buildup of complex code over time gradually increases code debt in agile projects (Li et al., 2015). Without effective management, this buildup degrades software quality and productivity as the project progresses, aligning with the theoretical framework of TD (Tom et al., 2013) and evidence from literature on TD in agile software development (Behutiye et al., 2017).

Managing software quality in the long term: In agile methodologies, a part of the project team is responsible for conducting intermittent quality assurance (QA) activities that include software testing and refactoring (Fojtik, 2011). These activities help reduce the complexity of the source code, fix any coding errors that have slipped past peer review, and keep the software at an acceptable standard of quality. However, in the face of increasing TD, if QA activities are not undertaken more frequently, they face diminishing returns, especially for large software projects with many interdependent components (Cao et al., 2010).

GenAI tools can support QA activities like software testing (Ebert & Louridas, 2024) and refactoring (AlOmar et al., 2024) by reducing the amount of developer effort required for these activities. Unlike coding, where GenAI-generated code may increase complexity because the code is developed with a limited understanding of the context in which the code will be used, using GenAI for test-case generation and refactoring provides the AI with a broader context in which the software will be used. This broader context helps GenAI perform QA activities with high levels of accuracy and speed, while reducing the human effort needed, ultimately enhancing development productivity.

The negative reinforcement loop in the dynamic hypothesis highlights how neglecting code refinement and rework activities due to schedule pressure can result in an exponential decline in software quality. Agile teams tend to de-prioritize code refinement and rework activities to meet tight schedules (Cao et al., 2010). If left unchecked, these actions lead to an accumulation of code complexity originating from the generated code. This growing code complexity amplifies technical debt and degrades software quality over time. As the cycle repeats, software quality degrades at an accelerating rate, low-quality software increases the amount of rework effort needed, which further increases schedule pressure, leading to even less team effort allocated to refinement. Without intervention, this negative reinforcing cycle causes the software to require ever-higher levels of review and rework effort to maintain as time progresses. Leveraging GenAI for performing rework activities like software testing, and refactoring can help break this cycle by improving software quality with less human effort in rework activities, thus reducing schedule pressure.

At the initial stages of the project, developers ensure that the generated code is usable by spending enough time and effort on refinement activities. This leads to a positive cycle of productivity with diminishing returns as increased productivity from using GenAI for coding needs to be compensated with increased refinement effort, which reduces productivity. In turn, software quality and team productivity continue to improve. However, as the project progresses, the system transitions into the negative reinforcement loop if the team becomes less vigilant in following code refinement practices. Rising code complexity and degraded software quality require additional time for rework and fixes, eroding the initial productivity gains. Consequently, the overall development productivity due to GenAI is high initially. But due to the impact of the balancing loop of refinement, it starts declining immediately. Over time, due to a lack of sufficient refinement, the reinforcement loop takes over and degrades software quality, which eventually requires rework and refactoring, resulting in further decrease in development productivity (Figure 7). As stated before, using GenAI for rework activities of testing and rework can break the dominating negative reinforcement cycle, providing consistent productivity gain as the positive balancing loop dominates.

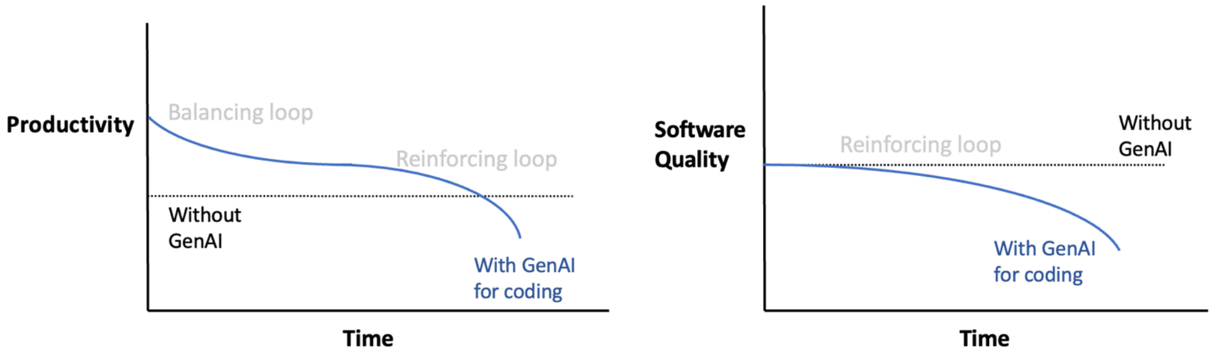

Figure 7. Hypothesized impact GenAI on development productivity and software quality

VI Model Development

This chapter introduces a new GenAI impact of software development subsystem that augments and extends the baseline system dynamics model of agile software development practices adapted from prior work (Cao et al., 2010; Abdel-Hamid, 1984). The baseline model already includes typical agile practices via its agile planning, refactoring and design quality, change management, and customer behavior subsystems. My proposed subsystem extends the baseline model in two ways. First, it models the practice of using GenAI tools in software development. Second, it captures the dynamic impacts of GenAI-generated code, test cases, and documentation, on agile practices and project outcomes.

Figure 8 gives an overview of the extended GenAI agile software development model and shows how the proposed subsystem (in green) connects to the other subsystems of the baseline model. Note that this dissertation does not discuss the "GenAI Impact on Human Resources" subsystem, which is borrowed from a work-in-progress (Negi and Kota, 2025) and models the effects of GenAI usage on project team members' learning, codebase knowledge, and evolving dependance on GenAI tools.

The GenAI impact on software development subsystem encompasses three primary areas of GenAI use, consolidated from prior literature and interviews with software development professionals. These are:

1. **GenAI for coding (GCode):** The proposed subsystem component GCode captures the impact of using GenAI tools to generate code in development tasks, including the amount of refinement needed to bring generated code on par with human developed coding standards, and the consequences of insufficient refinement of generated code.
2. **GenAI for testing (GTest):** The subsystem component GTest accounts for scenarios where GenAI tools are used to generate or enhance unit testing, thereby influencing overall test coverage and the detection of defects in developed code.
3. **GenAI for documentation (GDocument):** The subsystems component GDocument models how automated or semi-automated documentation with GenAI tools impacts the readability and maintainability of the developed software.

By introducing these three components or use cases of GenAI and their impacts into the baseline agile model, the new system dynamics model captures both the short and long-term implications of using GenAI in agile software development projects. The rest of this chapter details how these three use cases of GenAI interact with the baseline model's subsystems of refactoring and quality of design, quality assurance and testing, and development productivity, and highlights the reinforcing and balancing feedback loops (section 5) that are created because of these interactions.

VI.1 The GenAI Impact on Software Development Subsystem

The 'GenAI Impact on Software Development Subsystem' captures the direct and indirect dynamic impacts of using generated code (hereafter, "GCode"), unit test cases ("GTest"), documentation ("GDocument") and the refinement of generated code during the development process, on agile software development practice. This subsystem is built based on the literature review and interviews with software development professionals who are currently working on agile software development projects using GenAI tools. The subsystem's variables and relationships support the earlier laid hypothesis that while GenAI tools can boost the immediate productivity of developers by automating software development tasks, they also introduce additional complexities and defects over time, that, if not properly managed, can accumulate technical debt and degrade software quality and development productivity.

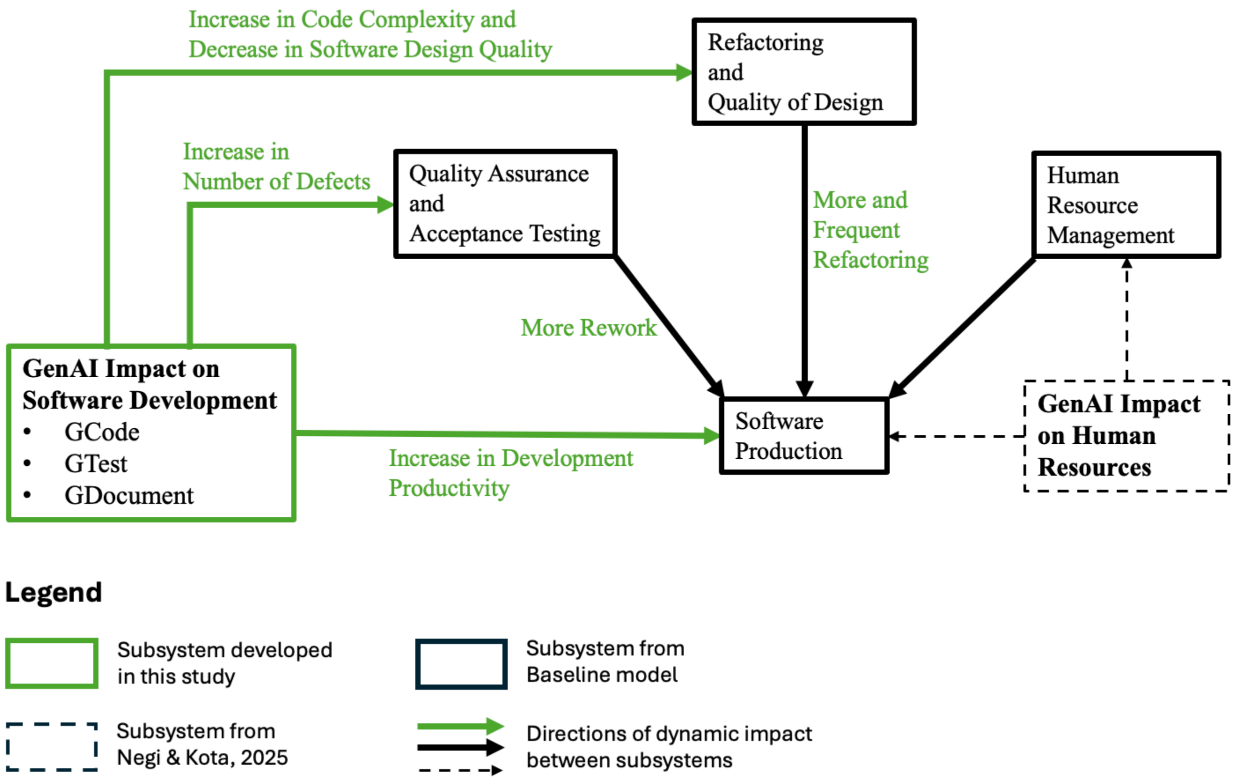

Figure 8. Model structure overview

Figure 9 presents a detailed view of the GenAI impact on software development subsystem. The GenAI use-case-related variables in this subsystem are described as follows, and are also in Table 7:

1. **GCode usage**: The fraction of total code in each iteration (sprint) generated by GenAI (e.g., ChatGPT, Copilot). Based on evidence from the literature and interviews, developers mainly use GenAI tools to generate code at the level of a snippet or a function/method, through either code-complete or prompt-to-code features. The value of the GCode usage variable is a fraction that can range from 0 to 1.
2. **GTest usage**: The fraction of unit-test cases in a given sprint that were generated by the developers using GenAI tools to test the developed code. Unit test cases are used to detect and fix potential defects in the developed code before it is sent to the QA engineer.
3. **GDocument usage:** This variable depicts the fraction of documentation generated using GenAI tools during the development phase by the team.

Table 7. GenAI use-case-related variables in the GenAI subsystem

Variable	Definition	User	Value
GCode usage	Fraction or percentage of total code generated by GenAI tool.	Software Developer	Continuous fraction [0 – 1]
GTest usage	Fraction or percentage of total unit test cases generated by GenAI tool.	Quality Assurance (QA) Engineer	Continuous fraction [0 – 1]
GDocument usage	Fraction or percentage of total documentation generated using GenAI tool.	Software Developer	Continuous fraction [0 – 1]

In the literature and interviews, it was also found that developers feel the need to refine or improve the code generated by GenAI tools. This refinement happens either through the user manually spending time fixing or improving the generated code or iteratively interacting with the GenAI tool through prompts to get improved code. However, whether sufficient refinement of the code is done is usually dependent on how much schedule pressure the development team faces to deliver the tasks, and whether the developer is a rookie or a pro. If the time pressure is high, refinement is usually neglected or insufficiently done. Also, in the interviews it was found that highly experienced developers generally write better prompts that lead to better quality of generated code and only accept code-complete suggestions that are of similar quality compared to or better than human developed code. These refinement-related variables that are modeled in the GenAI subsystem are depicted in Table 8 and described as follows:

1. **Needed refinement**: The portion of development effort saved by a developer from generating code using GenAI, that is required to refine generated code, so it meets the quality standards of human developed code.
2. **Actual refinement fraction**: The fraction of the needed refinement effort that the developer actually performs (influenced by schedule pressure) after generating code using GenAI.

3. **Gap of refinement**: The difference between the needed refinement effort and the portion of that effort which is actually performed (actual refinement effort) in a sprint.

Table 8. Refinement-related variables in the GenAI subsystem

Variable	**Definition**	**Value Range / Type**	**Impacted By**
Needed refinement	Fraction of total development effort required to refine generated code.	Continuous fraction [0 – 1]	GCode usage, Developer experience (rookie/pro)
Actual refinement fraction	Fraction of the needed refinement effort actually performed.	Continuous fraction [0 – 1]	Schedule pressure
Gap of refinement	Difference between needed refinement effort and actually performed effort.	Continuous fraction [0 – 1]	None

This GenAI impact on software development subsystem interacts with three major subsystems of the baseline agile software development model (Cao et al., 2010) briefly described as follows:

1. **Software production subsystem**: How GenAI usage affects the velocity of development tasks, and the net tasks delivered per sprint by junior (rookie) developers (less than 3 years of professional development experience) and senior (pro) developers (more than 3 years of professional development experience).
2. **Quality of design and refactoring**: How unrefined or under-refined generated code leads to design quality degradation and additional complexity in the software, requiring a change in refactoring practices.
3. **Quality assurance (QA) and acceptance testing**: How unrefined or under-refined generated code correlates with higher rates of bugs and defects, leading to more time spent in software testing and rework.

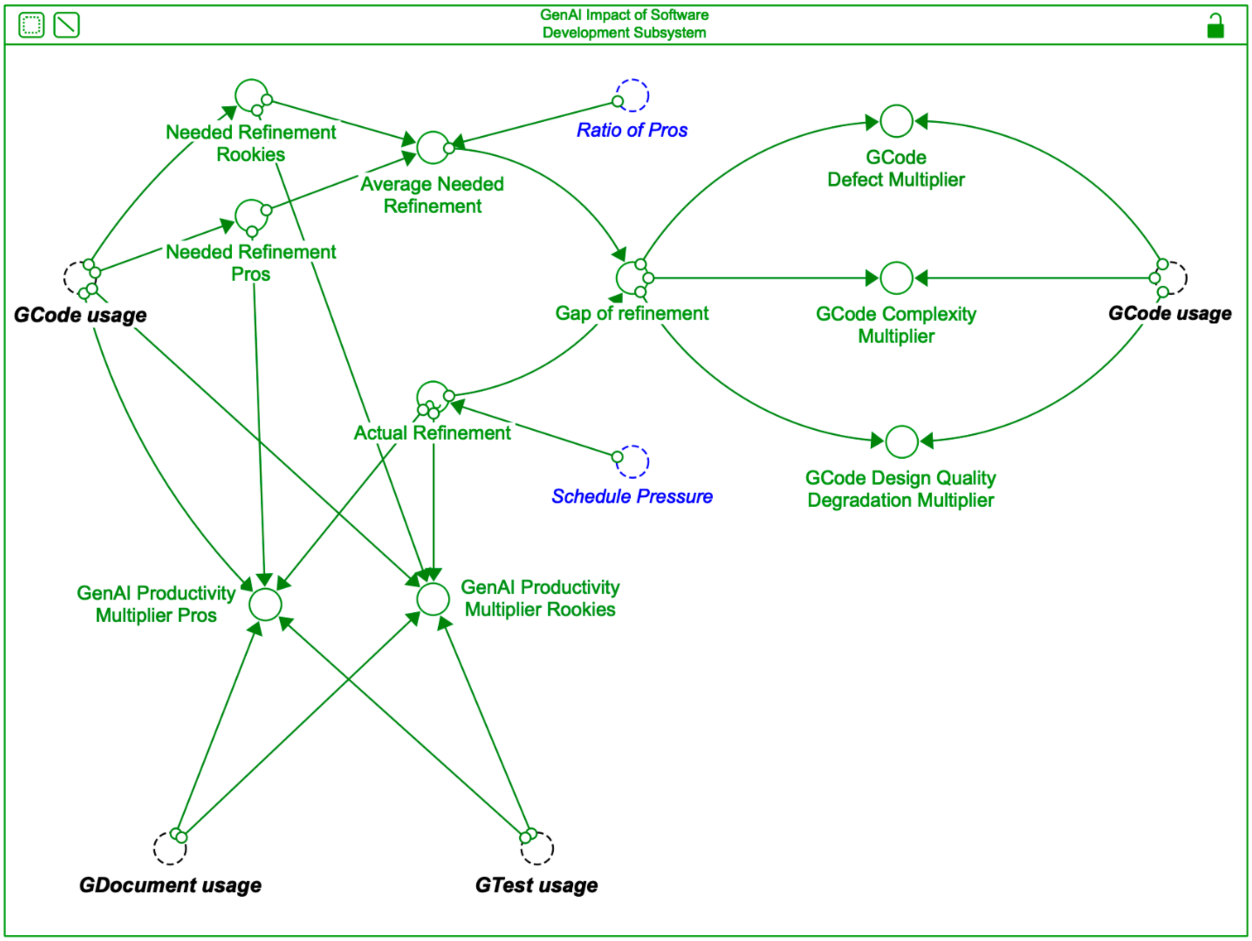

Figure 9. GenAI impact on software development subsystem

VI.2 Impact of GenAI on Subsystems of Baseline Model

VI.2.1 *Needed vs. Actual Refinement*

Developers often realize that AI-generated code can save coding time in the short run, but it requires additional refinement to ensure it meets the project's coding standards. Code refinement is the process of enhancing existing code by addressing potential issues, refactoring, and optimizing to improve quality and meet specific requirements before committing the code into the codebase (Guo et al., 2025). In the context of GenAI-enabled coding, code refinement is performed by the developer to improve, optimize, and customize the initial code produced by GenAI tools. GenAI can accelerate code generation, however, human review of generated code is crucial. Developers need to review GenAI generated code with the same rigor as human developed code to ensure quality and prevent technical debt. Refinement process includes checking for code consistency and style, readability and restructure, performance optimization, and security and best practices. Properly refined code improves code quality and maintainability, alignment with project-specific requirements, enhances performance and security, allows developers to understand the complex logic easily, and enables reusability.

One participant explained the need to refine generated code as below:

> "A new function might be generated and then the AI will get the function names and parameters and variable names, etcetera, pretty close. And then I go in and say, 'that's not exactly what I wanted' and make some tweaks to it. But it (GenAI) does increase productivity (P-013)."

Another participant mentioned:

> "Yes, if unchecked overtime, it will generate significant problems. Without that supervision, without the refinement it will create issues. I have the prompt done in say one minute and I spent the rest of the (time) refining it to get to what I need I do spend quite a bit of time refining it. (P-010)"

The same participant said that if less experienced developers (rookies) do not understand the generated code, it's a risk. This reiterates the need for refinement.

> "If the developer doesn't understand what the code itself is doing that is a problem. That's where I see a big risk (P-010)."

> "Refining that to meet my purposes I would say anywhere from like 10 to 20 minutes of refining, mostly because I want to ensure that I am not checking in any code that I don't understand (P-012)."

The total “Needed refinement” depends on the amount of code generated using GenAI tools (GCode usage) and on developer experience, where developers are classified as rookies (under 3 years of professional development experience) and pros (over 3 years of professional development experience) in the model. To refine the code, one participant said:

> “Definitely, we have to spend time (on refinement). If a tool gives me the code, I have to spend time to understand. I don’t want to go back to the tool tomorrow and work on it again, I need to absolutely make sure it is working well now (P-009).”
>
> “I have seen people just randomly copying and pasting that thing and when issue comes then they spend more time in troubleshooting or understanding the code (P-009)”

Another participant gave a numerical value for how much time they spend on refining generated code:

> “If we generated a significant amount of code, I would expect that, you know, 50% of the gains that we got from generating that code will be spent on making sure that it’s compliant and maintainable (P-014).”

For instance, in a typical 8-hour workday (480 minutes), with a load factor of 0.4, the developer spends 480*0.4=192 minutes of time on coding related work. If they write 100 lines of code in a day, out of which if 40% of code is generated using GenAI tool, it takes 192*0.4= 76.8 minutes less time to develop 100 lines of code. However, they need to spend 50% of that saved time on refinement, which is 76.8*0.5= 38.4 minutes. The net saving of developer time due to generating 40% of the code and refining it is 38.4 minutes out of the 192 minutes they would have spent otherwise. Thus, the time needed for refinement when 40% of the code is generated by GenAI is 38.4 minutes, which is 20% of the total development time of 192 minutes. In this case, the overall time taken to develop 100 lines of code is, 115.2 minutes (for 60% human coding) plus

38.4 minutes (for 40% generating and refining it), which totals to 153.6 minutes, that is 38.4 minutes less time (or saved time) due to GenAI.

Interview data suggested that developer experience plays an important role in how much time the developer needs to spend on refining generated code. Although the data shows a range of values for needed refinement, depending on developer experience the most common values of 50% and 70% were chosen for rookies and pros respectively. In other words, experienced developers (pros) who are proficient at prompting GenAI and refining its outputs effectively need to spend a less portion of their saved time (50%) on refinement, while less experienced developers (rookies) need to spend a larger portion of their saved time (70%) on refinement. The two variables in my model that capture the ratio of total development time needed to refine the generated code to the standards of human developed code are "needed refinement by pros" and "needed refinement by rookies", and are calculated as:

Needed refinement by pros = GCode usage × 0.5

Needed refinement by rookies = GCode usage × 0.7

In agile teams with a mix of pro and rookie developers, the **average needed refinement** is modeled as their weighted average:

Average needed refinement = (Ratio of pros) × (Needed refinement by pros) + (1-Ratio of pros) × (Needed refinement by rookies)

However, teams under tight schedules may not fully invest the time required to refine GCode (Austin et al., 2001). This behavior is captured via **Actual refinement fraction**, which decreases when schedule pressure rises:

Actual refinement fraction = 0.9 - (Schedule Pressure) × 0.05

(where schedule pressure ranges from 0 to 10)

Here, the actual refinement fraction is given a maximum value of 0.9, based on prior evidence from literature that suggests that developers usually have an imperfect understanding of generated code, which leads to the insufficient refinement and premature acceptance of the generated code (Weisz et al., 2021). Moreover, the schedule pressure influences developers to reduce or skip fraction of refinement. In the interviews, the schedule pressure that the developers faced as the project progresses was obtained in the form of a subjective value from 1-10, based on how much work the developer thought was remaining for the rest of the project duration (Cao et al., 2010). Data from interviews suggested that when the schedule pressure is extremely high (or 10), developers, on an average, spend only 40% of the needed time on refinement. Thus, in the formula, the actual refinement fraction reduces by a factor of 0.05 for every 1-point increase in schedule pressure, resulting in a maximum reduction of the actual refinement effort fraction due to schedule pressure from 0.9 (when schedule pressure is 0) to 0.4 (when schedule pressure is 10). In other words, if the schedule pressure reaches 10, the actual refinement fraction drops significantly (down to 0.4 in this linear representation), reflecting the evidence that high-pressure environments lead to a reduction in priority of quality-related tasks in favor of quickly delivering features (Austin et al., 2001). One participant described this phenomenon:

> "Yeah, it's definitely how people are using GenAI, because if they're sort of just dabbling in it, they'll probably push it (refining) to the side. When they feel the (schedule) pressure to get something done. But if they're truly using the product (GenAI tool), already comfortable using it, I know I would rely on it to help me save time. So, I can imagine most people would do the same, because why not use that tool more (P-018)."

VI.2.2 *Gap of Refinement*

When the actual refinement effort is less than what is needed, a **Gap of refinement** emerges.

This gap is modeled as:

Gap of Refinement = Average needed refinement × (1-Actual refinement fraction)

A high Gap of refinement indicates the team is accumulating “technical debt” in the form of under-refined generated code. Over time, this debt can hinder progress and degrade overall software quality (Tom et al., 2013).

VI.2.3 *GenAI Impact on Developer Productivity*

Prior literature shows that GenAI can boost productivity through automated code generation/completion (Peng et al., 2023), unit-test case generation (Bajaj & Samal, 2023), and documentation generation (Ebert & Louridas, 2023). Peng et al., (2023) found that the productivity boost is higher for less experienced developers. However, based on the evidence from interview data, the productivity improvement also depends on the actual refinement effort. Moreover, how much refinement is needed depends on the developer experience, where more experienced developers required less time for code refinement. It is evident that although using GenAI for coding leads to an overall increase in developer productivity, the effort spent on refinement mitigates a part of this increase, resulting in a diminished overall productivity boost. The overall development productivity boost is computed based on 1) percentage of code generated, 2) percentage of the saved time spent on refining generated code, 3) percentage of unit-test cases generated, and 4) percentage of documentation generated. The equations of productivity boost for rookie and pro developers are modeled using the two multiplier variables shown below:

GenAI productivity multiplier rookies

= 1 + (0.55 × GCode usage

× (1- (Needed refinement by rookies × Actual refinement)))

+ (GTest usage × 0.05)

+ (GDocument usage × 0.05))

GenAI productivity multiplier pros

= 1 + (0.8 × GCode usage

× (1- (Needed refinement by pros × Actual refinement fraction)))

+ (GTest usage × 0.05)

+ (GDocument usage × 0.05)

The values of 0.55 and 0.8 in the above equations indicate the respective expected productivity improvement rates for rookies (55%) and pros (80%) per development task, without considering the time spent on its refinement. The expected productivity improvement percentages constitute the % time saved when generating code using GenAI instead of writing it and are not 100% because developers spend some time on formalizing their thoughts on how to write prompts (in the case of prompt-to-text code generation) and initiating the code-complete recommendation of the GenAI tool, even before GenAI generates the code. These values are an average of the figures obtained from interviews with software developers, where they were asked how much time they would save if they accounted for the time spent on prompting GenAI or reading through its auto-complete suggestions, but did not account for refining the GenAI output. One participant claimed:

"Yeah, I can say like 70 to 80% coding productivity increase (before refinement). Sometimes it (GenAI tool) maybe inadequate, and I have to code myself, but mostly it will improve coding (productivity) for the application as well as helpful to enhance my coding skills (P-001)."

Another participant said:

"I'll put it (productivity boost before refinement) somewhere around a big interval like 25% to 55% (P-010)."

Third participant said:

"It would be like 30% to 40% (productivity improvement without refinement) definitely (P-009)."

"I would think it (GenAI) would save me at least two to three hours (of coding work) (P-011)."

Let us take an example of a senior (pro) developer and calculate their overall productivity boost based on the above equations. If the pro developer generates 50% of the code using GenAI, then the productivity boost they witness (not considering refinement of generated code and use of GenAI for generating documentation and unit-tests) is 80% for the generated code, that is 50% × 80% = 40% boost in productivity.

Next, the refinement-related term in the equations depicts how much of the productivity boost is throttled because of refinement. For the pro developer generating 50% of code, needed refinement is 25%, and the actual time spent on refinement is 90% × 25% = 22.5% of the time saved from generating code (assuming no schedule pressure). Thus, the productivity boost, considering refinement, is 50% × 80% × (100% - 22.5%) = 31%. Hence, the productivity boost, once refinement is considered, drops from 40% to 31%.

The last two terms in these equations correspond to the productivity boost witnessed from using GenAI to generate unit test cases and documentation respectively. Interviewees mentioned that they see productivity gains from generating test cases and documentation, but these gains are small, and usually under 10%. This is because they were already using IDE-based tools for unit-testing and documentation. So, the marginal gains in productivity from GenAI tools for unit-testing and documentation are modeled as 5% each. For the pro developer, using GenAI to generate 100% of the test cases and documentation leads to a total productivity boost of 31% + 5% + 5% = 41%. Thus, the net productivity boost for the pro developer generating 50% code after considering refinement, test-case generation and documentation generation is 41%. In other words, the GenAI productivity multiplier in the case of the pro developer is 1.41.

The highest productivity boost that can be achieved is by generating, testing, and documenting all code using GenAI, and spending the minimum 40% of the development time on its refinement (when schedule pressure >=10) leading to a productivity multiplier of 1.71 for pros and 1.44 for rookies, corresponding to a 71% increase in development productivity of pros and 44% increase in development productivity of rookies. This difference is intuitive as rookies need to perform more refinement of generated code to achieve acceptable code outputs compared to pros. Mentions of this phenomenon were also prevalent among interviews. For example, one participant said:

> "As a junior developer [in Python], I basically use it more like a tutor. Aas a senior developer [in Java], I use it more like a delegate. That's a difference in how I use it (P-008)."

It is worth noting that this observation is the opposite of what the majority of prior studies say about how expertise impacts the productivity boost attained from using GenAI coding tools (Peng et al., 2023). I argue that prior studies reach the opposite conclusion as they fail to consider the effort needed to refine generated code before committing it. In a production environment, developers perform rigorous checks on newly written code before committing it to production owing to accountability, and rookies must spend more time checking and improving generated code compared to pros. One study does mention that unrefined generated code takes much longer to integrate into the existing software compared to human-typed code (Song et al., 2024). One participant said:

> "I don't ask Copilot to write the entire class and methods. I define the function and then I keep in mind what I want that Copilot to do. That makes it more deterministic (P-008)."

Overall, the productivity multiplier equation for rookies and pros suggest that refinement effort eats away into the productivity boost attained from generating code instead of writing it manually, aligning with the behavior of the balancing feedback loop proposed in the dynamic hypothesis (section 5.2).

Finally, the overall enhanced productivity of rookie and pro developers is calculated using the productivity multipliers as follows:

Enhanced productivity rookies = Nominal potential productivity rookies × (GenAI productivity multiplier rookies)

Enhanced productivity pros = Nominal potential productivity pros × (GenAI productivity multiplier pros)

These equations represent the overall development productivity of rookie and pro developers, after accounting for how much code was generated, how much refinement was performed, and whether GenAI was used for generating unit-test cases and documentation.

VI.3 GenAI Impact on Refactoring and Quality of Design Subsystem

In agile development, refactoring is the process of reorganizing the structure of software code in a way that does not alter the original functionality. Refactoring aims to improve the software by making internal changes without altering its external behavior (Cao et al, 2010). Although GenAI can speed up code generation initially, insufficient refinement of generated code can threaten design quality of the software, requiring refactoring effort over time. Prior research in technical debt literature indicates that design debt accumulates quickly when development activities outpace quality checks and leads to a loss in productivity in the long run, since poorly designed software is harder to develop, improve, and maintain (Tom et al., 2013; Kruchten and Ozkaya, 2019). As one participant said:

> "I've found that sometimes the complexity gets out of hand the longer the process (of code generation) goes. You start to look at it and go - why would I need that? I do find that complexity of the code gets a little muddy sometimes when you have a lot that you're doing (generating). Reusability, maintainability suffers, especially when you're talking about large volumes (of generated code). It will rewrite these little functions (snippets) all over the place, and then you'll have a large code base, you may have 30 different implementations of that same bit of code. (P-015)"

VI.3.1 *Gap of Refinement, Code Complexity, and Design Quality*

When developers do not fully refine GenAI-generated code, the gap of refinement can have a detrimental impact on various quality metrics. As one participant said:

> "It [generated code] will introduce new bugs, and the code overlap over a period. The code will get unmanageable, and it would be difficult for us to manage it as its complexity increases."

In my model, the **GenAI design quality degradation multiplier** and **GenAI complexity multiplier** capture these detriments through the following equations:

GenAI design quality degradation rate multiplier = 1+ (GCode usage × Gap of refinement × 15)

As an agile software development project progresses, the quality of software naturally degrades due to the accumulation of technical debt as more code is created and integrated into the codebase. The rate of software quality degradation is faster when a portion of the added code is GenAI-generated. The GenAI design quality degradation rate multiplier represents how much GenAI-generated code, and its under-refinement contribute to deteriorating the overall design quality of the software. The more generated code there is (i.e., higher GCode usage) and the less refined it is (i.e., higher Gap of refinement), the steeper the decline in quality of the software design.

Interview data suggests that when about half of the code is generated by GenAI (i.e., GCode usage = 0.5), even under no schedule pressure (i.e., developers are not being pushed to skip refinement), the software design quality degrades roughly 20% faster. In other words, participants noted a ~20% more drop in software design quality over a given period of time due to the generated code. This effect of GenAI-generated code on design quality degradation rate is further amplified depending on whether it is refined sufficiently or not. Hence, the gap of refinement variable is multiplied to GenAI usage in the equation.

Mathematically, to result in a 20% increase in the software design degradation rate (i.e., GenAI design quality degradation rate multiplier = 1.2), when GCode usage = 0.5 and gap of refinement = 0.025, GCode usage and gap of refinement need to be multiplied by 15. This is how the final equation of the GenAI design quality degradation rate multiplier is achieved.

GenAI complexity multiplier = 1 + (2 × GCode usage) + (4 × GCode usage × Gap of refinement)

The **GenAI complexity multiplier** captures how the fraction of GenAI-generated code and its under-refinement together increase the overall complexity of the codebase, making it harder to understand, modify, and maintain.

The first term in the equation represents additional complexity of incorporating generated code in the codebase, irrespective of whether the generated code is sufficiently refined or not. Interviewees noted that as refinement is done for individual snippets of code that are generated, there is a tendency for unused or dead code to accumulate in the codebase when multiple generated code snippets are put together. As one participant said about dead code:

> "lot of unnecessary, duplicate code generated, it becomes dead code and not used for execution in production. The code bases get bloated and hard to understand and maintain. Maybe that can change in the future but now it's a problem to be mindful of. You can test it; it keeps on increasing the length of the answer (P-016)."

This additional dead/unused code leads to code complexity that is roughly double compared to that of human-developed code. This observation guided the choice of **2 × GCode usage** as the first component of the equation.

The second term reflects complexity introduced by under-refined generated code. Interview data suggests that when a generated code snippet was not refined, the complexity of the code snippet, in itself, was roughly 3-4 times that of a human-developed code snippet. Fitting this observation into the GenAI complexity multiplier required an extra term. This is how the factor **4 × GCode usage × Gap of refinement** was modeled in the GenAI complexity multiplier equation. Together, these two terms in the equation model the increase in code complexity due to 1) generated code and 2) the degree to which it is under-refined by the developer.

VI.3.2 *Refactoring to Address Complexity and Design Quality*

Addressing GenAI-exaggerated code complexity and design quality degradation rate requires more effort on refactoring activities. It is important to note the difference between code refinement and software refactoring. The code refinement refers to activities undertaken by a developer when the code is generated, before completing the coding task. Whereas refactoring is undertaken periodically when the software becomes less optimal and more complex due to accumulation of tech debt over time. Teams that regularly refactor may mitigate or eliminate accumulated design quality and complexity issues before they propagate. However, under time constraints or if the code "appears" functional, developers postpone or forego refactoring. As one interviewee described:

> "If we were heavily reliant on generative AI, it could get pretty costly. 15% maybe 20% of your time might be allocated to going back and refactoring (P-014)."
>
> "If we had the discipline from the beginning and refactor (code) as it's happening (being generated). But that is actually the issue with big corporations, they want to just move on to the next thing (postponing refactoring). I described the ideal scenario. In practice, that's not what we see (P-010)."

If the gap of refinement of generated code persists, the model tracks design quality degradation more rapidly, as more technical debt is accumulated faster. The accumulated debt eventually requires a "big refactoring," interrupting normal development to restore the system to a clean and maintainable state, which reduces development productivity during the time the big refactoring is taking place. This behavior of the model aligns with the reinforcing loop of quality improvement effort described in the dynamic hypothesis (section 5.2).

One participant said:

“I had a method which was about, I think 800 lines or something. The GenAI tool just did (refactored) it in like 3 minutes. But sometimes if I have to refactor code that’s when a lot of time goes into checking and refinement (P-012).”

Another participant mentioned that there is always a gap of refinement as developers do not fully understand the generated code:

“If the developer doesn’t understand what the code itself is doing, that is a problem. That’s where I see a big risk. Junior developers don’t have that discipline (P-10).”

VI.4 Impact on Quality Assurance and Acceptance Testing Subsystem

The QA and acceptance testing subsystem is also influenced by how thoroughly GenAI-generated code is refined. Specifically, the more under-refined the generated code submitted for QA is, the greater the likelihood of it containing hidden defects, bugs, and incomplete edge-case handling (Idrisov and Schlippe, 2024). Several interviewees said that while GenAI-generated code may compile successfully, it can hide deeper logic flaws that are missed when developers are unit-testing but are caught in the QA phase.

“But I'm working on a complex API. And so, when it [GenAI] guesses, it’s guess is wrong a lot. Especially when developing in Java it hallucinates and makes up fields that don't belong to that object. I think it's fair to say if we were to continue to rely on generated code, I think it would create a lot of issues, and it would be a disadvantage more so than an advantage. (P-012)”

“I did just recently find a bug that I created by myself when using GenAI. It took away what I had done, and I didn’t catch it. So, it definitely can increase defects. (P-012)”

This behavior is modeled through the following equation:

GCode defect multiplier = 1 + (Gap of refinement × GCode usage ×30)

Interviewees indicated that that how much code is generated by GenAI (GCode usage), and how much it under-refined by developers (gap of refinement) together increases defect density per unit submitted code. I obtained the value of 30 in the equation based on numerical data from interviews. Participants said that when half the code was generated by GenAI, the defects increased on average by roughly 40%, even when there was no schedule pressure and developers did not have an incentive to skip refinement of the generated code before submitting it for QA. I reverse engineered the value of the Gap of Refinement variable for such a situation using the previous equations of Needed Refinement and Gap of Refinement. Plugging in the values of GCode = 0.5 and Schedule pressure = 0, the Average Needed Refinement was calculated as 0.25, and the Gap of Refinement variable was 0.025. Thus, when Gap of Refinement = 0.025 and GCode =0.5, the GCode Defect Multiplier variable has a value of 1.4 (40% increase in defects). This gives us the value of 30, which needs to be multiplied by the GCode usage and Gap of Refinement in the GCode Defect Multiplier equation.

Thus, given that generated code has more defects, developers need to spend more time on rework and code integration activities to fix the under-refined generated code in the future. Moreover, QA engineers may need additional testing cycles to uncover and correct these defects, increasing the overall cost and duration of the project.

In agile environments, frequent testing and continuous integration can help catch these additional defects sooner but also requires more developer time to be allocated to fixing these defects. Thus, the short-term productivity gains from GCode generation can be eroded if QA and acceptance testing reveal numerous defects that force developers to increase rework effort.

VI.5 Summary of the GenAI Impact on Software Development Subsystem

The GenAI Impact on Software Development subsystem offers valuable opportunities to boost productivity in the short term, while also posing risks that may affect long-term quality and performance.

Short-Term Benefits: Accelerated new feature development for both rookie and pro developers

Long-Term Considerations: When refinement of generated code lags, technical debt can build up, leading to higher code complexity, faster design quality degradation, and increased defect density. Over time, this can lead to significant rework and refactoring, which may reduce the intended productivity gains.

A summary of the outcome variables capturing the impact of GenAI on other subsystems within the system dynamics model of agile software development is presented in Table 9. The effects of GenAI on the software development subsystem are closely linked to software production, refactoring and design quality, as well as QA and acceptance testing. This subsystem incorporates a balancing feedback loop (Section 5.2) driven by the additional refinement required for generated code, and a reinforcing feedback loop (Section 5.2) triggered by the accumulation of complexity, defects, and accelerated quality degradation. In instances where generated code is not sufficiently refined, increased rework and refactoring become necessary. Consequently, this subsystem highlights the importance of managing both refinement and quality enhancement efforts to sustain productivity gains in the near term without compromising long-term software quality.

Table 9. Variables in the GenAI subsystem impacting other subsystems

Outcome Variable	Definition	Impacted Subsystem	Value Range / Type
GenAI productivity multiplier pros	Factor affecting experienced developers' productivity.	Software production	Continuous fraction (≥1)
GenAI productivity multiplier rookies	Factor affecting less-experienced developers' productivity.	Software production	Continuous fraction (≥1)
GenAI quality degradation rate multiplier	Multiplier showing the increased rate of design quality degradation as unrefined AI-generated code is added to the codebase.	Refactoring and quality of design	Continuous fraction (≥1)
GenAI complexity multiplier	Multiplier representing increased complexity of unrefined AI-generated code.	Refactoring and quality of design	Continuous fraction (≥1)
GenAI defect Multiplier	Multiplier representing increase in defects in unrefined AI-generated code.	Quality assurance and acceptance testing	Continuous fraction (≥1)

VII Model Validation

Model validation focuses on establishing sufficient confidence in a model's structure and behavior so that its results can be trusted (Forrester 1961). In system dynamics, both structure tests (does the model's structure accurately represent the real-world systems' causal relationships?) and behavior tests (does the model reproduce the qualitative and quantitative patterns observed in real-world systems?) are used to validate a given model (Barlas 1996, Fang et al., 2018). No single test can fully validate a model, so multiple tests are combined to build confidence in the model's usefulness and relevance.

In the subsequent sections, I summarize how I validated both the structure and behavior of the system dynamics model described in Section 6.

VII.1 Behavior Validation - The TradeSoft Project (Release 2)

Behavior validation will ensure that my system dynamics model's outputs can replicate the dynamic patterns of a real-world software development project. I perform the behavior validation using data from one release of a real-world agile project (alias "TradeSoft"). I collected several key details for the TradeSoft project's release 2 (henceforth called TradeSoft R2), including the team composition, estimated versus delivered feature stories by each sprint, completion trends, productivity, defect and tech debt counts, etc. I then parameterize my system dynamics model by adding a small subset of the collected information into the model and then run the simulation. Finally, I compare the outcomes of the Tradesoft R2 project and the dynamic patterns they exhibit to those simulated by the model to validate its behavior.

VII.1.1 *Project Background*

- **Organization type:** Software development
- **Project name (Alias)**: TradeSoft (Release 2)
- **Type**: Mobile application development
- **Purpose**: Video-for-Sale mobile application

- **Agile methodology**: Scrum
- **Start date**: 25-Sep-24
- **End date**: 22-Jan-25
- **Sprint duration**: 2 weeks (Sprint 7 was 4 weeks long due to holidays but only 2 weeks worked)
- **Number of sprints**: 8 (80 workdays)
- **Team Composition**: 4 developers, 1 QA engineer, 1 product owner
- **Technology**: React Native
- **GenAI tool used**: GitHub Copilot (Enterprise license)
- **GenAI use cases**: Coding, code documentation, unit-testing

The TradeSoft R2 project was chosen for validation as it provides the needed data. In this project the whole team was a) using agile methodology, b) utilizing GenAI tool for software development, and c) documenting detailed story-wise information for every sprint, including defects and tech debt, throughout the duration of the project.

VII.1.2 *Validation Data Collection*

Detailed project data, including individual user stories for development, defects, and tech debt-related tasks, and velocity, burndown, and sprint reports were obtained through the internal project tracking tool Atlassian JIRA (https://www.atlassian.com/software/jira) used by the organization. Additional context such as project-related decisions, schedule changes, and resource allocation came from documentation provided by the project team. Details on the project data sources used for validation of the system dynamics model are below:

Project tracking tool: I obtained detailed story-wise project data for each sprint, for the project's estimated and actual velocity, logs for development, defects, and tech debt, and burn-down charts from the organization's internal project tracking tool, Atlassian JIRA.

Interviews: Semi-structured interviews with the project team provided context about sprint scope changes, unit-testing and refactoring strategies, and how GitHub Copilot assisted in software development-related tasks. The development team lead and product owner were interviewed for project details.

Sprint planning and review documents: Sprint-wise project-related documents such as sprint planning, sprint review, and sprint retrospective documents captured how the team adapted scope changes and handled unexpected issues, including holiday season delays in Sprint 7.

Next, I parameterize the system dynamics model for this project's environment.

VII.1.3 *Model Parameterization*

To simulate the TradeSoft Release 2 project, I specify four main groups of parameters: a) software development environment, b) planning environment, c) initial project estimates, and d) GenAI usage. The parameters from the first three groups are required to be set before running any project simulation of the baseline system dynamics model (Cao et al., 2010) as they are an integral part of the model structure and impact the simulation outcomes. Parameters related to GenAI usage are required to instantiate how and to what extent GenAI tools were used in the project. The parameters for TradeSoft Release 2 (R2) are obtained directly from or calculated using the project data obtained from sources previously mentioned.

VII.1.3.1 Software Development Environment Parameters

Baseline Nominal Potential Productivity of Pros and Rookies:

Nominal potential productivity is the productivity metric that is established for specific development environments, such as software tools, programming languages, hardware characteristics, and product complexity (Abdel-Hamid and Madnick, 1991). The value of nominal potential productivity corresponds to the potential productivity of each pro developer in the team whereas the potential productivity of rookies is considered half of this value (Abdel-Hamid and Madnick, 1991). It remains constant during the development process of the agile project used for validation purpose.

Since I am modeling the productivity boost to pros and rookies caused by using GenAI tools separately in the GenAI impact on software development subsystem (Section 6), I calculate the nominal potential productivity variable for a baseline scenario in which the team has not used GenAI tools. Thus, the baseline nominal potential productivity of developers in the TradeSoft R2 team is calculated based on the nominal potential productivity of developers in an agile project of similar scrum team size, project duration, and complexity, at the same organization, but where the project team has not adopted GitHub Copilot or any other GenAI tools yet. Such a baseline project (alias "BaseSoft") was identified. It had the same team size and composition of 4 developers, 1 QA engineer, and 1 product owner (a standard project team setup for this organization). This team was also consisting of all experienced developers (pros) working in a different module in the same product-suite in a similar type and complexity of a product release that lasted 8 two-week sprints. Sprint-by-sprint data of BaseSoft shows that a total of 238 story points were developed and delivered across 8 sprints, each lasting 2 weeks. Given the project's load factor of 0.4 (meaning developers only worked on the project-related tasks for 40% of total person-hours), the total person-days were 0.4*320 = 128. Therefore, the average actual productivity is 238/128 = 1.86 story points/person-day.

Thus, the baseline nominal potential productivity for this project that includes the multiplier due to motivation (0.6) and communication loss (0.02) (Abdel-Hamid and Madnick, 1991; Cao, 2005), is 1.86/(0.6*(1-0.02)) = 3.16 story-points/person-days.

Enhanced Nominal Potential Productivity:

The "Enhanced nominal potential productivity" variables for pros and rookies are calculated by multiplying their respective baseline nominal potential productivity with their respective GenAI productivity boost multipliers.

Baseline Defect Rate (Assuming No Generative AI usage):

Similar to the baseline nominal potential productivity variable, the baseline defect rate is also calculated for the Basesoft project. The 'Adjusted defect rate' then measures how the Across the 8 sprints, defects worked totaled 51 story points for 155 story points of development work. Therefore, the defect rate was 51/155 = 0.33 defect story points per developed story point.

Enhanced Defect Rate (Assuming Generative AI usage):

The "Enhanced Defect Rate" variable is calculated by multiplying the baseline defect rate with the GenAI Defect Rate Multiplier.

Developer Manpower Allocation for Design Analysis, New Feature Development, and Unit-Testing:

Like most Scrum teams, the TradeSoft R2 development team began development with minimal up-front analysis of software design, as it was performed by the product owner before sprint 1. Being 2nd of 4 releases, it was assumed that 25% of the initial software design was known to the product owner as release 1 was completed prior to start of release 2. Roughly 80% unit-test coverage was expected from developers before they submitted their developed code for QA. This 80% of unit-test coverage translated to 20% of development work allocated to unit-testing by the developer, according to the interviews. The unit-testing effort percentage is much less than what is mentioned in prior work (Cao et al., 2010 used 40%) as developers in the modern day use tools built in the IDE for unit-testing, making the process of generating unit-test cases much faster. Thus, the developer spends 80% of development time on developing new features, 20% in unit-testing the developed features, and no time (0%) on the analysis of software design.

QA Manpower Allocation:

Testing was performed throughout each sprint on the delivered features. Because of one dedicated QA engineer among 5 total team members, QA manpower allocation was set to 20%.

Iteration Duration:

Sprints were 2 weeks (10 workdays) each, except Sprint 7 which was considered 4 week-long due to holidays, however worked only 10 days. In the model, we represent the standard sprint cycle as 2 weeks so the sprint 7 considered same as other sprints.

Degree of Pair Programming:

Pair programming variable was set to 0% (the team did not adopt pair programming).

Unit Test Coverage:

Unit test coverage was planned for 80% throughout sprints. The developers rely on IDE-integrated unit testing tools and GenAI for generating unit tests for each function/module. Manual acceptance testing is then performed by the QA engineer once developers submit unit-tested code.

Degree of Customer Involvement:

The product owner acted as a surrogate customer who was fully available for the complete duration of the project, evaluating delivered features at the end of each sprint and allocating work for the next. Therefore, the customer involvement is set to 100%.

Requirements Volatility:

Interview data indicated about 20% of the requirements were expected to be revised or added after start of the project. Therefore, the value of requirements volatility parameter is set to 20%.

VII.1.3.2 Planning Environment

Maximum Tolerable Completion Date:

The team planned for 7 sprints, but indicated they could extend to 8 or even 9 sprints if absolutely necessary. I therefore set 9 sprints (90 workdays) as the maximum tolerable completion date

for the project in the model.

Maximum Workforce:

Due to budget and resource constraints, the maximum tolerable development team size for the project was the same as the initial development team size of 5 (4 developers + 1 QA engineer).

Refactoring Policy:

There was no fixed refactoring policy in place for the TradeSoft R2 project. The team created stories labeled “Tech Debt” in each sprint and worked on them in the subsequent sprints. The tech debt tickets were considered low to medium priority and were usually worked on after 2-3 sprints. However, the residual tech debt stories that accumulated towards the end of the release cycle were worked on in the last 1-2 sprints before the release was shipped. This behavior corresponds to minor (continuous) refactoring being practiced every sprint, with residual being cleared towards the end of the project. Therefore, the refactoring policy is modeled as continuous minor refactoring in the refactoring and quality of design subsystem of the system dynamics model.

VII.1.3.3 Initial Project Estimates

Initial Staffing Level:

The project started with 5 full-time development team members (4 developers + 1 QA engineer).

Initial Design Percent:

Since this is release 2 of 4, it's assumed as about 25% of the system's high-level design architecture of the software was already established at the beginning of release 2.

Initial Estimated Velocity:

The initial estimated velocity is calculated based on staffing level, project duration, and development effort as below:

Initial Staffing Level * Initial Estimated Project Duration (workdays) / Initial Estimated Number of total story points. Initial estimate of total story points is considered as 250, given that a total of 300 story points were completed, 20% of which were unknown at the beginning. Therefore, initial estimated velocity is calculated as:

5*70/250 = 1.4 Person-days/story.

In early planning, the team anticipated roughly (derived from prior release data).

Ratio of Pros:

All developers on the project were considered proficient in React Native and in the software product domain, having 3+ years of industry experience. Therefore, the ratio of pros variable is set to 1 (or 100%).

VII.1.3.4 GenAI Usage

Based on interviews**,** it was found that the team used GitHub Copilot to generate roughly 30% of all code lines (GCode usage) in the project. Github Copilot was also used for documentation and unit-testing support. Because the model focuses on the GenAI use cases of code generation, unit testing, and documentation, GitHub Copilot's usage is treated as exogenous constants with the values GCode = 0.3; GDocument = 1; GTest = 1.

By incorporating these parameters into the agile simulation model, I aligned the system dynamics model's structure with that of the TradeSoft R2 project. Simulated model outputs (e.g., development velocities by sprint, defects accumulated and worked, tech debt accumulated and worked, etc.) were then compared to the sprint-by-sprint project data to validate the model behavior. The results are shown in the next section.

VII.1.3.5 Summary of TradeSoft R2 Model Parameters

The TradeSoft model parameters are summarized in Table 10 below.

Table 10. TradeSoft project parameters (table adapted from Cao, 2005)

Parameter Name	**Units**	**Value**
Nominal potential productivity	Development story points/person-day	3.16
Error rate	Defects/developed story point	0.33
QA Allocation	%	20
Iteration (sprint) duration	Weeks	2
Refactoring Policy		Continuous
Initial estimated project size	Story points	238
Initial estimates of project duration	Workdays	70
Initial staffing level	Persons	5
Initial Estimated Velocity	Person-days/story point	1.4

VII.1.4 *Actual vs. Simulated Project Behavior*

The model is configured with initial parameter values from TradeSoft Release 2 project as specified in Table10 and simulations are performed. The model's output values are compared with TradeSoft Release 2's actual values. The behaviors of the following four project variables in the actual and simulated projects are compared:

1. Features delivered by sprint (new feature development + rework story points delivered by sprint)

2. Rework (defect story points resolved by sprint)
3. Refactoring (story points related to tech-debt resolved by sprint)

Cumulative work done by task type (new feature development, rework, and refactoring).

VII.1.4.1 Development Productivity (Features Including Rework)

This variable captures how much new functionality is delivered and includes the newly developed features in the current sprint that are accepted by product owner after passing QA, and the rework items from features developed in the last sprint that did not pass QA due to defects identified. In both actual and simulated data (Figure 11), development productivity followed a U-shaped pattern, reflecting periods of high new feature development at the beginning and end, with refactoring being performed in the middle sprints to address accumulated technical debt exasperated by the use of GenAI-generated code.

The interesting feature in both real and simulated results is that using GenAI for coding yielded short bursts of high productivity in the initial sprints (due to high speed of development) and final sprints (due to high schedule pressure), along with productivity dips in the middle sprints, when the team must address the software quality degradation or tech-debt introduced by partial or inadequate refinement of generated code.

Overall, the short-term bursts in feature delivery appear in both the simulation and the real project's data. The model slightly underestimates delivered features near the end of the project, as the real team worked overtime and delivered more features in order to meet the deadline, moving any known issues (leftover tech-debt and defects) to the next release.

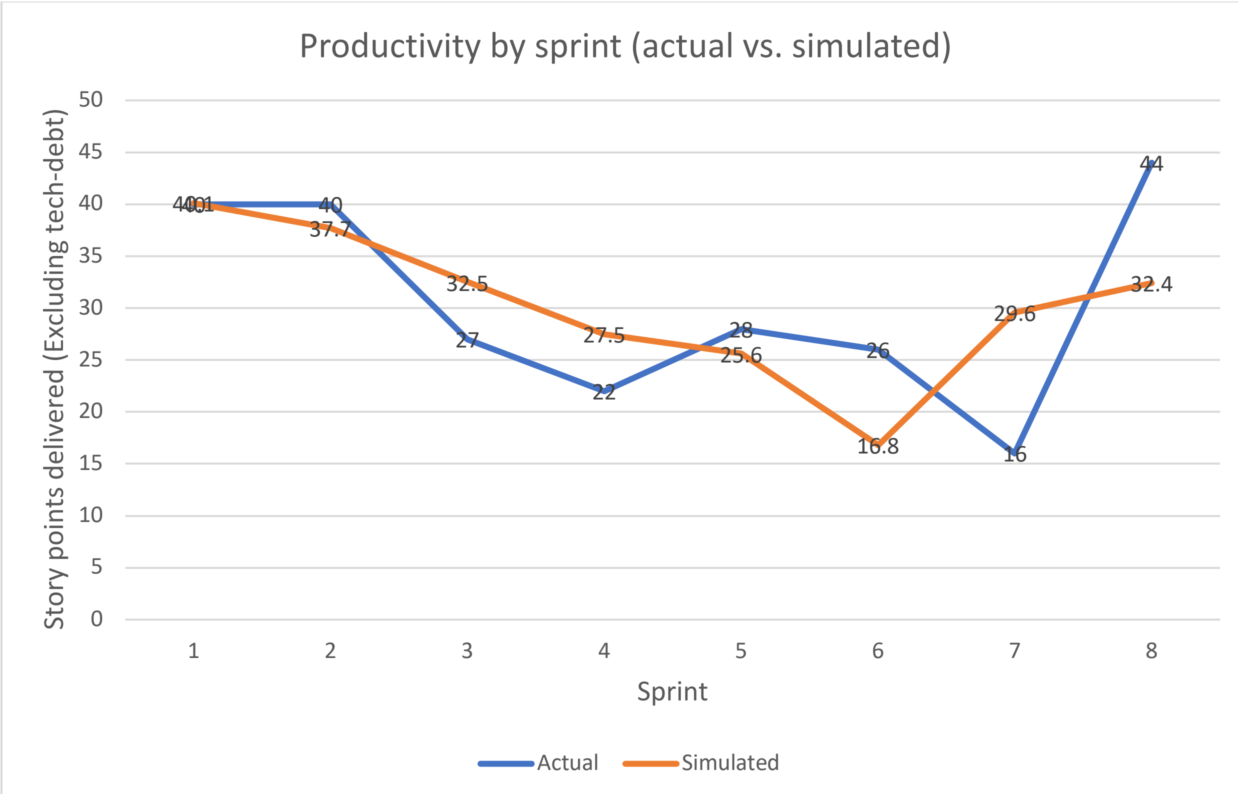

Figure 10. Story points of feature delivery (new feature development + rework) by sprint

VII.1.4.2 Defects Resolved

Both the actual and simulated projects exhibit a cyclical pattern of defect resolution (Figure 11), that follow the spikes in coding activity at the beginning and end of the project. Periods with intense coding see a jump in defects that are resolved in the next sprint.

In general, the simulation overestimates the defect rates slightly (simulated defects resolved = 85 vs. 74 actual). This discrepancy in my opinion is largely attributable to the interviewed developers underestimating the additional defects created due to GenAI. However, the actual project data indicates that the additional defect story points are roughly 7 more than in the simulation model over the course of 8 sprints. Considering each defect as roughly 3 story points, the simulation model underestimates the number of defects created by GenAI by 2.

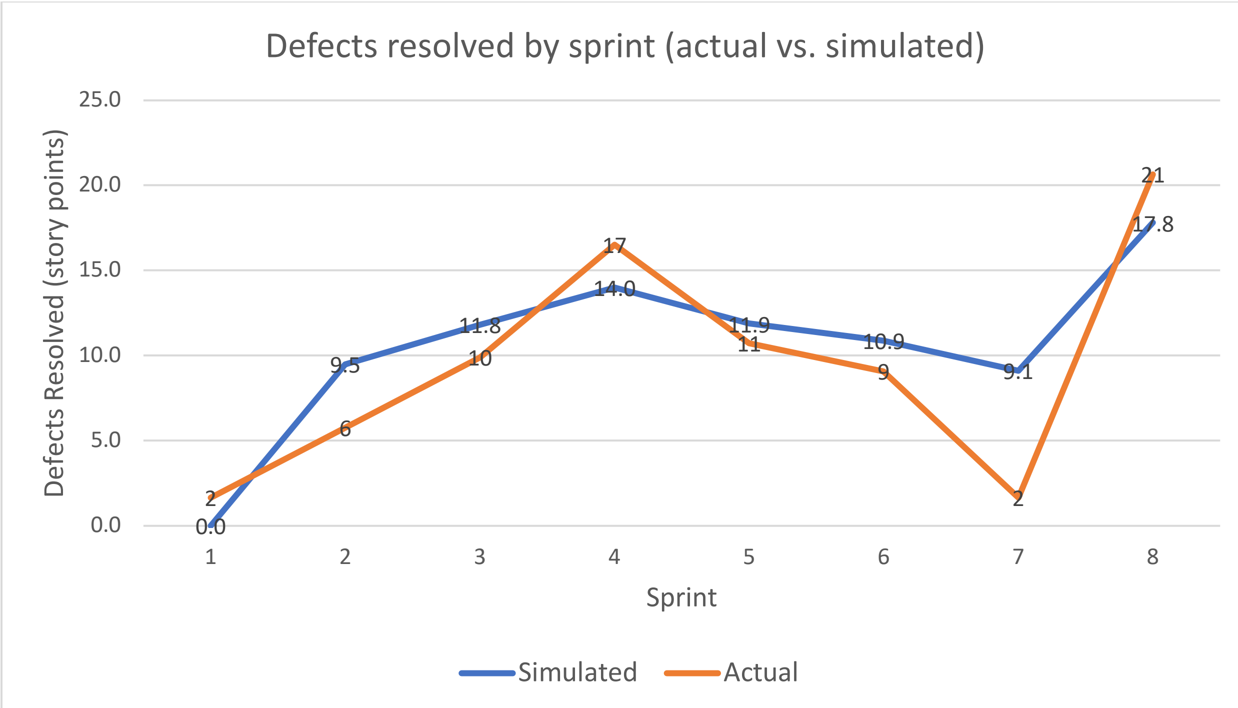

Figure 11. Story points related to defect resolution by sprint

VII.1.5 *Cumulative work done*

The % time allocated to new development, rework, and refactoring are compared between the actual and simulated project in Table 11. We see that the model slightly underestimates the time spent on new development (52.88% simulated vs. 55.67% actual), and overestimates the time spent on rework (28.35% simulated vs. 25.33% actual). This may be because the GCode defect multiplier was overestimated in the model development phase. Developers are skeptical towards GenAI-generated code in terms of defects and may have overstated the increase in defects it introduces. Whereas the model closely predicted the time spent on tech-debt (18.78%) compared to the actual project (19%).

Table 11. Cumulative work done actual vs. simulated

Work type	Person-days (Actual project)	%	Person-days (Simulated project)	%
All	320	100.00%	320.0	100.00%
New development	178.13	55.67%	169.2	52.88%
Rework (Fixing defects)	81.07	25.33%	90.7	28.35%
Refactor (Removing TD)	60.8	19.00%	60.1	18.78%

These comparisons reveal that the model successfully reproduces the major dynamic patterns observed in the real project's data. Minor deviations stem from differences between the modeled and actual increase in defects generated by GenAI-generated code. Overall, the model captures the essential behaviors that arise from integrating GenAI tools in agile software development projects.

VII.2 Structure Validation

In addition to validating the model's output, I conducted structure validation to verify that the model's equations, relationships, and assumptions are aligned with real-world agile software development (Sterman, 2000).

To assess the structure of the model, I have reviewed the feedback loops and the model structure with two domain experts who have expertise in agile software development and GenAI. These professionals were not interviewed in model construction phase. They are interviewed after model is developed to seek feedback on the model structure and relationships. Details of the experts including their education, title, current role, and experience in agile software development and GenAI are summarized in table 12. Reviewing with multiple experts helps expose any biases in the model development and refinement (Cao et al., 2010).

Table 12. Validation Experts Background Summary

	Expert 1	**Expert 2**
Industry	Consulting Services	Information Technology
Education	B. S. in Computer Science	B.S. in Computer Science and MBA
Title	Agile Consultant	VP of Software Development
Role	Coaching and consulting for agile software development teams	Managing agile software development teams and projects
Experiences in software development	30+ years	20+ years
Experience in agile methods	15 years	10 years
Familiarity with agile methods	Expert in agile methodology. Works for a consulting company specialized in agile transformation/coaching.	Very familiar, has managed agile software development projects and teams at multiple technology companies.
Familiarity with GenAI	Familiar; has been consulting on the integration of GenAI.	Familiar; introduced GenAI tool for software development.

An overview of the research question, causal feedback loops, dynamic hypothesis, system dynamics, and model structure was provided to the agile development experts prior to the interview so that they can help validating the model. The validation was done with these experts virtually over a Zoom meeting lasting about 60 minutes. The experts probed into the model and validated the dynamic hypotheses, feedback loops, and the GenAI impacts. They were asked if the model captured the appropriate variables and their relationships to accurately reflect the impact of GenAI on agile development outcomes. The experts indicated that the variables, relationships and strength of the relationships were in line with their understanding and experience of agile and GenAI. There were no discrepancies and/or anomalies surfaced in the review of the model structure with the experts.

For a thorough validation of the model, all relevant tests are conducted as described in Table 6. The summary of tests and results are listed in Table 13.

Table 13. Tests for model validation

Test	**Purpose**	**Procedure**	**Results**
1. Boundary Adequacy	Ensure important concepts are included	1. Domain expert reviewed feedback loops and model subsystems 2. Literature review	The model boundary deemed adequate
2. Structure Assessment	Check model logic against domain knowledge	Domain expert reviewed variables and relationships	Passed
3. Dimensional Consistency	Verify each equation is dimensionally sound	1. Used Stella's dimensional check utility 2. Manually inspected the model equations	Passed
4. Parameter Assessment	Confirm parameter values are realistic	Every variable in the model was derived from literature, interviews, or real project data	Passed
5. Extreme Conditions	Evaluate if model behaves reasonably under extreme inputs	1. Inspected each equation 2. Tested response to extreme values of each input	Passed

6. Integration Error (N/A)	Confirm stability of solutions under different time steps	Time-step set to daily basis	Not performed
7. Behavior Reproduction	Check if model reproduces main behaviors in real project data	Compared outputs with an actual agile project's GenAI usage and dynamic patterns	Performed well
8. Behavior Anomaly	Assess if removing key relationships causes random behavior in the model	Removed AI-related refinement loops, model exhibited anomalous behavior	Performed well
9. Family Member	See if model can represent other systems of the same class	This model is enhanced for a specific class of systems: GenAI-impacts on agile software development	Not applicable
10. Surprise Behavior	Identify unexpected behaviors	No surprising or contradictory behaviors found	Performed well
11. Sensitivity Analysis	Explore changes in model due to parameter variations	Univariate sensitivity analysis is performed on GCode, GTest, GDocument values.	Passed
12. System Improvement	Evaluate how modeling improved the real system	Model improvement is not the focus of this research	Not applicable

The model successfully passed the structural validation tests, including boundary adequacy, dimensional consistency, parameter assessment, extreme conditions, behavior anomaly, and sensitivity analysis. Domain expert reviews confirmed that the variables and causal relationships in the model appropriately reflect real-world GenAI-enabled agile software development. Thus, I conclude that the model is structurally sound and capable of replicating the key behaviors of interest.

VIII Model Exploration

Having verified that the model is structurally valid and capable of reproducing observed behaviors in a real-world agile project using GenAI, I now explore the broader implications of GenAI-enabled agile software development. This chapter demonstrates the model's utility as an experimental tool for investigating the interplay of critical variables such as GenAI usage levels, refinement effort, schedule pressure, refactoring policies, etc.

VIII.1 Model Behavior

A typical mid-sized agile project scenario was simulated to illustrate key dynamics. Major behaviors observed in the model include:

1. Non-Linear Productivity Gains: Initially, GenAI usage accelerates development, but insufficient refinement of generated code degrades design quality, increases accumulation of tech debt and defects, causing more refactoring and rework, which slows down velocity of new development in later sprints.

2. Importance of Generated Code Refinement: GenAI-generated code requires sufficient refinement by developers during the coding phase to mitigate its potential negative long-term implications such as high defect count and tech debt accumulation.

3. Continuous Refactoring Needs: Regular refactoring typically keeps design quality in check, but if tight deadlines compel developers to skip refinement of generated code, technical debt accumulates faster, and a major refactoring is required after a few sprints, that disrupts development. If no refactoring is done, to compensate for insufficient refinement of generated code, integrating newly written code into the codebase becomes difficult, which slows down new development velocity in later sprints.

4. Increased Defect Generation: Defect rates spike during intensive coding phases when new code is generated with the help of GenAI tools. Higher refinement effort to improve the generated code by developers helps mitigate these spikes.

Overall, simulations show how generative AI can offer significant short-term acceleration while introducing new challenges: (1) a need for immediate refinement of generated code in the coding phase, and (2) a higher potential for technical debt and defect accumulation due to insufficient refinement.

VIII.2 Examining the Impact of Key Variables

Using the model as a virtual laboratory, I systematically vary key variables to analyze their effects on the software project outcomes (e.g., development productivity, tasks delivered, defects generated, tech-debt accumulated, software quality, amount of effort allocated for feature development, rework, and refactoring, etc.).

My simulation shows that as the proportion of code generated by GenAI (GCode usage) increases from 0% to 100%, productivity (measured in story points per person-day) increases. However, the accumulation of technical debt and defects also accelerates since the generated code is not sufficiently refined in the coding phase. Under moderate or low schedule pressure, consistently high refinement can keep technical debt and defects manageable. Conversely, when deadlines loom, developers are tempted to submit under-refined GenAI-generated code, which spikes defect rates and leads to the faster accumulation of tech debt.

The impact of different values of GCode usage (0%, 25%, 50%, and 75%) on productivity (story points/person-day) is shown in Figure 12. In the initial stages of the project, the productivity is higher for higher values of GCode. However, productivity starts declining in all cases almost immediately as technical debt and defects accumulate. This decline in productivity is faster for higher values of GCode usage, as the software quality degradation rate and defect generation rate are proportional to GCode usage, and improving software quality and fixing defects takes developer effort. The decline in productivity continues until a major refactoring is performed by the team. Following a major refactoring, the productivity in all conditions starts stabilizing with a slight decline compensated by continuous minor refactoring. However, this time, if developers spend even less time on the refinement of generated code than before faced by higher schedule pressure, the productivity decline will get steeper. Thus, higher GCode usage leads to faster decline in code quality and more defects.

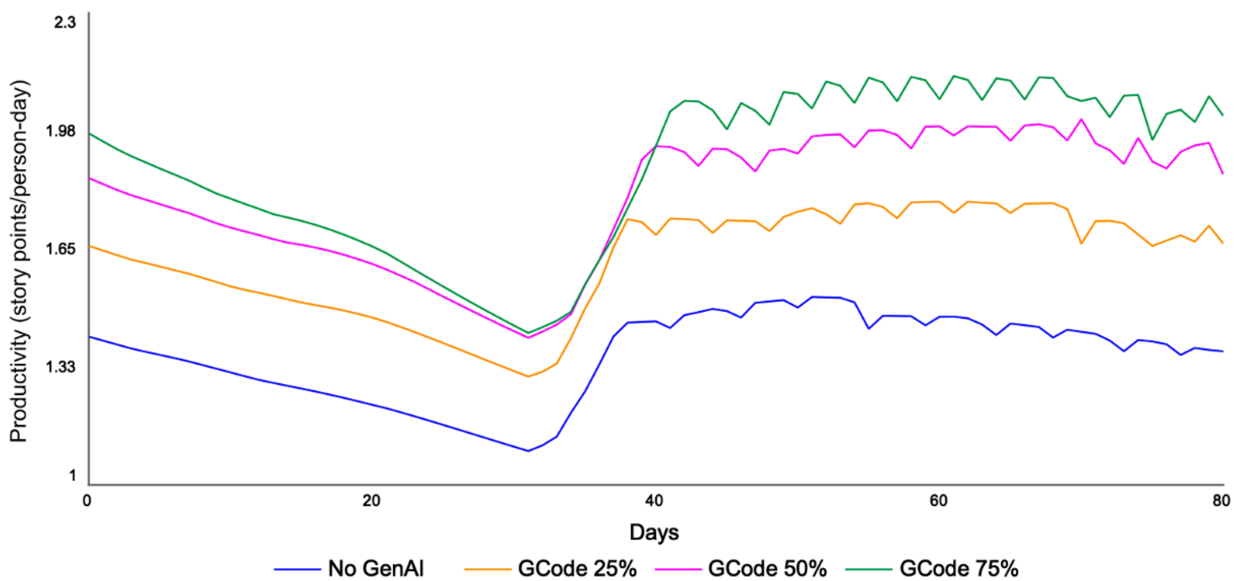

Figure 12. Productivity over time with different levels of GCode.

Setting the values of GTest and GDocument variables to maximum (100%), which means that if all unit test cases and code documentation are generated by GenAI tools, the overall productivity is enhanced without having any additional ill-effects on defect generation and tech-debt accumulation.

The productivity impact of GCode usage of 50%, in combination with GTest and GDocument usage of 100% is shown in Figure 13. When GTest and GDocument are set to maximum (100%), that is, when all unit-test cases and code documentation are generated with GenAI tools, the overall productivity increases marginally. The same behavior is observed with GCode usage of 25% and 75%.

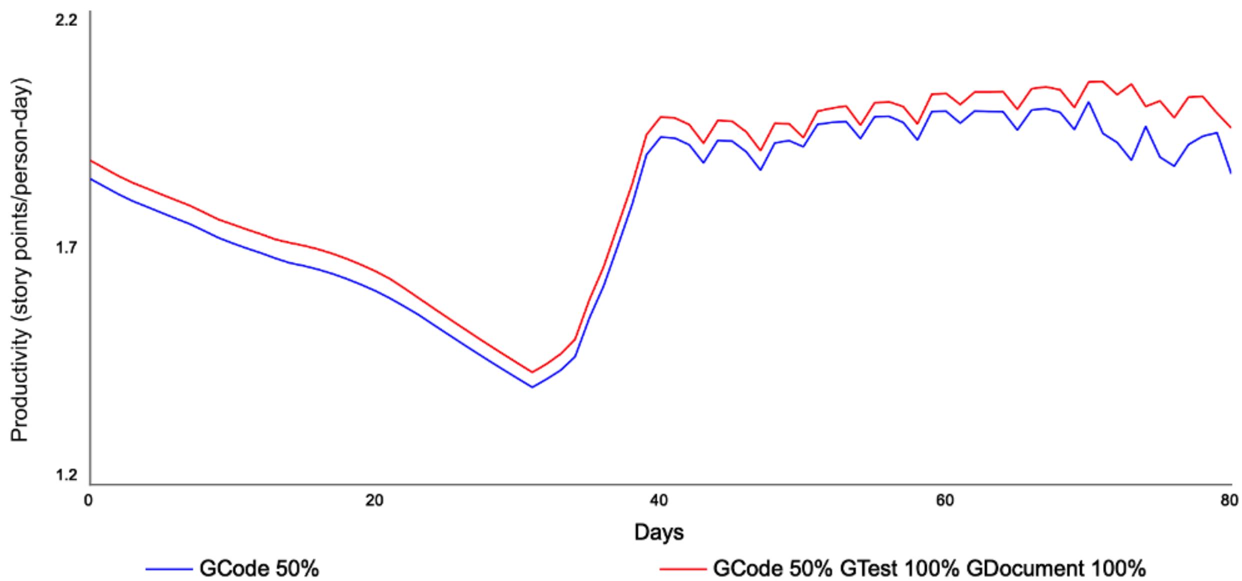

Figure 13. Productivity over time using GCode, GTest and GDocument

The impact of GCode on quality of software design is shown in Figure 14. The quality started degrading over time, however, when a major refactoring is done the quality came back to the initial level (Cao et al., 2010). It started degrading again in later sprints and is regulated by continuous refactoring. This reflects a higher amount of effort needed for refactoring generated code, to keep the software design quality at acceptable levels.

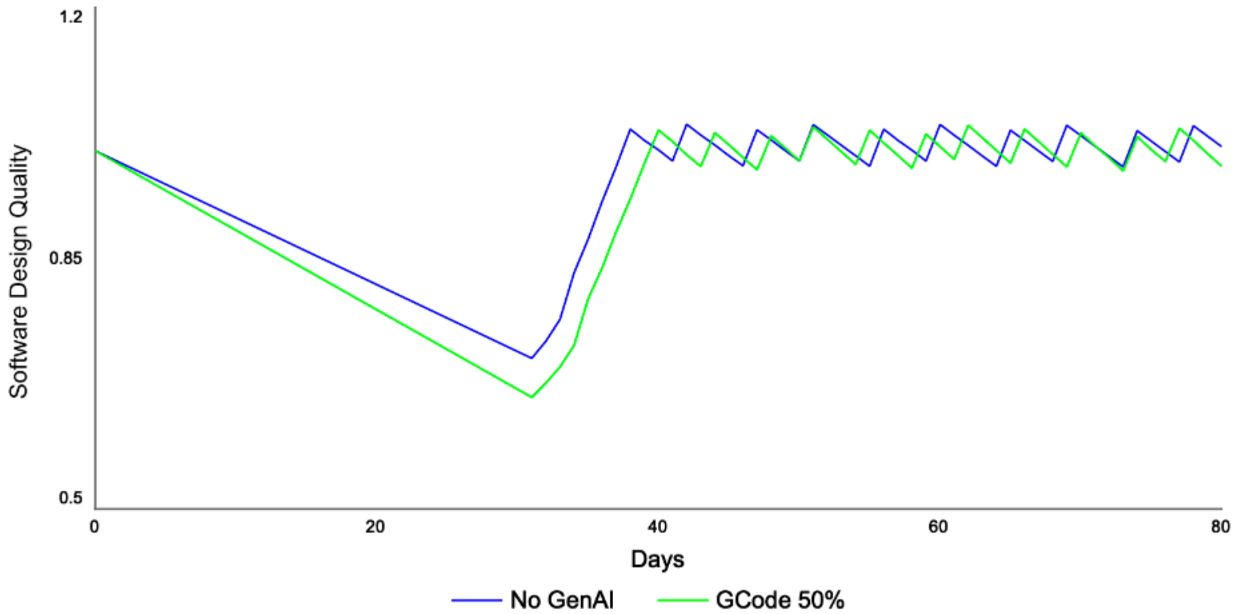

Figure 14. Quality of software design over time

The impact of GCode on the number of defects created in each sprint (10 workdays) is shown in Figure 15. The number of defects created increases over time since as the software becomes large and complex, it gets more difficult to write new code that seamlessly integrates with the codebase effectively. The simulation suggests that the increase in defects over time is much higher when 50% of the code is generated by GenAI, compared to when 0% of the code is generated. This is because under-refined generated code contains more defects compared to human-developed code.

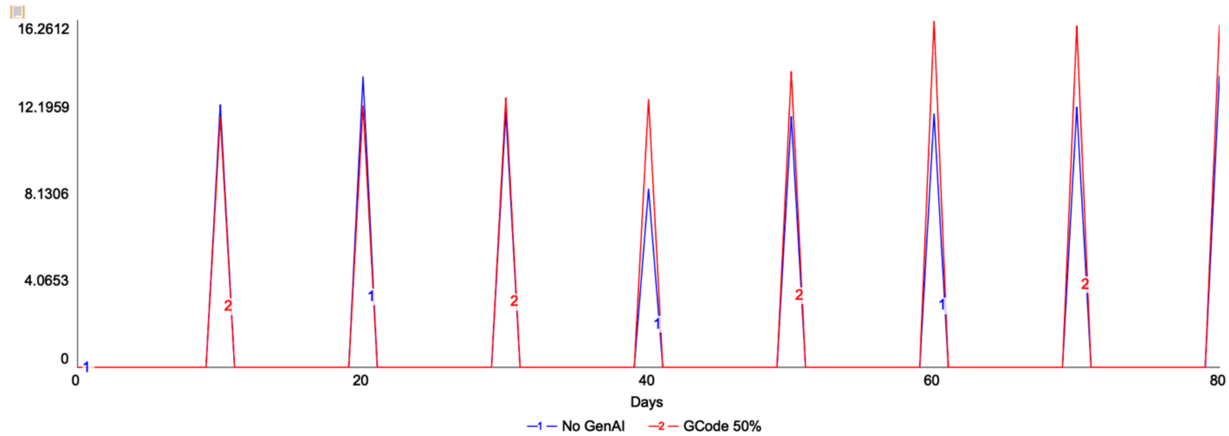

Figure 15. Errors generated per task last iteration

The effort reallocation from new feature development to rework and refactor, when GCode increases from 0% to 50%, is shown in Figure 16. As more code is generated using GenAI, more developer effort is needed to fix defects and clear the accumulated tech debt arising from under-refined generated code. Thus, more effort needs to be spent on rework and refactoring, especially in later sprints when schedule pressure is high, and the generated code is even less sufficiently refined. This results in the significant reallocation of developer effort from new feature development tasks to rework and refactoring.

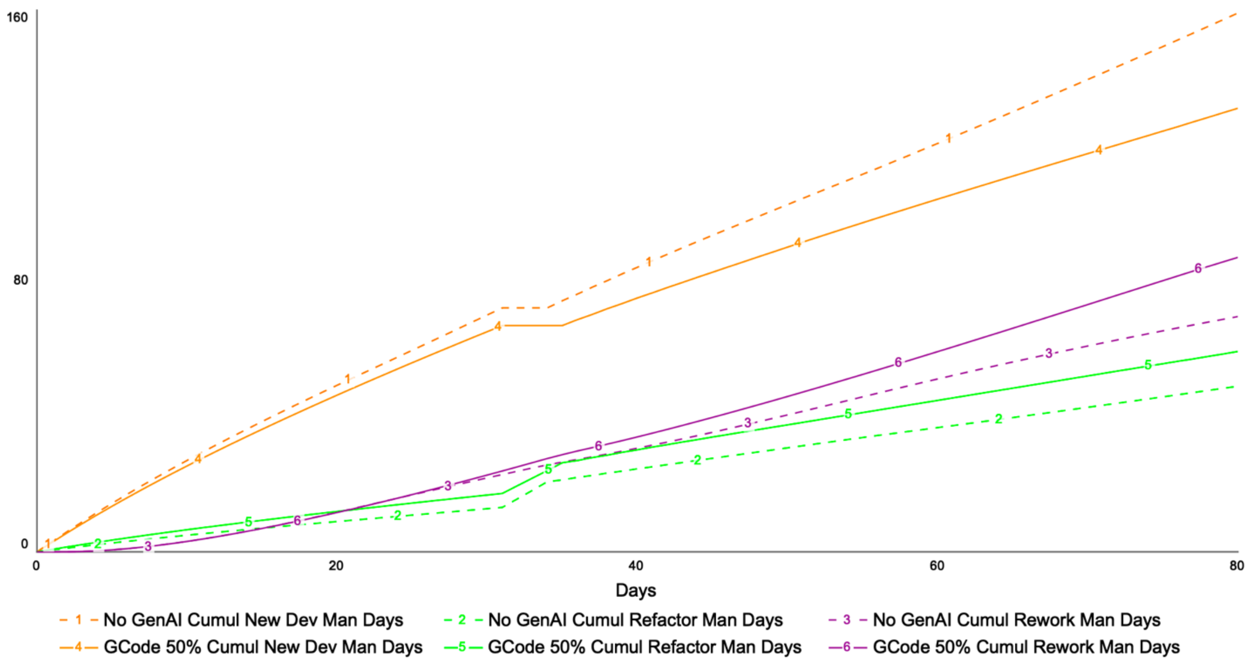

Figure 16. Effort reallocation With Use of GCode

The impact of productivity when the composition of pros and rookies are changed is shown in Figure 17. As expected, since pros experience a higher productivity boost compared to rookies, at 50% GCode use, the team composition of 100% pros resulted in highest productivity over the course of the project. This behavior aligns with interview data where participants said that pro developers experience the highest boost from GenAI. However, it is interesting to note that the major refactoring in the 50% pro-50% rookie condition happened one sprint after the 100% rookie condition. Thus, the productivity drops in the 50% pro-50% rookie condition due to refactoring leads to almost similar levels of productivity compared to the 100% pro condition during this time.

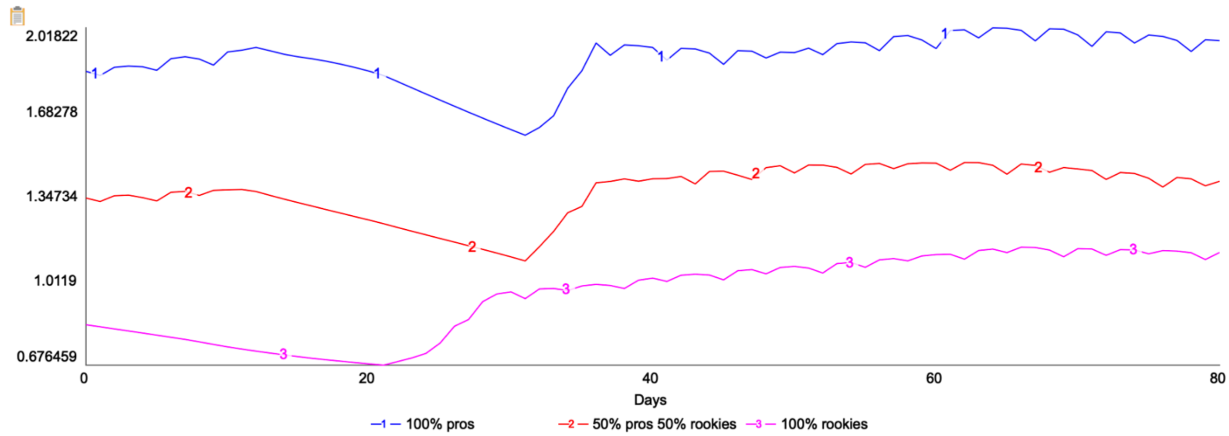

Figure 17. Productivity of Pros and Rookies (for GCode usage 50%) over time

VIII.3 Summary of Model Exploration

To explore model behavior under different conditions, I systematically varied key model parameters such as the proportion of code that is generated (GCode usage) and team composition to understand their impact on project outcomes. These outcomes include development productivity, defects generated, refactoring effort, software design quality, and the allocation of effort between feature development, rework, and refactoring. The simulation reveals that increasing GCode usage initially boosts productivity (measured in story points per person-day). However, if teams do not adequately refine the AI-generated code, the rate of defect creation and the accumulation of technical debt increase. Under moderate or low schedule pressure, teams can maintain disciplined refinement and keep defects and technical debt in check. In contrast, looming deadlines push developers to submit under-refined code, which inflates defect rates and accelerates technical debt accumulation in the later stages of the project, ultimately eroding productivity gains from GenAI.

In conclusion, the model serves as a decision-support tool for managers and practitioners to better understand the dynamic interplay between generative AI and agile development activities. It enables experimentation with alternative project-related variables to identify approaches that maximize short and long-term software development productivity and quality.

IX Conclusions and Future Research

IX.1 Discussion

The rapidly evolving landscape of software development powered by GenAI tools presents both opportunities and challenges for agile teams. My research examines this tension through a system dynamics (SD) approach. The study answers the research question as follows: Generative AI tools, such as coding assistants, test case generators, and documentation tools, can accelerate development in agile environments. However, this study finds that their impact on productivity and software quality is far more nuanced. While GenAI-generated outputs may speed up task completion, insufficient refinement, particularly under time pressure or by less experienced developers, leads to increased code complexity, defects, and the accumulation of technical debt.

Importantly, this study highlights a distinct and underexplored source of technical debt. Unlike traditional technical debt arising from poor design decisions, lack of documentation, or postponed refactoring, the debt observed here originates from GenAI-generated code that is not adequately refined or adapted to the project context. This "refinement debt" is subtle and often invisible in fast-paced agile workflows focused on feature delivery. Yet, it can degrade maintainability, hinder adaptability, and ultimately slow future development cycles.

Moreover, the effects of GenAI use are shaped by a dynamic web of interacting factors, including developer experience, the volume and nature of generated code, code complexity, refinement effort, defect occurrence, and schedule pressure. The interactions of these factors produce compounding effects over time, making software development outcomes difficult to predict. The simulation demonstrates that GenAI's benefits are highly contingent: teams that manage these interactions strategically can harness its potential, while others may inadvertently trade short-term gains for longer-term complications.

Insights from qualitative interviews and model simulations suggest that organizations can realize significant short-term gains in development productivity, but these gains are only sustainable when teams effectively manage generated code through rigorous refinement. To control the accumulation of tech debt and defects effectively, agile teams must balance their reliance on GenAI with more effort spent on refinement, refactoring, and rework activities. Another key insight from this research is the role of developer experience in benefiting from GenAI tools. While both junior (rookie) and senior (pro) developers can improve their throughput, the level of refinement required for generated code varies, suggesting that experienced developers often reap larger net productivity benefits.

Overall, this study enhances our understanding of how GenAI-driven software development fits within the dynamic environment of agile projects. By embedding GenAI use-cases into a well-established system dynamics model of agile software development, it provides a new lens for practitioners and researchers to analyze potential project outcomes, develop targeted management policies, and forecast the dynamic implications of GenAI adoption in software development teams.

The following insights were derived and summarized from this research:

1. **Significant improvement in productivity:** GenAI tools show an increase in productivity of developers in agile software development projects. Most of the interviewed participants used GenAI tools to generate code snippets, unit test cases, and code documentation, and reported an overall productivity boost for new feature development, ranging from 10% to 50% depending on the extent to which they used GenAI.
2. **Diminished productivity gains over time:** Evidence from interviews, model simulations, and project-level data indicates that initial productivity gains derived from GenAI use do not sustain over time as additional effort is required to refine the generated code, rework, and refactor it to address the additional defects and accumulated tech debt.
3. **Importance of refining generated code:** When code is generated using a GenAI tool, it is critical to refine the generated code to ensure it functions as expected, meets the requirements, and fits well with coding standards. Developers need to review and refine GenAI generated code with the same rigor as human developed code to ensure quality and prevent technical debt. The refinement process includes checking for code consistency and

style, readability and restructure, performance optimization, security, and best practices. Properly refined code improves code quality and maintainability, alignment with project-specific requirements, enhances performance and security, allows developers to understand the complex logic easily, and enhances code reusability.

4. **Impact of insufficient refinement:** GenAI generated code, when not properly refined, introduces additional defects and tech debt that accumulate over time in the software. This leads to more rework and refactoring activities undertaken over the course of the project, that reduce productivity of new development, as developers need to spend extra time to fix these issues introduced by generated code. Thus, insufficient refinement of generated code shifts developer effort from new development to rework and refactoring.
5. **Accumulation of duplicate and unused code (dead code):** As developers generated code frequently using GenAI tools, a larger volume of unused, duplicate or unnecessary code accumulates over time. This is because it is easier for developers to generate new code rather than finding existing code that can be reused. This results in accumulation of duplicate code, which is a component of tech debt that is difficult to address for developers during the refinement process. Thus, there is always insufficient refinement of generated code that aggravates software quality degradation and tech debt accumulation, requiring additional rework and refactoring effort.
6. **Schedule pressure impacts refinement of generated code:** As project deadlines approach and schedule pressure increases, developers tend to increase the use of GenAI and skip the refinement to complete tasks faster, exasperating the negative impacts of insufficiently refined generated code.
7. **Developers experience impacts GenAI productivity boost:** Overall productivity increase from use of GenAI is observed in both senior (pro) and junior (rookie) developers. However, senior developers derive higher productivity improvement as they use GenAI more effectively and require lower effort to refine generated code.

IX.2 Contributions

This study investigated the influence of GenAI on agile software development project outcomes by adopting the system dynamics modeling approach. By doing so, it advances the body of knowledge of how GenAI tools impact software development and offers valuable insights and guidance for practitioners to help them integrate GenAI into their software development practice effectively.

IX.2.1 *Contribution to literature*

This research contributes significantly to the literature on how GenAI impacts agile software development outcomes. Recent empirical evidence highlights the immediate productivity gains developers experience from using GenAI tools in software development, particularly in automated code generation and completion tasks (Peng et al., 2023; Nguyen-Duc et al., 2023). These studies primarily focused on individual developers' productivity in code generation tasks. For instance, Peng et al., (2023) finds that using GenAI enables developers to write code for a given task faster in experimental settings, but do not systematically explore if and how these benefits translate to real-world, team-based agile projects over time. My research extends the above work by systematically studying how GenAI impacts software development outcomes such as productivity and quality at the project level, wherein multiple actors with different roles work together as a team over a period to deliver software.

Bahi et al., (2024) and Ulfsnes et al., (2024) explored the impacts of GenAI in agile software development projects. Bahi et al., (2024) identifies pain-points in agile software development and the promise of improvements in coding productivity and quality from using GenAI in agile projects but does not provide any empirical evidence for these claims. Using qualitative interviews, Ulfsnes et al., (2024) studies how developers use GenAI in agile software development but offers no empirical evidence on its impact on project level outcomes. My research challenges the prevailing assumption in the literature that the productivity gained at the individual level will persist at the project level over time in two ways. First, I find that in a team environment, developers refine the generated code to align it with the requirements of the project. This act of refining requires time to be spent on understanding the generated code and then improving it, which reduces productivity of developers. Second, I find that if developers bypass code refinement, the generated code can increase in complexity and have additional defects, which degrade software quality. Fixing these issues at the later stages of the project leads to a loss in development productivity. Thus, the gains in development productivity witnessed among individuals, in code generation tasks do not persist in an agile project environment over time.

The issues and concerns regarding the quality and complexity of generated code are expressed in a handful of studies (Idrisov & Schlippe, 2024; Imai, 2022). These studies found that although using GenAI tools increases productivity of developer by generating code, the quality of generated code is inferior as it contains additional defects and has higher complexity than code that is purely human written. Our findings extend this literature concerning the double-edged sword of using GenAI in coding tasks by analyzing the downstream impacts of generated code on software development outcomes in agile projects over time. By integrating both qualitative and quantitative findings, this research also provides more nuanced insights which suggest how the diminished quality of generated code can be improved by refining it.

The outcome of my research is a system dynamics model of the impact of GenAI on agile software development projects. Early work on system dynamics modeling of software development (Abdel-Hamid, 1984) illustrated how workforce-related decisions, error detection and resolution activities create feedback loops in traditional, plan-driven (waterfall) software development projects. Building on this foundational work, Cao et al., (2010) adapted system dynamics to agile projects, explicitly modeling iterative releases, continuous refactoring, and frequent customer involvement. This body of work primarily treated software development as a fully human-driven process, without accounting for emerging tools that automate software development tasks such as coding, unit-testing, and documentation. Considering the recent proliferation of GenAI in software development, my work extends that body of work by explicitly modeling GenAI-specific variables (e.g., the proportion of code generated, GenAI-driven unit-testing and documentation) and the process of refining generated code. Modeling these new practices in the context of agile software development projects expands our understanding of how automated techniques interact with agile practices dynamically to impact key project outcomes such as productivity and software quality.

IX.2.2 *Contribution to practice*

This study offers multifaceted contributions to practice, benefiting not only individual software developers and managers, but also organizations as a whole. For software developers, it provides novel insights into the effective integration of GenAI tools in their day-to-day development tasks. The findings of my research not only help developers understand if and how GenAI can improve their productivity but also guide them towards sustaining these improvements over time. The findings of my study also identify challenges associated with insufficient refinement of generated code, including its impact on code complexity, quality, and rework that can diminish the gains in developer productivity. Finally, the findings offer guidance to developers on mitigating these challenges through rigorous code review and refinement practices.

For software development managers, the study provides a team-level perspective on the benefits and challenges of using GenAI in agile software development. The SD model acts as a tool enabling managers to simulate and establish policies that ensure the effective use of GenAI in agile projects. For example, the SD model can help managers determine the appropriate level of GenAI usage, find the right balance between experienced and novice developers, and optimize manpower allocation between new development, rework, and refactoring. Identifying and executing such contextually aligned and effective policies can help managers drastically increase project-level software development outcomes sustainably. My research also provides key insights into a new form of technical debt accumulated in the generated code, and the need for frequent refactoring to sustain productivity gains. Furthermore, it warns managers about the accumulation of large volumes of unused, unnecessary, and dead code through frequent code generation, and offers strategies to prevent it.

For organizations, this study provides a holistic understanding of GenAI's impact, facilitating the development of organization-wide policies and strategies that maximize its benefits while mitigating long-term challenges. A key guideline emerging from this research is that while GenAI can significantly enhance productivity, its effective use requires sufficient code refinement to prevent the accumulation of defects and technical debt, ensuring sustained long-term productivity benefits to agile software development projects. Thus, developers, managers, and organizations need to understand that in order to derive sustained productivity boosts, some of the time saved from using GenAI needs to be reinvested in refinement efforts.

Furthermore, the study highlights the critical role of experienced developers and informs how GenAI's benefits are skewed by the expertise levels of team members. It emphasizes the importance for organizations and managers to retain senior developers rather than replacing them with GenAI-augmented novice developers. The SD model can be used by managers and organizations alike to simulate the optimal team composition of experienced and novice developers that maximizes the effectiveness of GenAI while still keeping costs under control. Through these contributions, my findings serve as a practical guide for organizations, managers, and software developers seeking to integrate GenAI effectively in their agile software development projects.

IX.3 Limitations

Despite its contributions, this research has several limitations that future work can address:

1. **Scope of Model Boundaries:** The current SD model focuses primarily on project-level outcomes such as complexity, defects, and refactoring, without explicitly modeling product-

level architectural and organizational factors (e.g., incentives to use GenAI tools, GenAI usage policies for system security and data privacy, etc.).

2. **Use of self-reported data in model development:** Much of the qualitative assessment of GenAI's benefits and drawbacks is based on interviews and team-reported metrics. Although steps were taken to ensure data accuracy and integrity, self-reports may introduce biases, as seen with the participants overemphasizing the quality degradation caused by GenAI-generated code. Longitudinal real-world project data could be used to further calibrate the formulae, finetune the model, and improve its accuracy.
3. **Evolving GenAI tools and capabilities:** The model treats GenAI tools uniformly, though real-world usage spans multiple LLM-based products with varying performance, integration capabilities, domain specificity, and developer expertise with the use of GenAI tools. As the GenAI software development tool ecosystem evolves, new features (e.g., larger context windows or specialized models for domain-specific software development) and improvements in performance and reasoning capabilities of GenAI tools may alter both their short- and long-term impacts on agile projects. Moreover, as GenAI tools continually improve through training and developers learn how to effectively use and prompt them, some of the challenges observed in the study may be addressed in the future. Further studies could focus on collecting quantitative metrics to measure the strength of the impacts and calibrate the model for current capabilities of GenAI tools.
4. **Project context variation:** Agile projects differ widely in scope, domain, regulatory needs, and technological complexity. A single model cannot capture all these contextual differences. While the proposed model offers a general framework, calibrating its parameters to specific domains or larger-scale systems is essential for accurate and actionable results. This research focused on business application development and not on complex platforms development.

IX.4 Future Research

The integration of GenAI into agile software development is still at an early stage, with much still to be explored. Building on the insights and limitations identified above, the following avenues are suggested for future research:

Model extensions: Future SD models could explicitly incorporate architectural and organizational factors and knowledge management practices (e.g., developer onboarding, domain knowledge transfer) that can influence how GenAI impacts agile software development. Moreover, examining other use cases of GenAI in software development such as requirements elicitation, code explanation and summarization, training, and software maintenance may yield more holistic insights.

Many agile teams allocate some portion of development capacity for maintenance work when the product releases are shipped to customers or deployed in production. Future studies can incorporate the overlap between new development and ongoing maintenance work into the SD model, capturing more realistic project trajectories spanning over multiple releases of the software. While this study draws on interviews and single-release project data, studies spanning multiple releases would offer deeper insights into how GenAI impacts change over time.

Human–AI Collaboration Dynamics: As GenAI models become more agentic, the role of human oversight and team collaboration patterns may evolve. Future research can develop a more nuanced understanding of how delegation of tasks and responsibilities takes place between human developers and GenAI agents, and how it impacts project outcomes.

Contextualized system dynamics models: Different organizations and domains (healthcare, financial services, government) have unique requirements with respect to software quality, security, and regulatory compliance. Future studies could refine the SD model to accommodate these differences.

By extending or refining the system dynamics model presented in this study, researchers and practitioners can further understand the dynamic implications of integrating GenAI into agile software development. Such initiatives can foster a deeper theoretical understanding and yield nuanced practical strategies for effectively harnessing the power of GenAI in agile software development projects.

In summary, this research helps understand how GenAI impacts agile software development through a system dynamics approach. The study shows that using GenAI for coding, testing, and documentation can significantly enhance productivity, but only if rigorous refinement is undertaken to avoid the hidden pitfalls of quality degradation and technical debt accumulation. The SD model provides both a dynamic perspective and actionable insights to effectively incorporate GenAI in agile software development projects.

REFERENCES

Abdel-Hamid, T. K. (1984). *The dynamics of software development project management: An integrative system dynamics perspective* (Doctoral dissertation, Massachusetts Institute of Technology).

Abdel-Hamid, T. K. (1989). The dynamics of software project staffing: a system dynamics based simulation approach. *IEEE Transactions on Software engineering*, *15*(2), 109-119.

Abdel-Hamid, T., & Madnick, S. E. (1991). *Software project dynamics: an integrated approach*. Prentice-Hall, Inc.

Akkermans, H., & van Helden, K. (2002). Vicious and virtuous cycles in ERP implementation: a case study of interrelations between critical success factors. *European journal of information systems*, *11*(1), 35-46.

AlOmar, E. A., Venkatakrishnan, A., Mkaouer, M. W., Newman, C., & Ouni, A. (2024, April). How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations. In *Proceedings of the 21st International Conference on Mining Software Repositories* (pp. 202-206).

Arora, C., Grundy, J., & Abdelrazek, M. (2024). Advancing requirements engineering through generative ai: Assessing the role of llms. In *Generative AI for Effective Software Development* (pp. 129-148). Cham: Springer Nature Switzerland.

Austin, R. D. (2001). The effects of time pressure on quality in software development: An agency model. *Information systems research*, *12*(2), 195-207.

BAHI, A., GHARI, J., & GAHI, Y. (2024). Integrating Generative AI for Advancing Agile Software Development and Mitigating Project Management Challenges. International Journal of Advanced Computer Science & Applications, 15(3).

Bajaj, Y., & Samal, M. K. (2023). Accelerating Software Quality: Unleashing the Power of Generative AI for Automated Test-Case Generation and Bug Identification. *International Journal for Research in Applied Science and Engineering Technology*, *11*(7).

Bala, B. K., Arshad, F. M., & Noh, K. M. (2017). *Causal loop diagrams*. System Dynamics: Modelling and Simulation, 37-51.

Bandi, A., Adapa, P. V. S. R., & Kuchi, Y. E. V. P. K. (2023). The power of generative ai: A review of requirements, models, input–output formats, evaluation metrics, and challenges. Future Internet, 15(8), 260.

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.

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... & Thomas, D. (2001). Manifesto for agile software development. Retrieved from http://agilemanifesto.org

Behutiye, W. N., Rodríguez, P., Oivo, M., & Tosun, A. (2017). Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, *82*, 139-158.

Black, L. J., Carlile, P. R., & Repenning, N. P. (2004). A dynamic theory of expertise and occupational boundaries in new technology implementation: Building on Barley's study of CT scanning. *Administrative Science Quarterly*, *49*(4), 572-607.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., ... & Zazworka, N. (2010, November). Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (pp. 47-52).

Cao, L. (2005). Modeling dynamics in agile software development. Georgia State University.

Cao, L., Ramesh, B., & Abdel-Hamid, T. (2010). Modeling dynamics in agile software development. *ACM Transactions on Management Information Systems (TMIS)*, *1*(1), 1-26.

Charmaz, K. (2014). Constructing grounded theory.

Cheng, H., Husen, J. H., Peralta, S. R., Jiang, B., Yoshioka, N., Ubayashi, N., & Washizaki, H. (2024). Generative AI for Requirements Engineering: A Systematic Literature Review. *arXiv preprint arXiv:2409.06741*.

Chui, M., Hazan, E., Roberts, R., Singla, A., & Smaje, K. (2023). The economic potential of generative AI.

Conboy, K. (2009). Agility from first principles: Reconstructing the concept of agility in information systems development. *Information systems research*, *20*(3), 329-354.

Coyle, R. G. (1997). System dynamics modelling: a practical approach. *Journal of the Operational Research Society*, *48*(5), 544-544.

Coutinho, M., Marques, L., Santos, A., Dahia, M., França, C., & de Souza Santos, R. (2024, July). 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, K. Z., Demirer, M., Jaffe, S., Musolff, L., Peng, S., & Salz, T. (2024). The Productivity Effects of Generative AI: Evidence from a Field Experiment with GitHub Copilot.

Cunningham, W. (1992). The WyCash portfolio management system. ACM Sigplan Oops Messenger, 4(2), 29-30.

Curtis, B., Sappidi, J., & Szynkarski, A. (2012, June). Estimating the size, cost, and types of technical debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)* (pp. 49-53). IEEE.

Cusumano, M. A., & Smith, S. A. (1995). Beyond the waterfall: Software development at Microsoft. *MIT Sloan School of Management*, WP# 3844-BPS-95.

Dhyani, P., Nautiyal, S., Negi, A., Dhyani, S., & Chaudhary, P. (2024, February). Automated API Docs Generator using Generative AI. In *2024 IEEE International Students' Conference on Electrical, Electronics and Computer Science (SCEECS)* (pp. 1-6). IEEE.

Dutta, A. (2001). Business planning for network services: A systems thinking approach. *Information Systems Research*, *12*(3), 260-283.

Dutta, A., & Roy, R. (2005). Offshore outsourcing: A dynamic causal model of counteracting forces. *Journal of Management Information Systems*, *22*(2), 15-35.

Ebert, C., & Louridas, P. (2023). Generative AI for software practitioners. *IEEE Software*, *40*(4), 30-38.

Fang, Y., Lim, K. H., Qian, Y., & Feng, B. (2018). System Dynamics Modeling for Information Systems Research. MIS quarterly, 42(4), 1303-A5.

Fojtik, R. (2011). Extreme programming in development of specific software. *Procedia Computer Science*, 3, 1464–1468.

Ford, D. N., & Sterman, J. D. (1998). Dynamic modeling of product development processes. System Dynamics Review: The Journal of the System Dynamics Society, 14(1), 31-68.

Forrester, J. W. (1968). *Industrial Dynamics—After the First Decade*. *Management Science, 14*(7), 398–415.

Forrester, J. W. (1971). *Principles of systems*. Cambridge, Mass., Wright Allen.

Forrester, J. W. (1980). Information sources for modeling the national economy. *Journal of the American Statistical Association*, *75*(371), 555-566.

Forrester, J.W. (1961) *Industrial Dynamics*. MIT Press, Cambridge, Mass.

Forrester, J. W., & Senge, P. M. (1996). Tests for building confidence in system dynamics models. *Modelling for management: simulation in support of systems thinking*, *2*(414-434).

Gallivan, M. J., Eynon, J., & Rai, A. (2003). The challenge of knowledge management systems: Analyzing the dynamic processes underlying performance improvement initiatives. *Information Technology & People*, *16*(3), 326-352.

Garud, R., & Kumaraswamy, A. (2005). Vicious and virtuous circles in the management of knowledge: The case of Infosys Technologies. *MIS quarterly*, 9-33.

Glaiel, F. S., Moulton, A., & Madnick, S. E. (2014). Agile project dynamics: A system dynamics investigation of agile software development methods. *ESD Working Paper Series, MIT Engineering Systems Division.*

Guo, Q., Xie, X., Liu, S., Hu, M., Li, X., & Bu, L. (2025). Intention is All You Need: Refining Your Code from Your Intention. *arXiv preprint arXiv:2502.08172*.

Hnatushenko, V. V., & Pavlenko, I. V. (2024). THE USE OF GENERATIVE ARTIFICIAL INTELLIGENCE IN SOFTWARE TESTING. *System technologies*, *2*(151), 113-123.

Höst, M., Regnell, B., och Dag, J. N., Nedstam, J., & Nyberg, C. (2001). Exploring bottlenecks in market-driven requirements management processes with discrete event simulation. *Journal of Systems and Software*, *59*(3), 323-332.

Huang, Y., Chen, Y., Chen, X., Chen, J., Peng, R., Tang, Z., ... & Zheng, Z. (2024). Generative software engineering. *arXiv preprint arXiv:2403.02583*.

Idrisov, B., & Schlippe, T. (2024). Program Code Generation with Generative AIs. *Algorithms*, *17*(2), 62.

Imai, S. (2022, May). Is github copilot a substitute for human pair-programming? an empirical study. In Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings (pp. 319-321).

Kampmann, C. E., & Oliva, R. (2008). Structural dominance analysis and theory building in system dynamics. *Systems Research and Behavioral Science, 25*(4), 505-519.

Kokol, P. (2024). The Use of AI in Software Engineering: A Synthetic Knowledge Synthesis of the Recent Research Literature. *Information*, *15*(6), 354.

Kruchten, P., & Ozkaya, I. (2019). *Managing technical debt: reducing friction in software development*. Addison-Wesley Professional.

Larman, C., & Basili, V. R. (2003). Iterative and incremental development: A brief history. *Computer*, 36(6), 47–56.

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.

Lim, W. M., Gunasekara, A., Pallant, J. L., Pallant, J. I., & Pechenkina, E. (2023). Generative AI and the future of education: Ragnarök or reformation? A paradoxical perspective from management educators. *The international journal of management education*, *21*(2), 100790.

Luna-Reyes, L. F., & Andersen, D. L. (2003). Collecting and analyzing qualitative data for system dynamics: methods and models. *System Dynamics Review: The Journal of the System Dynamics Society*, *19*(4), 271-296.

Maruping, L. M., Venkatesh, V., & Agarwal, R. (2009). A control theory perspective on agile methodology use and changing user requirements. Information systems research, 20(3), 377-399.

Meso, P., & Jain, R. (2006). Agile software development: Adaptive systems principles and best practices. *Information systems management*, *23*(3).

Negi, K., & Kota, M. (2025). The impact of Generative AI on agile software development: A system dynamics approach. *SSRN Electronic Journal*. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5201382

Nguyen-Duc, A., Cabrero-Daniel, B., Przybylek, A., Arora, C., Khanna, D., Herda, T., Rafiq, U., Melegati, J., Guerra, E., Kemell, K.-K., Saari, M., Zhang, Z., Le, H., Quan, T., & Abrahamsson, P. (2023). Generative Artificial Intelligence for Software Engineering—A Research Agenda (arXiv:2310.18648). arXiv. http://arxiv.org/abs/2310.18648

Nguyen-Duc, A., Cabrero-Daniel, B., Przybylek, A., Arora, C., Khanna, D., Herda, T., ... & Abrahamsson, P. (2023). Generative Artificial Intelligence for Software Engineering--A Research Agenda. *arXiv preprint arXiv:2310.18648*.

Ooi, K. B., Tan, G. W. H., Al-Emran, M., Al-Sharafi, M. A., Capatina, A., Chakraborty, A., ... & Wong, L. W. (2025). The potential of generative artificial intelligence across disciplines: Perspectives and future directions. *Journal of Computer Information Systems*, *65*(1), 76-107.

Ozkaya, I. (2023). The Next Frontier in Software Development: AI-Augmented Software Development Processes. IEEE Software, 40(4), 4–9. https://doi.org/10.1109/MS.2023.3278056

Pakalapati, N., Venkatasubbu, S., & Sistla, S. M. K. (2023). The Convergence of AI/ML and DevSecOps: Revolutionizing Software Development. *Journal of Knowledge Learning and Science Technology ISSN: 2959-6386 (online)*, *2*(2), 189-212.

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*.

Quaddus, M., & Intrapairot, A. (2001). Management policies and the diffusion of data warehouse: a case study using system dynamics-based decision support system. *Decision Support Systems*, *31*(2), 223-240.

Rajbhoj, A., Somase, A., Kulkarni, P., & Kulkarni, V. (2024, February). Accelerating Software Development Using Generative AI: ChatGPT Case Study. In *Proceedings of the 17th Innovations in Software Engineering Conference* (pp. 1-11).

Randers, J. (1997). Elements of the system dynamics method. Journal of the Operational Research Society, 48(11), 1144-1145.

Richardson, G. P. (2011). Reflections on the foundations of system dynamics. *System Dynamics Review, 27*(3), 219-243.

Richardson, G. P., & Pugh III, A. L. (1997). Introduction to system dynamics modeling with DYNAMO. *Journal of the Operational Research Society*, *48*(11), 1146-1146.

Rodrigues, A. G., & Williams, T. M. (1997). System dynamics in software project management: towards the development of a formal integrated framework. European Journal of Information Systems, 6(1), 51-66.

Sahaf, Z., Garousi, V., Pfahl, D., Irving, R., & Amannejad, Y. (2014, May). When to automate software testing? decision support based on system dynamics: an industrial case study.

In *Proceedings of the 2014 International Conference on Software and System Process* (pp. 149-158).

Sauvola, J., Tarkoma, S., Klemettinen, M., Riekki, J., & Doermann, D. (2024). Future of software development with generative AI. *Automated Software Engineering*, *31*(1), 26.

Schoenenberger, L., Schmid, A., Tanase, R., Beck, M., & Schwaninger, M. (2021). Structural analysis of system dynamics models. *Simulation Modelling Practice and Theory*, *110*, 102333.

Song, F., Agarwal, A., & Wen, W. (2024). The impact of generative AI on collaborative open-source software development: Evidence from GitHub Copilot. *arXiv preprint arXiv:2410.02091*.

Stallinger, F., & Grünbacher, P. (2001). System dynamics modelling and simulation of collaborative requirements engineering. *Journal of Systems and Software*, *59*(3), 311-321.

Sterman, J. D. (2000). *Business dynamics: systems thinking and modeling for a complex world*. Irwin/McGraw-Hill, Boston.

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

Ulfsnes, R., Moe, N. B., & Stray, V. (n.d.). Transforming Software Development with Generative AI: Empirical Insights on Collaboration and Workflow.

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022, April). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In Chi conference on human factors in computing systems extended abstracts (pp. 1-7).

Weisz, J. D., Muller, M., Houde, S., Richards, J., Ross, S. I., Martinez, F., ... & Talamadupula, K. (2021, April). Perfection not required? Human-AI partnerships in code translation. In

Proceedings of the 26th International Conference on Intelligent User Interfaces (pp. 402-412).

Weisz, J. D., Muller, M., Ross, S. I., Martinez, F., Houde, S., Agarwal, M., ... & Richards, J. T. (2022, March). Better together? an evaluation of ai-supported code translation. In Proceedings of the 27th International Conference on Intelligent User Interfaces (pp. 369-391).

White, J., Hays, S., Fu, Q., Spencer-Smith, J., & Schmidt, D. C. (2024). Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In Generative AI for Effective Software Development (pp. 71-108). Cham: Springer Nature Switzerland.

Wood, S., Michaelides, G., & Thomson, C. (2013). Successful extreme programming: Fidelity to the methodology or good teamworking?. *Information and Software Technology*, *55*(4), 660-672.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., & Li, S. (2017). Measuring program comprehension: A large-scale field study with professionals. IEEE Transactions on Software Engineering, 44(10), 951-976.

VITA

Madhu Kota is a seasoned information technology executive with over 30 years of experience in software product development, consulting, professional services, and executive leadership. Throughout his career, he has served in a variety of roles, including Software Engineer, Technical Architect, Product Manager, Vice President of Development, Chief Information Officer, and Chief Technology Officer. Most recently, he led global product engineering at Riskonnect, Inc., where he was responsible for the research and development of integrated risk management solutions.

Mr. Kota possesses deep expertise in software engineering, agile methodologies, IT architecture, and the leadership of globally distributed development teams. He has successfully built and led high-performing software organizations across the United States and offshore, delivering enterprise-grade technology platforms and solutions.

He holds a Bachelor of Technology from Sri Venkateswara University (India), master's degrees from Anna University (India) and Auburn University (USA), and has completed the General Management Program at Harvard Business School. He earned his Doctor of Business Administration from Georgia State University.

Mr. Kota lives in Atlanta, Georgia, with his wife and two sons. In his personal time, he enjoys running, hiking, and spending time outdoors with his family and their dog.