

THE EFFECT OF ARTIFICIAL INTELLIGENCE CODE GENERATION ON SOFTWARE
DEVELOPER PRODUCTIVITY

By

Scott Morgan

John Jamison, PhD, Committee Chair

Nathan Green, PhD, Committee Member

Rongen “Sophia” Zhang, PhD, Committee Member

Anne Magro, PhD, Dean

College of Business, Innovation, Leadership, and Technology

A Dissertation Presented in Partial Fulfillment

of the Requirements for the Degree

Doctor of Business Intelligence

April 18, 2025

Dissertation Approval

DBA in Business Administration
School of Business
Marymount University

We hereby approve the dissertation of

Scott Morgan

Candidate for the Doctor of Business Administration in Business Intelligence

Date:

Committee Members:

May 1, 2025



Chair Signature
Name: John Jamison

May 1, 2025




Internal Member Signature
Name: Nathan Green

May 1, 2025



External Member Signature
Name: Rongen (Sophia) Zhang

Accepted by

DocuSigned by:


428E463BBF08456...
Anne Magro

Dean of Marymount University College of
Business, Innovation, Leadership and Technology

Copyright

© 2025 by Scott Morgan

Dedication

This dissertation is lovingly dedicated to my wife, Amy, for her endless support and partnership in our adventures; to my children, Emilee and Ethan, who inspire me to reach higher; and to the memory of my parents, whose example of resilience, sacrifice, and hard work shaped my character and inspired the journey that led to this accomplishment.

Acknowledgements

I would like to sincerely thank my dissertation committee for their guidance, feedback, and support throughout this process. To Dr. John Jamison, my chair, for providing steady leadership, clear expectations, and valuable encouragement at every stage. To Dr. Nathan Green and Dr. Sophia Zhang, for their insightful feedback, critical questions, and thoughtful guidance, which strengthened this research in meaningful ways. I am deeply grateful for the time, expertise, and commitment each committee member invested in helping me complete this work.

I would also like to acknowledge Ed Middlebrook, who instilled in me the importance of professionalism, and Grady Christie, who showed me that a little irreverence can be just as valuable.

I am grateful to my colleagues, past and present, whose support, encouragement, and insights helped shape my professional journey.

A special thank you to Team Bravo — Jeffrey Harris, Furrukh Irshad, and Lydia Eden.

Finally, I would like to thank my Little League teammates from Northcrest Food Store for some of the earliest lessons in grit, teamwork, and believing that anything is possible with the right team behind you.

For every lesson, every encouragement, and every example — thank you.

Abstract

AI code-generation tools promise to improve developer productivity, but realizing these gains depends on understanding how developer attributes and work environments interact with these technologies. This quantitative study analyzed professional developers from the 2023 and 2024 Stack Overflow Developer Surveys, conducting confirmatory, exploratory, and predictive analyses to assess the impact of AI code generation on developer productivity, measured as time spent searching for programming solutions. Confirmatory analyses found that AI code-generation usage alone did not significantly reduce search time. However, developer experience, country population, and specific tool–language combinations significantly moderated outcomes. Less experienced developers and developers from smaller-population countries experienced greater efficiency gains. Predictive analyses identified years of professional experience, frequency of workplace interruptions, and country population as the strongest predictors of search behavior. Interaction effects revealed that AI tools such as Codeium and GitHub Copilot influenced productivity differently across programming language environments. Notably, combinations such as Codeium with Systems languages and GitHub Copilot with Rust/R and Ruby were associated with significant changes in search time. These findings underscore the complexity of AI adoption in professional software development, emphasizing that the benefits of AI code generation depend not only on tool selection but also on developer demographics, experience levels, and technical ecosystems.

Keywords: AI code generation, software development, developer experience, developer productivity, Stack Overflow

Table of Contents

Copyright.....	iii
Dedication.....	iv
Acknowledgements	v
Abstract	vi
List of Figures	x
List of Tables.....	x
Chapter One – Introduction	1
Chapter Two – Literature Review	3
Developer Productivity	3
Code Generation	6
Compilers.....	6
Programming Languages.....	8
Integrated Development Environments (IDEs)	10
Intellisense	12
Artificial Intelligence.....	13
Dartmouth Project.....	13
Natural Language Processing.....	15
Machine Learning	17
Large Language Models	18
AI Code Generation	19
GitHub Copilot.....	20
Practicality.....	22
Benchmarking and Performance	25
Security	27
Code Quality	29
Developer Interaction.....	31
Developer Productivity	34
Developer Characteristics.....	38
Developer Country	38
Developer Experience.....	41
Multiple Programming Languages	44
How This Study Differs	47
Chapter Three – Data Preparation and Transformation	48
Research Approach	48
Stack Overflow Developer Survey Archival Datasets	49
2023 Stack Overflow Developer Survey Dataset.....	50
2024 Stack Overflow Developer Survey Dataset.....	50
R Studio	51
Combine Datasets.....	51
Time Searching.....	52
AI Code Generation Usage.....	52

Number of Programming Languages	53
Developer Country	53
Developer Experience	54
Frequency of Outside Interactions	54
Developer Age	55
Education Level	56
Organization Size	56
Work Location	57
Filters - Professional Developer Criteria	57
Individual Contributor	58
Developer Type	58
Employment Status	58
Developer Role	59
Organization Size	59
Other Filters	60
Results	60
Chapter Four – Confirmatory Analysis	62
Analysis	62
Main Effect	62
Moderating Effects	65
Control Variables	66
Additional Demographic Analysis	68
Results	68
Descriptive Statistics	68
Moderator and Control Variable Factor Analysis	69
Programming Language Factor Analysis	70
Correlation Matrix	71
Linear Regression Models	72
Curvilinear Relationship	74
Research Questions and Hypotheses Discussion	74
Research Question One	74
Hypothesis One	74
Research Question Two	75
Hypothesis Two	75
Hypothesis Three	76
Chapter Five – Exploratory Analysis	77
Analysis	78
Predictors of AI Code Generation Effectiveness	78
AI Code-Generation Tools	78
Country	79
Results	80
Predictors of AI Code Generation Effectiveness	80
AI Code-Generation Tools	81
Country	82
Chapter Six - Discussion	85
Interpretation of Confirmatory Hypotheses	85
Exploratory Findings	86

Comparison to Prior Research	86
Limitations of Study	87
Practical Implications	88
Theoretical Implications	89
Recommendations For Future Research.....	90
Conclusion	91
Broader Reflections on AI, Creativity, and Human Ingenuity.....	91
<i>References</i>	93
<i>Figures</i>	107
<i>Tables</i>.....	113
<i>Appendix A</i>.....	125

List of Figures

Figure 1 - <i>How this Study Differs</i>	107
Figure 2 – <i>Conceptual Model</i>	108
Figure 3 – <i>Filter Process Results</i>	108
Figure 4 – <i>Box Plot for Descriptive Statistics</i>	109
Figure 5 – <i>Impact of Experience and AI Code-Generation on Search Time</i>	109
Figure 6 – <i>Predictive Variable Importance Plot</i>	110
Figure 7 – <i>Mean Time Searching by AI Tool</i>	110
Figure 8 – <i>Difference in Time Searching by Country</i>	111
Figure 9 – <i>Average Time Savings by Country World Map</i>	111
Figure 10 – <i>Interaction Effect of AI Code-Generation Usage and Country Population on Search Time</i>	112

List of Tables

Table 1 – <i>Developer Inclusion Criteria</i>	113
Table 2 – <i>Number of records, Mean, and Standard Deviation with and without AI Code-Generation</i>	113
Table 3- <i>Demographic Factor Analysis</i>	114
Table 4 – <i>Programming Language Factor Analysis</i>	115
Table 5 – <i>Correlations Among AI Code-Generation Usage, Time Searching for Answers, Moderators, and Other Fields with Descriptive Statistics</i>	116
Table 6 – <i>Correlations Among AI Code-Generation Usage, Time Searching for Answers, Moderators, and Programming Languages</i>	117

Table 7 – <i>Correlations Among AI Code-Generation Usage, Time Searching for Answers, and AI Code-Generation Tools</i>	118
Table 8 – <i>Time Searching Linear Regression Models with No Interactions</i>	119
Table 9 – <i>Time Searching Linear Regression Models with Interactions</i>	120
Table 10 - <i>Time Searching by AI Code Generation Tool</i>	123
Table 11 - <i>Time Savings Using AI Code Generation for the Top 25 Countries</i>	124

Chapter One – Introduction

Software developers rarely receive a feature request, helpdesk ticket, or work item that does not require some form of research. There always seems to be a new library, design pattern, or language enhancement to learn. Finding answers quickly and accurately has become a fundamental part of modern software development. As libraries, frameworks, and languages evolve, developers dedicate substantial time to searching for programming solutions (Sadowski et al., 2015).

Initially, developers relied on books, manuals, and direct peer communication (Fisher et al., 1978). With the rise of the internet, search engines such as Google largely replaced physical resources and face-to-face interactions with digital information. Platforms like Stack Overflow emerged, enabling developers to ask and answer technical questions in a community-driven forum (Hucka & Graham, 2018; Treude et al., 2011). Large language models (LLMs) like ChatGPT have recently transformed developer workflows by offering code suggestions, explanations, and solutions in natural language.

Despite the technological advancements, these tools require developers to step outside their coding environments, engaging in a process known as context switching (Forsgren et al., 2023). Context switching disrupts the cognitive flow, reducing developer efficiency, job satisfaction, and overall productivity (Forsgren et al., 2023; Meyer et al., 2022; Parnin & Rugaber, 2011; Tregubov et al., 2017). Each interruption forces developers to mentally shift between tasks and tools mentally, increasing cognitive overload and lengthening the time required to return to focused work.

Generative AI tools, such as GitHub Copilot, offer a potential solution to this problem. AI code generation seeks to enhance the developer experience by minimizing context switching,

allowing developers to stay within their integrated development environments while accessing programming solutions (Bird et al., 2022). These tools predict likely next steps based on the application's development context, delivering timely, relevant code suggestions. Research suggests that AI code generation can significantly reduce search time, enhance flow state, and boost developer productivity (Kaliavakou, 2023).

However, despite these potential benefits, professional developers often encounter barriers when adopting new technologies. Accrued technical debt, the burden of legacy code maintenance, competing priorities, and the fear of fleeting trends all contribute to adoption hesitancy (Besker et al., 2020; Lakhanpal, 1993; Meyer et al., 2021; Rifat & Viitanen, 2021; Zhang et al., 2023). The collaborative nature of large software projects further complicates individual technology adoption decisions (Lakhanpal, 1993).

To date, no archival study has examined explicitly whether professional developers experience reduced search time due to AI code generation. Prior research has primarily relied on experimental data or self-reports rather than analyzing large-scale survey responses across diverse demographics. This study addresses that gap by leveraging the Stack Overflow Developer Survey to explore how AI code generation tools impact time spent searching for programming solutions among professional developers. It also seeks to understand the demographic and professional factors that may moderate this relationship.

Given the need for deeper exploration into professional developers' search behaviors, this study aims to determine whether software developers realize benefits from AI code generation by analyzing responses from the Stack Overflow Developer Survey. The findings will provide valuable insights into the efficiency gains of AI code generation, potentially informing developers, organizations, and IT leaders.

Chapter Two – Literature Review

The combination of AI and code generation has led to a new technological development: AI code generation. This paper explores the emerging field through the product, process, and person. In this context, productivity frameworks, such as SPACE (Satisfaction, Performance, Activity, Collaboration, and Efficiency), offer a helpful way to conceptualize developer productivity and performance outcomes, although this study does not adopt a formal theoretical framework. The "product" refers to AI code-generation tools, exploring their feasibility, advancements, and security considerations to ensure they do not introduce new risks. The "process" focuses on how developers interact with and utilize these tools. Finally, the "person" highlights the role of developer characteristics, skills, demographics, and interactions with AI systems in shaping the overall impact of AI code generation on productivity and outcomes.

Early research in AI code generation concentrated on evaluating advancements and benchmarking performance. Subsequent investigations delved into ensuring the security of AI-generated code and assessing its quality. More recent studies have shifted toward understanding the dynamic relationship between developers and AI systems, ultimately exploring the broader implications of AI code generation on productivity and the development lifecycle. This evolving focus highlights the critical interplay of product, process, and person in realizing the potential of AI code generation.

Developer Productivity

Developer productivity is a complex and nuanced topic not measurable by a single metric, such as lines of code (Albrecht & Gaffney, 1983; Forsgren et al., 2021; Storey et al., 2022). Developers and managers are not always well-aligned in their views of productivity

(Storey et al., 2022). Microsoft researchers proposed the SPACE Framework, a multidimensional productivity model, to understand this complexity better. While not serving as a formal theoretical foundation for this study, the SPACE Framework offers valuable insight into the multiple factors that influence developer productivity. This productivity model identifies five core dimensions: Satisfaction and Well-Being, Performance, Activity, Communication, and Efficiency and Flow (Forsgren et al., 2021). Collectively, these dimensions address the factors influencing developer productivity, allowing organizations to measure and improve it holistically (Forsgren et al., 2021).

Satisfaction and well-being are foundational to the SPACE Framework. They capture how fulfilled developers feel about their work, team, tools, and organizational culture and how these factors affect their happiness and health (Forsgren et al., 2021). Studies have shown that satisfaction correlates strongly with productivity, making it a potential leading indicator for performance (Forsgren et al., 2021, 2023). For instance, a decline in satisfaction may signal impending burnout, reduced engagement, and lower productivity. Organizations can measure this dimension through employee surveys, gauging factors like job satisfaction, burnout levels, and developers' perceptions of their tools and workflows (Forsgren et al., 2021).

During the COVID-19 pandemic, some organizations reported increased activity metrics like pull requests and commits, but qualitative data revealed that many developers experienced a decline in well-being (Forsgren et al., 2021). This discovery demonstrates the need for balanced measures considering both quantitative output and qualitative satisfaction to capture a complete picture of productivity.

Performance refers to the outcomes of a developer's work rather than their output. High performance encompasses reliable, high-quality code that delivers business value and satisfies

customer needs (Forsgren et al., 2021). Individual contributions and team dynamics significantly contribute to performance (Forsgren et al., 2021). For example, metrics like software reliability, customer satisfaction, and feature adoption can indicate performance (Forsgren et al., 2021). However, it is critical to account for the context in which developers work, as assigning less impactful tasks to certain developers may misrepresent their overall performance (Forsgren et al., 2021, 2023).

Activity metrics quantify developers' actions, such as the number of commits, pull requests, lines of code, or code reviews completed (Forsgren et al., 2021). While these metrics can provide valuable insights into individual work patterns, they cannot measure productivity alone. For instance, a developer with many commits may not necessarily be writing high-quality or impactful code (Forsgren et al., 2021). Additionally, activity metrics often fail to account for collaborative efforts like mentoring, brainstorming, or architectural guidance, which are essential but less visible components of a developer's work (Forsgren et al., 2021).

Modern software development is a collaborative undertaking (Forsgren et al., 2021). Developers must communicate with product managers, project managers, testers, business analysts, subject matter experts, and other developers. Collaboration, the C in the SPACE Framework, measures the developer's ability to communicate effectively (Forsgren et al., 2021). Software developers who collaborate at a high level create higher-performing teams and increase the entire team's productivity (Project Management Institute, 2017).

Efficiency and flow measure the ability to complete work with minimal interruptions or delays (Forsgren et al., 2021). This dimension emphasizes uninterrupted focus time for individuals and streamlined workflows for teams. For instance, metrics like change lead time, deployment frequency, and the number of handoffs in a process can help organizations identify

inefficiencies in their development pipelines (Forsgren et al., 2021). However, optimizing for efficiency must be balanced with other dimensions, as an excessive focus on speed may compromise code quality or collaboration.

The SPACE Framework provides a structured approach to understanding and improving developer productivity by focusing on multiple dimensions simultaneously. This multidimensional view helps teams avoid relying on single metrics like lines of code, which can lead to undesirable behaviors and inaccurate conclusions. Instead, the framework encourages a more holistic perspective, enabling organizations to identify the root causes of productivity challenges and implement targeted solutions (Li et al., n.d.). For example, better collaboration can enhance satisfaction, improving performance and efficiency.

Organizations can construct a balanced view of productivity by integrating perceptual data such as surveys with system telemetry (Forsgren et al., 2021). This approach benefits management and empowers developers by providing insights into their work patterns, enabling them to optimize their time and energy. Ultimately, the SPACE Framework facilitates a deeper understanding of the interplay between individual, team, and organizational productivity, paving the way for sustained improvements in software development outcomes (Forsgren et al., 2021).

Code Generation

Compilers

The concept of automatic source code generation for computer programs dates back to 1951 when Grace Hopper developed the first compiler, the A-0 System (Hopper, 1952). Before the advent of compilers, engineers programmed computers using machine code, a tedious process requiring programmers to write long sequences of binary instructions (0s and 1s) that directly interacted with the hardware. Hopper's A-0 compiler marked a transformative step in

computing, enabling programmers to use symbolic instructions that were more human-readable. The compiler then translated These symbolic instructions into machine code, automating what was previously a labor-intensive process (Hopper, 1952).

Hopper's innovation significantly reduced programming time and minimized errors associated with manual coding. By introducing the cataloged subroutines, the A-0 compiler allowed programmers to reuse predefined operations, streamlining the process of creating complex programs (Hopper, 1952). This shift to automated translation was not merely a technical innovation. It represented a conceptual leap that laid the foundation for developing high-level programming languages like COBOL and Fortran.

Hopper described the compiler as a "library of routines" the programmer could invoke without writing extensive machine code (Hopper, 1952). This approach democratized programming, making it accessible to more individuals, including those without specialized hardware knowledge. The compiler automated routine tasks such as memory allocation, address modification, and error checking, elevating the programmer's role to focus on higher-level problem-solving rather than low-level machine instructions (Hopper, 1952). This concept is still in practice today, just with different technology.

The success of the A-0 compiler spurred rapid advancements in compiler technology, eventually leading to more sophisticated systems capable of handling increasingly complex programming tasks. For instance, Hopper's later work on the FLOW-MATIC programming language introduced even greater abstraction, allowing business professionals to write instructions in a natural language-like syntax closer to English (Sammet, 2000). This evolution highlighted the dual goals of compilers: improving efficiency while making computing more inclusive and user-friendly.

Hopper's compiler design principles, such as modularity, abstraction, and reusability, underpin modern compiler architecture. Like those used in AI-assisted programming environments, today's compilers build upon these foundational ideas, integrating optimization, error detection, and parallel processing capabilities.

Programming Languages

Programming languages serve as the bridge between human logic and computer hardware, enabling humans to communicate instructions to machines in a structured and understandable way. Over time, programming languages have evolved from low-level machine code, which directly controls hardware, to high-level languages that abstract away technical complexities and focus on problem-solving. This evolution has made programming more accessible and efficient, allowing developers to create complex systems with less effort and fewer errors. High-level programming languages, in particular, have revolutionized the field of computer science by empowering diverse communities of developers, scientists, and business professionals to write code that addresses a broad range of challenges. Among the earliest and most influential of these languages were FORTRAN, designed for scientific and engineering applications, and COBOL, created for business data processing (Backus et al., 1957; Sammet, 2000). These languages defined the early decades of software development and set the stage for modern programming paradigms by introducing innovative features such as compilers, modularity, and portability.

Developed by IBM in the 1950s and released in 1957, FORTRAN (Formula Translation) is widely considered the first high-level programming language (Backus et al., 1957). FORTRAN simplified the complex and time-consuming task of coding mathematical and

scientific problems for computers like the IBM 704. Before Fortran, programming required writing machine code manually, a process prone to errors and inefficiencies. Fortran allowed programmers to express complex mathematical formulas and algorithms in a human-readable and concise way (Backus et al., 1957). Its key innovation included a compiler that translated this human-readable language into machine code, effectively hiding the intricacies of the underlying hardware (Backus et al., 1957). The language's adoption significantly reduced the time required for writing, debugging, and running scientific programs, making it a cornerstone of computing in scientific and engineering domains (Backus et al., 1957).

Following closely, COBOL (Common Business-Oriented Language) emerged as a business-oriented programming language (Sammet, 2000). A committee composed of representatives from major computer manufacturers and government agencies created COBOL in 1959 (Sammet, 2000). COBOL was explicitly designed to be English-like and self-documenting, focusing on business data processing tasks such as file handling, report generation, and record manipulation (Sammet, 2000). The language's design reflected a push to democratize programming for business users who were not necessarily computer scientists, enabling them to write software that addressed organizational needs (Sammet, 2000). The compiler for COBOL played a crucial role in converting its English-like syntax into machine code, further cementing its accessibility. Over the decades, COBOL has become a dominant language in sectors like banking, insurance, and government, and it remains in use today for many legacy systems (Sammet, 2000).

The development of Fortran and COBOL marked the beginning of a new era in software engineering. Programming became more about problem-solving and less about understanding the minutiae of machine operations. Together, these languages demonstrated the power of

abstraction in programming, allowing for greater productivity and enabling broader participation in software development.

Integrated Development Environments (IDEs)

Integrated Development Environments (IDEs) have transformed the software development process by centralizing coding, debugging, and testing tools into a single interface (Hou & Wang, 2009). By streamlining workflows, IDEs reduce cognitive load and enhance developer productivity, which is pivotal in improving the overall developer experience (Forsgren et al., 2021; Hou & Wang, 2009). They offer a variety of features, including syntax highlighting and debugging support, which reduce context switching, simplify complex workflows, and make software development more efficient (Forsgren et al., 2023; Hou & Wang, 2009).

One of the earliest examples, Visual Basic 1.0, released in 1991, allowed developers to conceal low-level code complexities while focusing on designing user interfaces (Schulz, 1991). Visual Basic enabled developers to create fully functional applications without requiring an in-depth understanding of the underlying code (Schulz, 1991) through features such as drag-and-drop forms and prebuilt controls like buttons and text boxes. This ease of use further democratized software development, making programming more accessible to those outside traditional technical fields and aligning with early efforts to enhance developer experience by reducing friction in the development process (Forsgren et al., 2023; Schulz, 1991).

As IDEs evolved, they began incorporating features that directly addressed developer challenges. For instance, real-time error detection and inline feedback within modern IDEs reduce frustration and interruptions, enabling developers to maintain flow, one of the measures in the SPACE Framework (Forsgren et al., 2021, 2023; Hou & Wang, 2009). Features like code

snippets and templates are designed to save time and improve consistency by offering prewritten solutions tailored to common patterns (Hou & Wang, 2009). These tools empower developers to focus on higher-order problem-solving rather than repetitive tasks, a hallmark of improving productivity and developer experience (Forsgren et al., 2023; Hou & Wang, 2009).

Furthermore, IDEs contribute to collaboration and communication, two essential elements of the developer experience (Hou & Wang, 2009). Modern environments like Visual Studio Code and IntelliJ IDEA integrate version control systems like Git, enabling seamless collaboration across distributed teams. Developers can review, merge, and manage code changes without leaving the IDE, reducing context switching and fostering team alignment (Hou & Wang, 2009). These collaborative features align with the SPACE Framework's emphasis on teamwork and efficiency, further demonstrating the integral role of IDEs in shaping a positive developer experience (Forsgren et al., 2021, 2023; Hou & Wang, 2009).

The DevEx Framework highlights how tools and environments can affect developer well-being by reducing frustration and cognitive overload (Forsgren et al., 2023). IDEs play a crucial role in this context by automating repetitive tasks, such as refactoring and code formatting, and providing visual cues that guide developers through complex workflows (Hou & Wang, 2009). For instance, IDEs like Eclipse and PyCharm offer integrated testing environments and performance profiling tools, allowing developers to identify bottlenecks and optimize applications without external tools (Hou & Wang, 2009). These features enhance productivity and contribute to a sense of accomplishment and satisfaction, which is essential for developer well-being (Forsgren et al., 2021; Hou & Wang, 2009).

As IDEs continue to evolve, their design increasingly prioritizes usability and accessibility. Developer experience research underscores the importance of intuitive design and

minimal friction in development tools (Forsgren et al., 2023). Modern IDEs address this by offering customizable interfaces, dark mode options, and drag-and-drop capabilities (Hou & Wang, 2009). They also support integrations with third-party plugins, enabling developers to tailor their environments to meet specific needs (Hou & Wang, 2009). This flexibility reflects the growing recognition that a positive developer experience fosters creativity and innovation (Forsgren et al., 2023).

IDEs are now indispensable across software development domains, from enterprise solutions to mobile apps and embedded systems. Their ability to consolidate essential tools into a unified platform while addressing key developer experience elements like flow, collaboration, and satisfaction underscores their critical role in modern development workflows. By incorporating user-centric design principles, IDEs improve technical efficiency and enhance the human aspects of programming, ensuring developers remain engaged, productive, and fulfilled in their work (Forsgren et al., 2021, 2023; Hou & Wang, 2009).

IntelliSense

IntelliSense, a cornerstone feature of modern IDEs, epitomizes how technology enhances productivity and reduces cognitive load for developers (Míšek, 2017). By leveraging real-time context awareness, IntelliSense provides a seamless way for programmers to access functions, methods, properties, and events. Combined with "dot notation," it automatically generates suggestions and code completions as programmers type (Míšek, 2017). This feature accelerates coding speed and significantly minimizes errors by reducing manual effort in remembering and typing exact syntax (Míšek, 2017). Since the IDE integrates IntelliSense, Developers can quickly make informed decisions by presenting relevant suggestions and code snippets without

constantly referring to external documentation or switching between tools (Meyer et al., 2022). This reduction in context-switching streamlines the overall workflow and prevents disruptions (Forsgren et al., 2023; Meyer et al., 2022).

In addition to speeding up code writing, IntelliSense ensures syntactical correctness (Meyer et al., 2022). For example, when programmers encounter unfamiliar libraries or APIs, IntelliSense directly provides inline documentation, such as method definitions and expected parameters, in the editor. This guidance improves code accuracy and promotes better readability and maintainability (Meyer et al., 2022). Therefore, IntelliSense also serves as an invaluable tool for onboarding junior developers or those new to a specific language or framework. They can rapidly gain proficiency by learning through real-time feedback (Meyer et al., 2022). Moreover, IntelliSense fosters team collaboration, as consistent suggestions across IDEs promote uniform coding practices (Forsgren et al., 2021; Meyer et al., 2022).

Artificial Intelligence

Dartmouth Project

Artificial intelligence as a field of study formally began in the 1950s, with the Dartmouth Summer Research Project on Artificial Intelligence often credited as its foundational moment (McCarthy et al., 1955; Moor, 2006). John McCarthy, Marvin Minsky, Nathaniel Rochester, and Claude Shannon proposed the 1956 Dartmouth Conference to explore the possibility of simulating human intelligence using machines (McCarthy et al., 1955; Moor, 2006). The proposal boldly hypothesized that "every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it" (McCarthy et al., 1955).

No discussion of artificial intelligence would be complete without mentioning Ada Lovelace's seminal notes on Charles Babbage's Analytical Engine, written over a century earlier (Fuegi & Francis, 2003). In her "Notes," Lovelace envisioned a machine capable of manipulating symbols and solving problems beyond numerical calculations, effectively predicting the potential of AI long before its formal conception (Fuegi & Francis, 2003). Her insights into symbolic processing and computational possibilities remain foundational to the broader understanding of intelligent systems (Fuegi & Francis, 2003).

The Dartmouth Conference laid the groundwork for AI as a formal discipline, attracting leading thinkers to collaborate on diverse problems, such as machine learning and neural networks (Moor, 2006). While the conference's outcomes were more fragmented than initially hoped, it ignited a shared vision of machines performing tasks requiring human intelligence. Early advancements included foundational work on the Information Processing Language (IPL) and the Logic Theory Machine by Allen Newell and Herbert Simon, demonstrating how computers could mimic human problem-solving capabilities (Moor, 2006).

This era of AI research was marked by ambitious goals, such as enabling machines to understand and generate language, solve abstract problems, and adapt to new environments (Moor, 2006). However, limited computational power and incomplete theoretical understanding limited progress (Moor, 2006). Despite these constraints, the Dartmouth Conference catalyzed interest in specialized areas like Natural Language Processing, Machine Learning, and neural networks, setting the stage for AI's rapid evolution in subsequent decades (McCarthy et al., 1955; Moor, 2006).

By establishing AI as a distinct academic discipline, the Dartmouth Conference coined the term "artificial intelligence" and created a vision that continues to drive innovation today

(McCarthy et al., 1955). This vision, focused on replicating intelligent behavior, has since expanded to encompass many applications, including natural language processing and machine learning.

Natural Language Processing

Natural Language Processing (NLP) has witnessed remarkable advancements due to breakthroughs in deep learning, attention mechanisms, and transfer learning (Collobert et al., 2011; Devlin et al., 2019; Vaswani et al., 2023). Collobert et al. (2011) introduced one of the pioneering frameworks for applying deep learning to NLP. Their work laid the groundwork for training neural networks to perform multiple NLP tasks simultaneously, including part-of-speech tagging, named entity recognition, and semantic role labeling (Collobert et al., 2011). Using word embeddings, the researchers demonstrated that a single, unified neural network could achieve state-of-the-art performance across various tasks without relying on handcrafted features (Collobert et al., 2011). This work highlighted the importance of distributed representations for words and demonstrated the versatility of deep learning models in NLP (Collobert et al., 2011). However, the architecture faced scalability issues and could not capture long-range dependencies effectively (Collobert et al., 2011). This limitation paved the way for innovations like the attention mechanism and transformer models.

Vaswani et al. (2023) introduced the Transformer model, a paradigm-shifting architecture that addressed the limitations of recurrent neural networks (RNNs) and convolutional neural networks (CNNs). The core innovation of the Transformer is the self-attention mechanism, which allows the model to weigh the importance of different words in a sequence when making predictions (Vaswani et al., 2023). Unlike RNNs, Transformers process entire sequences in

parallel, significantly improving computational efficiency and enabling the modeling of long-range dependencies (Vaswani et al., 2023).

Multi-head attention enhanced the model's ability to capture diverse linguistic patterns. Vaswani et al. (2023) demonstrated the Transformer's performance in machine translation tasks, achieving state-of-the-art results on the WMT 2014 English-to-German and English-to-French datasets (Vaswani et al., 2023). Their findings underscored the importance of attention-based architectures, which have become a cornerstone of modern NLP (Vaswani et al., 2023).

Building on the Transformer architecture, Devlin et al. (2019) introduced Bidirectional Encoder Representations from Transformers (BERT), a pre-trained language model that revolutionized NLP. BERT employs a two-stage training process: unsupervised pretraining on large corpora using masked language modeling and supervised fine-tuning on specific tasks (Devlin et al., 2019). This approach enables the model to capture bidirectional context, a critical factor for understanding nuanced language constructs (Devlin et al., 2019).

BERT achieved state-of-the-art performance on various benchmarks, including the GLUE (General Language Understanding Evaluation) tasks and SQuAD (Stanford Question Answering Dataset) (Devlin et al., 2019). Its success demonstrated the power of transfer learning in NLP, where pre-trained models can be adapted to downstream tasks with minimal additional training. The introduction of BERT marked a paradigm shift, making NLP tools accessible to a broader audience and significantly reducing the time and resources required to develop task-specific models.

The combined insights from Collobert et al., Vaswani et al., and Devlin et al. illustrate the evolution of NLP from task-specific models to unified frameworks capable of handling diverse applications. Collobert's early work emphasized the importance of shared

representations, while Vaswani's Transformer architecture provided the computational backbone for modern NLP systems. Devlin's BERT showcased the transformative impact of pretraining and transfer learning (Collobert et al., 2011).

Machine Learning

Machine Learning (ML) is a subfield of AI that focuses on developing algorithms and statistical models that enable computers to learn and improve performance on a task without being explicitly programmed (Mitchell, 2013). ML operates on the principle that computers can automatically learn from data, identify patterns, and make decisions or predictions without human intervention. Key ML components include the data used to train models, algorithms that identify patterns, and the training process, where data is fed into algorithms to optimize performance (Mitchell, 2013). ML's ability to learn from data and improve over time made it possible for AI systems to move beyond hardcoded rules to generate solutions dynamically. The ability to learn from data led to Large Language Models, which leverage machine learning to process large datasets and respond to complex prompts (Mitchell, 2013). As NLP and ML intersected, AI systems could understand human input and generate highly relevant and accurate responses, including programming code (Mitchell, 2013).

ML models have evolved significantly due to advances in computational power, the availability of large datasets, and novel algorithms (Crossley et al., 2006). Notable breakthroughs include the development of Convolutional Neural Networks (CNNs), which revolutionized image recognition tasks by mimicking human visual processing (Krizhevsky et al., 2012). CNNs leverage convolutional layers to extract hierarchical features, making them highly effective for large-scale image classification tasks (Krizhevsky et al., 2012).

Further advancements have been driven by attention mechanisms, particularly with the introduction of the Transformer architecture. The Transformer eliminates the reliance on recurrent neural networks, instead employing self-attention to model relationships within data sequences (Vaswani et al., 2023). This approach has significantly improved the efficiency and parallelization of training, leading to state-of-the-art performance in natural language processing (NLP) tasks like translation. Additionally, the scalability of ML has been facilitated by advances in hardware and optimization techniques. For example, the Adam optimizer and dropout regularization have successfully trained robust models at scale. The field continues to innovate with applications extending beyond text and images, encompassing multi-modal data like audio and video.

Large Language Models

Large Language Models (LLMs) are machine learning models based on advanced neural network architectures, such as transformers, which have demonstrated impressive performance on various natural language processing tasks (Chen et al., 2021). These models are trained on massive amounts of text data, enabling them to understand language broadly and generate meaningful output. In AI code generation, LLMs are trained on vast quantities of program source code, allowing them to generate code based on user prompts (Chen et al., 2021). Examples of such models include OpenAI's GPT (Generative Pre-Trained Transformer) and Google's BERT (Bi-Directional Encoder Representations from Transformers) (Wolf et al., 2020). By using LLMs, AI can assist developers by generating code based on the context provided by the user. This capability is at the heart of AI tools like GitHub Copilot, which leverages LLMs to improve developer productivity by reducing the time spent searching for solutions.

Recent advancements in LLMs have focused on improving efficiency in training and deployment and enhancing their alignment with human intentions (Y. Liu et al., 2024). Techniques such as model compression, mixed-precision training, and parameter-efficient fine-tuning have been developed to reduce the computational resources required for training large models. Also, fine-tuning strategies like prompt learning, alignment tuning, and reinforcement learning with human feedback (RLHF) refine LLM responses, making them more aligned with user expectations and ethical standards (Y. Liu et al., 2024). These advancements optimize the performance of LLMs. In addition, these advancements address ethical concerns, enabling developers to produce functional and ethically appropriate (Y. Liu et al., 2024)code.

AI Code Generation

AI code generation research has evolved progressively. First, researchers examined the feasibility and potential of advancing code generation (Allamanis et al., 2019; Chen et al., 2021; Feng et al., 2020; Kim et al., 2021; Sun et al., 2020). That analysis led to benchmarking and performance research, where experts evaluated the capability and effectiveness of AI-enhanced coding tools (Ernst & Bavota, 2022; Hendrycks et al., 2021; Sarsa et al., 2022; Sobania et al., 2022; Weisz et al., 2022). Next, researchers focused on examining the security of AI code generation (Asare et al., 2024; Horne, 2023; Pearce et al., 2022). These studies focused on whether AI code generation introduced new security risks. Next, research was conducted on code quality (Dakhel et al., 2023; Evtikhiev et al., 2023; Finnie-Ansley et al., 2022; Mastropaolo et al., 2023). The next step was the research's dynamics of developer interaction with AI systems (Barke et al., 2022; Mozannar et al., 2024; Vaithilingam et al., 2022). Finally, researchers studied

the impact on developer productivity (Bird et al., 2022; Kaliamvakou, 2023; Sarkar et al., 2022; Xu et al., 2022; Ziegler et al., 2022).

GitHub Copilot

GitHub Copilot is a prime example of an AI-powered code generation tool that aids software developers by providing real-time code suggestions, autocompletion, and context-aware code snippets as they write code (Bird et al., 2022). It is designed to operate within integrated development environments (IDEs) and supports various programming languages and frameworks (Bird et al., 2022). By operating within the IDE, Copilot reduces context switching. Furthermore, It can expedite the development process by offering code suggestions (Bird et al., 2022).

Copilot is built on OpenAI's Codex, a language model trained on source code, and GitHub, the largest source code repository globally (Bird et al., 2022). Microsoft owns both GitHub and a share of OpenAI, so they used their public code repositories to train the OpenAI Codex model. For instance, the model underwent training on over 159 GB of Python code alone (Bird et al., 2022).

Copilot allows developers to browse different suggestions, enabling them to explore code and choose the most suitable option for their needs (Bird et al., 2022). By examining multiple suggestions, the programmer can better understand an unfamiliar topic. This practice aligns with deliberate decision-making and conscious thought (Barke et al., 2022). By reducing context switching and providing code snippets in real-time, Copilot allows developers to maintain focus and flow (Bird et al., 2022). This focus decreases the time spent searching for solutions, increasing efficiency. Exploring code empowers developers to concentrate on higher-level tasks,

such as creating resilient and secure architectures and creating more innovative and successful software solutions (Bird et al., 2022).

However, Copilot's capabilities come with notable challenges. Sarkar et al. (2022) observed that while Copilot helps developers focus on higher-level abstractions, it also necessitates carefully checking the generated code for errors or inefficiencies. Developers frequently shift their role between a "navigator" who designs higher-level logic and a "driver" who verifies and corrects Copilot's outputs (Dakhel et al., 2023; Sarkar et al., 2022). This dynamic resembles aspects of pair programming but without the collaborative verbalization and shared responsibility that human pairing entails (Dakhel et al., 2023; Sarkar et al., 2022).

The paper also noted that the specificity and clarity of developers' prompts affect the acceptance of Copilot's suggestions (Sarkar et al., 2022). Poorly defined prompts can lead to irrelevant or incorrect suggestions, necessitating additional effort to refine or rework the generated code (Sarkar et al., 2022). This finding aligns with other studies suggesting that the successful use of Copilot relies on developers' ability to communicate intent clearly through prompts (Bird et al., 2022; Dakhel et al., 2023; Sarkar et al., 2022).

In addition, Sarkar et al.'s (2022) research explored the broader implications of Copilot's integration into software development workflows. While developers using Copilot reported increased productivity and reduced cognitive load, they also expressed concerns about overreliance on the tool, particularly in contexts requiring nuanced problem-solving or domain expertise (Sarkar et al., 2022). For example, Copilot struggled with tasks involving highly specialized knowledge, edge cases, or ambiguous problem statements, sometimes producing inefficient or insecure solutions (Sarkar et al., 2022).

To address these challenges, Sarkar et al. (2022) recommended that organizations using Copilot adopt complementary strategies such as prompt engineering, continuous oversight, and encouraging a verification-first approach, where developers carefully review and test Copilot's suggestions for correctness and security (Sarkar et al., 2022).

Despite its limitations, Copilot has demonstrated the potential to significantly enhance developer productivity (Sarkar et al., 2022). Sarkar et al. (2022) reported that developers using Copilot could complete tasks more quickly in experimental settings than their counterparts who relied solely on traditional tools. This advantage was particularly evident in scenarios involving repetitive tasks or standard code patterns, where Copilot's contextual suggestions provided substantial time savings (Sarkar et al., 2022).

By integrating Copilot into their workflows, development teams can reduce mundane coding tasks, enabling a greater focus on creativity and innovation. However, as Sarkar et al. (2022) emphasized, this requires a structured approach that balances the tool's strengths with a keen awareness of its limitations.

Practicality

In their seminal work, Allamanis et al. (2019) explore machine learning and software engineering through the lens of "Big Code" and the inherent "naturalness" of programming languages. Also, their research demonstrated how statistical learning techniques can effectively predict and generate source code (Allamanis et al., 2019). In addition, their research reviewed how LLMs can handle large codebases, enhancing developer productivity. By analyzing the predictability and repetitiveness of code, the research underscores the potential of AI-assisted tools to assist code completion, bug detection, and even complex program creation (Allamanis et

al., 2019). The study revealed the capabilities and challenges of using machine learning to create source code. It also highlighted the transformative potential of AI tools to increase developer productivity (Allamanis et al., 2019). AI code generation could lead to classical human software developers becoming obsolete (Barenkamp et al., 2020).

Several large language models (LLMs) use transformer architecture to improve contextual understanding, and numerous researchers have employed this architecture to comprehend and produce code in conjunction with human language (Feng et al., 2020; Kim et al., 2021; Sun et al., 2020). For example, CodeBERT, which integrates programming and natural languages through a bimodal pre-trained model, has excelled in tasks that combine code with descriptive text, such as automated code documentation and code generation from natural language descriptions (Feng et al., 2020). CodeBERT automates routine programming activities, enhancing code generation accuracy and reducing developers' cognitive load (Feng et al., 2020).

Kim et al. (2021) integrated structured programming syntax with advanced machine learning to enhance code prediction. This approach used the code structure in abstract syntax trees (ASTs), which were then processed using a transformer-based model (Kim et al., 2021). By inputting ASTs into transformers, the model captured both the syntactic and semantic nuances of programming languages, enabling more accurate and contextually relevant code predictions (Kim et al., 2021). This approach demonstrated significant improvements over traditional models that relied solely on linear text processing. Furthermore, this approach offered substantial benefits in terms of developer productivity by reducing errors and streamlining the coding process (Kim et al., 2021).

Sun et al. (2020) integrated tree structures into a transformer framework. This architecture used abstract syntax trees (ASTs) to capture the hierarchical and structural

complexities present in programming languages, enabling the generation of syntactically and semantically accurate code (Sun et al., 2020). TreeGen significantly improved over traditional linear text-based models through the Transformer's self-attention mechanism (Sun et al., 2020). These advancements resulted in higher accuracy. Benchmarking revealed the model's high-performance, highlighting its potential to reduce manual coding workload, streamline debugging, and boost developer productivity (Sun et al., 2020). TreeGen's capacity to automate and refine the code-writing process demonstrates the impact of advanced AI technologies in software development. TreeGen is a point of reference for discussions on AI-enhanced programming tools to enhance coding efficiency and accuracy (Sun et al., 2020).

In their pioneering study, OpenAI researchers studied large language models trained on code, notably focusing on the Codex model (Chen et al., 2021). This model, a successor to GPT-3, underwent fine-tuning using a diverse and extensive dataset of publicly available source code. This fine-tuning enabled the model to comprehend and generate programming languages proficiently (Chen et al., 2021). Furthermore, the study assessed the model's performance across different coding tasks, including code completion, bug fixing, and code generation. One key finding was that Codex reduced the time and effort required for coding tasks. Therefore, Codex could aid programmers by providing suggestions and automating routine programming activities (Chen et al., 2021). However, the research also highlighted challenges, including handling complex logic, maintaining context over longer code sequences, and ensuring the security and reliability of generated code. This study demonstrated the practical applications of AI in software development and emphasized the need for further improvements in model robustness (Chen et al., 2021).

Benchmarking and Performance

After establishing the practicality and potential of AI code generation through research, a subsequent wave of researchers undertook additional studies to assess its performance (Ernst & Bavota, 2022; Hendrycks et al., 2021; Sarsa et al., 2022; Sobania et al., 2022; Weisz et al., 2022). In addition to benchmarking, these studies also address the importance of intuitive user interfaces in developers' IDEs (Ernst & Bavota, 2022; Hendrycks et al., 2021; Sarsa et al., 2022; Sobania et al., 2022; Weisz et al., 2022). Furthermore, some researchers have raised concerns about security vulnerabilities, copyright infringement, and the potential for emerging (Ernst & Bavota, 2022; Sarkar et al., 2022).

Hendrycks et al. (2021) introduced the APPS benchmark, an evaluation tool for assessing the proficiency of large-scale language models. The tool evaluated Python code generated from natural language prompts (Hendrycks et al., 2021). This benchmark addressed the challenges of objectively and comprehensively measuring coding ability, covering many problems, from simple tasks to complex algorithmic challenges. The tool tested models for accuracy, specifications, and algorithm creation (Hendrycks et al., 2021). The findings indicated reduced syntax errors and improved correctness of AI-generated code (Hendrycks et al., 2021). Furthermore, these findings signaled a significant advancement in automated code generation. This study underscored the capabilities of AI and the need for ongoing improvements in model training to achieve reliable and practical code generation (Hendrycks et al., 2021).

Researchers at IBM studied the interaction between human software engineers and AI by translating code from Java to Python (Weisz et al., 2022). The findings revealed that, despite the imperfect nature of AI-generated translations, the combined efforts of human engineers and AI substantially improved the accuracy and efficiency of the code translation process (Weisz et al.,

2022). The study, which involved 32 software engineers, showed that AI assistance reduced errors compared to when engineers worked independently (Weisz et al., 2022). In Addition, the research examined the impact of AI-generated translations and concluded that having access to multiple translations had a more significant influence on the engineers' translation process. This research highlighted the potential for AI to improve human performance in software development. It also emphasized the importance of intelligent user interfaces when working with AI tools (Weisz et al., 2022).

In a comparative study, researchers evaluated the capabilities of GitHub Copilot against traditional genetic programming (GP) methods in automatic program synthesis (Sobania et al., 2022). The study focused on benchmark problems from standardized suites to assess both methodologies under consistent conditions. The team discovered that while GitHub Copilot and GP show comparable performance on many tasks, GitHub Copilot demonstrated a distinct advantage in terms of speed and ease of use (Sobania et al., 2022). Therefore, Copilot is more practical for real-time coding assistance. This study demonstrates the strengths and weaknesses of using large language models like Codex for coding tasks. In addition, it points to the importance of developing more intuitive interfaces and efficient processing techniques to enhance the utility of AI-driven programming tools (Sobania et al., 2022). The findings indicate a significant step forward in integrating AI into everyday development practices, proposing a hybrid approach where AI complements traditional programming expertise (Sobania et al., 2022).

While acknowledging the efficiency gains these tools offer by automating mundane aspects of coding, some researchers caution against potential shadows such as the emergence of biases, security vulnerabilities, and the blurring of legal lines around code ownership and

copyright (Ernst & Bavota, 2022). This perspective presents AI code generation as a technological advancement, but not without socio-technical challenges. This research serves as a reflection on the potential and limitations of AI in software development. It also reflects on how AI tools can be harnessed responsibly to support developers without compromising the craft and quality of software engineering (Ernst & Bavota, 2022).

Security

In light of previous research highlighting potential security vulnerabilities in AI-generated code, several studies have been conducted to evaluate the security aspects of AI code-generating tools (Asare et al., 2024; Horne, 2023; Pearce et al., 2022). One study utilized a dataset of documented vulnerabilities and found that Copilot replicated vulnerabilities present in human-generated code approximately 33% of the time while generating correct, non-vulnerable code in about 25% of cases (Asare et al., 2024). The findings indicated that, like human programmers, Copilot is susceptible to introducing security flaws. However, Copilot is no worse than humans, creating security flaws (Asare et al., 2024). In addition, the study emphasized the capabilities of Copilot, recognizing its potential to adapt and mitigate existing vulnerabilities based on the context of the prompt and the specific type of vulnerability. This research has contributed to a better understanding of security implications in AI code generation tools (Asare et al., 2024). Also, this research laid the groundwork for improved development practices to mitigate the risks associated with automatic code generation (Asare et al., 2024).

Pearce et al. (2022) comprehensively examined GitHub Copilot's ability to generate secure and vulnerable code in various contexts. The study systematically evaluated Copilot's outputs across different programming scenarios, particularly those associated with MITRE's

"Top 25" Common Weakness Enumeration (CWE) list, identifying critical security weaknesses (Pearce et al., 2022). The results showed that around 40% of the code generated by Copilot in these high-risk scenarios was vulnerable, therefore highlighting significant security concerns (Pearce et al., 2022). Furthermore, the research investigated the impact of different prompts on the security of the generated code, offering a detailed analysis of the conditions that could lead Copilot to produce insecure code (Pearce et al., 2022). These findings have important implications, suggesting that while Copilot can improve developer productivity, developers should exercise careful oversight and integrate rigorous testing and security protocols to minimize potential risks in code generation (Pearce et al., 2022).

Building on these findings, Horne (2023) introduced the concept of "PwnPilot," reflecting on the trust challenges posed by AI code assistants. The study emphasized that vulnerabilities in generated code often stem from issues in the training data, poisoning attacks, and insufficient post-generation analysis (Horne, 2023). The study outlined the potential for backdoors to be introduced into AI models, highlighting how seemingly innocuous or functional outputs could conceal malicious elements (Horne, 2023). The research further recommended integrating automated static code analysis tools like Pylint and Bandit into development workflows, advocating for AI-driven multi-version programming and feedback loops to mitigate risks associated with untrusted AI outputs (Horne, 2023).

Furthermore, the concept of trust in AI-generated code extends beyond the tools to the lifecycle of AI development (Horne, 2023). The document underscores the importance of securing training data, fine-tuning processes, and distribution mechanisms to mitigate the possibility of data poisoning or model compromise (Horne, 2023). Horne also highlights that while AI tools like Copilot are improving security awareness, relying solely on AI without robust

human oversight and complementary tools can exacerbate risks (Horne, 2023). Implementing safety guardrails and adopting diverse feedback mechanisms are proposed as necessary steps to enhance the security of AI-generated code.

These studies collectively stress that while AI code assistants hold transformative potential, their use in secure programming contexts requires an integrated approach involving rigorous testing, static analysis, and continuous improvements to training methodologies and model transparency. Developers must remain vigilant, adopting a "trust but verify" mindset to ensure that AI outputs meet security standards.

Code Quality

The emerging research on AI code generation has shown its potential benefits and demonstrated that the security of AI-generated code is comparable to that of human-produced code (Evtikhiev et al., 2023). As research continues, researchers have focused on evaluating the code quality generated by AI tools (Evtikhiev et al., 2023). Evtikhiev et al. (2023) emphasized the importance of establishing metrics for evaluation. They comprehensively assessed traditional and code-specific metrics such as BLEU, METEOR, ROUGE-L, ChrF, CodeBLUE, and RUBY (Evtikhiev et al., 2023). Their study revealed the challenges of applying these metrics, initially designed for natural language processing, to code generation, particularly in their correlation with human judgment (Evtikhiev et al., 2023). Across various datasets, the researchers found that commonly used metrics often failed to align with human evaluations, primarily when the metric scores were closely clustered (Evtikhiev et al., 2023). This finding uncovered the need to develop new evaluation frameworks to capture the quality of AI-generated code, ensuring that

they align more closely with human code functionality and readability (Evtikhiev et al., 2023) standards.

A recent study found that although Copilot effectively generated solutions for a wide range of algorithmic challenges, its outputs often contained bugs and exhibited limitations in integrating multiple methods to create a coherent solution (Dakhel et al., 2023). The study also noted that while Copilot's solutions occasionally had bugs, they generally required less effort to correct than those produced by humans. These findings suggest that Copilot can significantly assist experienced developers by accelerating the coding process and proposing potentially valuable solutions (Dakhel et al., 2023). However, Copilot may pose risks when used by novices who cannot effectively identify and rectify the tool's errors. The study highlights the dual nature of AI tools in software development: they can enhance productivity when used appropriately. Still, they can introduce risks if relied upon too heavily without adequate oversight (Dakhel et al., 2023).

Another recent study examined the impact of different phrasings in function descriptions on the consistency and quality of the generated code (Mastropaolo et al., 2023). The research found that almost half of the code outputs varied when the input descriptions were phrased differently. Since developers rely on code generation accuracy, this finding created concerns in professional environments (Mastropaolo et al., 2023). By highlighting the need to enhance the accuracy of AI-based code generators, the findings underscored the importance of creating more stable and reliable AI-driven coding assistants capable of handling these language nuances without compromising code quality (Mastropaolo et al., 2023).

Developer Interaction

After establishing reliable, stable, and secure code generation, researchers focused on how developers utilize AI code generation (Barke et al., 2022; Harrison Oke Ekpobimi et al., 2024; Mozannar et al., 2024; Vaithilingam et al., 2022; Wang et al., 2024). Vaithilingam et al. (2022) found that while AI code generation may not always fully address developers' queries, it does provide a valuable starting point. One study highlighted considerable time savings from analyzing developer interactions and AI code-generating tools (Mozannar et al., 2024). A grounded theory study revealed that developers use GitHub Copilot in two modes: acceleration and exploration (Barke et al., 2022).

A comparative study by Liu et al. (2023) provided valuable insights into how developers interact with ChatGPT and Stack Overflow during various programming tasks, including algorithm development, library usage, and debugging. The study found that ChatGPT excels in providing quick and high-quality solutions for algorithmic and library-related tasks, offering developers a streamlined interaction experience through natural language prompts (J. Liu et al., 2023). Developers preferred ChatGPT for its ability to generate runnable code consistently, reducing the time spent searching and integrating scattered resources (J. Liu et al., 2023). However, developers found Stack Overflow more effective for debugging tasks due to its repository of explicit exceptions and detailed solutions that include community-vetted responses (J. Liu et al., 2023). Furthermore, the study emphasized that developers using ChatGPT required additional time to refine and understand generated code, highlighting a cognitive overhead in adapting AI-provided solutions to specific contexts. This finding underscores the importance of designing AI tools that better align with developer workflows and enhance productivity and comprehension (J. Liu et al., 2023).

Mozannar et al. (2024) introduced the CodeRec User Programming States (CUPS), a comprehensive taxonomy that classifies various activities of programmers as they interact with Copilot. By observing 21 programmers who used CUPS to label their interaction behaviors, the authors gained insights into how programmers engage with, respond to, and are potentially impeded by AI-generated code suggestions (Mozannar et al., 2024). This study indicated that accepting, modifying, or rejecting suggestions cost developers time. Additionally, this discovery highlighted how these systems impact developer productivity. Through this analysis, the researchers provided an understanding of the potential and limitations of AI in software development (Mozannar et al., 2024). They suggested opportunities for enhancing interface design and integrating AI tools in programming environments (Mozannar et al., 2024).

Another recent study conducted an in-depth analysis of how software developers perceive and interact with AI-assisted code generation tools such as GitHub Copilot (Vaithilingam et al., 2022). By performing a user study with 24 participants, the researchers found that although Copilot did not consistently improve task completion times or success rates, developers preferred it over traditional code-completion tools because it provided substantial starting points that reduced initial coding efforts (Vaithilingam et al., 2022). However, the study also revealed significant challenges. For example, participants often encountered issues understanding, editing, and debugging the AI-generated code, sometimes hindering task-solving effectiveness (Vaithilingam et al., 2022). The paper discusses the need to improve the design of such tools to make them more intuitive and supportive in real-world programming environments. It suggests that while AI tools can offer valuable assistance, they need to evolve to accommodate better the complex nature of programming tasks and the cognitive processes of developers (Vaithilingam et al., 2022).

Trust in AI-powered code-generation tools is shaped by multiple factors, including their perceived benefits, alignment with developers' goals, and the situational context in which they are used (Horne, 2023; Wang et al., 2024). Prior research has emphasized the need for empirical studies examining users' trust in AI tools in real-world scenarios. Retrospective interviews with developers who have used these tools reveal that trust is influenced by practical benefits (ability), the tool's alignment with developers' goals (benevolence), and its trustworthy processes (integrity) (Wang et al., 2024). Additionally, situational factors, such as use cases and programming tasks' complexity, further shape trust attitudes. These findings support existing literature suggesting that trust in AI tools is not static but evolves, is situationally dependent, and is affected by social and organizational. Understanding how developers build and maintain trust in AI-driven code generation is crucial for improving adoption and usability, particularly as these tools become more embedded in professional software development workflows (Wang et al., 2024).

The increasing adoption of AI code assistants like GitHub Copilot and the code-generation capabilities of ChatGPT demonstrates their potential to enhance developer productivity by automating routine programming tasks (Horne, 2023). Additionally, studies suggest that AI-assisted coding may improve developer satisfaction, reinforcing its value in modern programming environments (Horne, 2023). However, concerns about security vulnerabilities in AI-generated code have emerged alongside these benefits. Research on AI security highlights risks such as undetectable backdoors in machine learning models and threat vectors in AI-driven software supply chains, raising the possibility of malicious AI coding assistants like "PwnPilot" compromising system integrity. Several commercial tools and security frameworks have been developed to mitigate these risks, including GitHub Copilot's enhanced

vulnerability prevention and NVIDIA's Guardian. As AI code assistants become more embedded in software engineering, balancing productivity gains with security considerations will remain a key research and industry challenge (Horne, 2023).

Barke et al. (2022) conducted critical grounded theory research in this study. The study observed 20 participants interacting with Copilot during various programming tasks (Barke et al., 2022). The researchers discovered that developers engage with Copilot in two distinct modes: acceleration and exploration. In acceleration mode, programmers use Copilot to speed up the coding process when clear about their next steps without disrupting their workflow (Barke et al., 2022). Acceleration mode is similar to intellisense. Alternatively, programmers use Copilot as a heuristic tool in exploration mode to navigate through unfamiliar code or solve problems when unsure of the solution pathway. This study underscored the potential of AI code-generation tools to improve developer productivity and job satisfaction (Barke et al., 2022).

Developer Productivity

In considering how developers utilize AI code generation, it is crucial to examine the impact of AI code generation on developer productivity (Bird et al., 2022; Coutinho et al., 2024; Ebert & Louridas, 2023; Kalamvakou, 2023; Sarkar et al., 2022; Sun et al., 2023; Weber et al., 2024; Xu et al., 2022; Ziegler et al., 2022). Microsoft studied the interactions between programmers and GitHub Copilot (Bird et al., 2022). This study provided valuable insights into how Copilot facilitates a new approach to pair programming by serving as the "navigator" in the coding process, enabling programmers to focus on strategic tasks rather than routine coding (Bird et al., 2022). Microsoft's study highlighted the advantages and challenges of using Copilot, describing how AI code-generating tools can adapt to different coding styles and preferences

(Bird et al., 2022). Furthermore, the study revealed that Copilot could increase developer productivity. However, the study also noted that while Copilot can generate a wide range of functional code snippets, it occasionally produces vulnerable or inappropriate suggestions due to its reliance on pre-trained models (Bird et al., 2022). Microsoft's research emphasized the need to refine AI tools to ensure they offer reliable and contextually appropriate support, aligning with developers' expectations and real-world coding standards (Bird et al., 2022).

Sarkar et al. (2022) studied the practical implications of programmers using AI-assisted tools such as GitHub Copilot. Their research explored the impact of large language models on programming and compared them with traditional search and pair programming methods (Sarkar et al., 2022). The authors identified distinct challenges and benefits of LLM-assisted programming, including the capacity to generate code from natural language descriptions and the need for programmers to refine their prompt-writing skills to communicate with the AI effectively (Sarkar et al., 2022). Key findings revealed mixed effects on productivity, with some developers experiencing faster coding times while others grappled with the accuracy and safety of the generated code. The research underscored the transformative potential of LLMs in programming while cautioning against excessive reliance on such tools due to their current limitations in comprehending complex user intents and generating error-free code (Sarkar et al., 2022).

In a recent study, researchers addressed the inefficiencies in AI code generation, explicitly focusing on the high frequency of unhelpful completions that create more burden for developers rather than aiding them (Sun et al., 2023). The study found that developers do not use approximately 70% of AI-generated completions, significantly wasting computational resources and developer time (Sun et al., 2023). To address this issue, the researchers proposed an early-

rejection mechanism that can predict the usefulness of code completions before their entire generation. This mechanism prevents unnecessary processing by identifying "low-return prompts" unlikely to yield valuable completions, enhancing productivity and reducing computational waste (Sun et al., 2023). By implementing a lightweight Transformer-based estimator, the researchers demonstrated the feasibility of this approach, showing substantial savings in computational costs while maintaining or even enhancing developer productivity (Sun et al., 2023). This study emphasizes the importance of developing sustainable and efficient AI tools and highlights the potential for AI to evolve into more context-sensitive and resource-efficient forms (Sun et al., 2023).

Xu et al. (2022) researched in-IDE code generation in PyCharm. Their study involved a controlled experiment with 31 participants assigned Python programming challenges with and without AI code generation (Xu et al., 2022). The research revealed no significant improvement in task completion time or program correctness when utilizing the code generation tools. This finding highlights a limitation of AI in real-world programming, specifically in effectively translating natural language prompts into helpful code suggestions. While these tools may improve the coding experience by offering potential solutions, they may not increase productivity or reduce error rates (Xu et al., 2022). The study also noted several areas for improvement, including the need for AI-generated code to better align with user intentions (Xu et al., 2022).

A study by GitHub examined the impact of neural code completion tools on developer productivity (Ziegler et al., 2022). Researchers analyzed programmers' behavior using GitHub Copilot to understand how developers interacted with AI code generation tools (Ziegler et al., 2022). The study found that developers' perception of productivity correlated to the acceptance

rate of AI suggestions. The research focused on acceptance rates and usage analytics, which provide insights into the actual world effectiveness of AI code generation (Ziegler et al., 2022). The study highlighted the complexity of measuring productivity enhancements due to AI tools. It furthermore suggested that higher acceptance rates correlate to positive productivity perceptions among developers. This study emphasized optimizing AI interfaces to enhance the developer coding experience (Ziegler et al., 2022).

GitHub conducted additional research to investigate the practical and psychological effects of AI-powered tools like GitHub Copilot on developer productivity and satisfaction (Kalamvakou, 2023). This research was initiated after the launch of Copilot in 2021 and aimed to quantitatively and qualitatively assess its impact on productivity. The study emphasized the challenges of defining and measuring developer productivity (Kalamvakou, 2023). It suggested that developer productivity should include more than just lines of completed code. Developer productivity should include emotional and cognitive well-being elements, such as reduced frustration and sustained mental focus during coding tasks (Kalamvakou, 2023). The findings, obtained through large-scale surveys and controlled experiments, demonstrated that while Copilot can significantly accelerate coding tasks and improve job satisfaction, it also requires a new methodology for evaluating productivity, one that integrates traditional efficiency metrics with a nuanced understanding of developer well-being and task enjoyment (Kalamvakou, 2023). That new methodology is the SPACE Framework of developer productivity.

Based on prior research, this study hypothesizes:

H1: Developers using AI code generation tools will spend less time searching for answers to programming questions than those not using those tools.

Developer Characteristics

Developer Country

Technology adoption and impact can vary significantly across different countries due to various factors, including economic development, technological infrastructure, and cultural attitudes towards technology (Arora & Gambardella, 2005; Coccia, 2014; Gonzalez-Barahona et al., 2008; Yildirim & Ansal, 2011). Understanding these variations is crucial for assessing the overall impact of AI code-generating tools on developer productivity. Additionally, research indicates an uneven distribution of developers across different regions, with lower participation from countries in Asia and Africa (Gonzalez-Barahona et al., 2008). Internet penetration and economic wealth influence these disparities. For instance, countries like India and China have higher wealth-adjusted contributions to global open-source development than other regions, indicating that economic factors and resource allocation significantly affect participation levels (Gonzalez-Barahona et al., 2008).

Countries with advanced technological infrastructure, such as the United States, Germany, and Japan, have higher adoption rates of new technology (Yildirim & Ansal, 2011). These countries typically have better internet connectivity, better access to high-performance computing resources, and more companies investing in new technologies. For instance, high-speed internet and cloud computing services can significantly enhance the usability and efficiency of AI tools, leading to more significant productivity gains (Yildirim & Ansal, 2011). These differences mean that while developers in countries with advanced technological infrastructure can easily access AI code generation tools, developers in less developed countries may experience slow connectivity and limited access to these tools. This disadvantage hinders

the developer's ability to access AI code generation (Yildirim & Ansal, 2011). Bridging this gap will ensure that developers in emerging markets can keep pace with technological advancements.

The country's economics is another consideration when adopting AI technologies (Yildirim & Ansal, 2011). Developed nations might have more financial resources to invest in new technologies. GitHub data indicates that open-source AI innovation has increased exponentially since February 2022 (Dohmke et al., 2023). However, developing countries may encounter budget restrictions that hurt adoption (Yildirim & Ansal, 2011). Nevertheless, Free/Libre Open Source Software (FLOSS) initiatives have demonstrated that developing countries can cultivate innovative capabilities through community collaboration and open-source contributions (Yildirim & Ansal, 2011). While FLOSS initiatives provide valuable opportunities for collaboration and innovation, the long-term effects of uneven access to AI-driven tools may widen the gap between regions (Yildirim & Ansal, 2011). This disparity could slow innovation in areas where developers face budgetary or technological barriers, limiting their ability to compete in the global software industry.

Cultural factors could also play a role in accepting new (Yildirim & Ansal, 2011). Different countries have different levels of trust and receptiveness towards new technologies. For instance, the FLOSS community in Turkey has showcased substantial involvement and creativity despite economic challenges, highlighting how cultural elements like communal cooperation and the aspiration for technological autonomy can propel the embrace of pioneering technologies (Yildirim & Ansal, 2011). Cultural attitudes toward AI adoption vary significantly across regions. For example, countries wary of AI due to concerns about job displacement may experience slower adoption rates. In contrast, countries with a cultural drive for independence, like Turkey's FLOSS community, may embrace AI tools more readily, demonstrating the power

of culture to drive technological innovation despite economic challenges (Yildirim & Ansal, 2011).

Arora and Gambardella (2005) studied the globalization of the software industry and discovered several factors that can contribute to success in emerging economies. For example, India, Ireland, and Israel have experienced significant growth due to their focus on exports, government policies, and a highly skilled workforce (Arora & Gambardella, 2005). Despite initially limited resources and infrastructure, these countries have effectively utilized their unique advantages to become significant players in the global software industry (Arora & Gambardella, 2005). The country-specific factors of economic development, infrastructure, and culture deeply influence the ability of developers to adopt AI code-generation tools and boost their productivity (Arora & Gambardella, 2005).

A study on different languages and coding efficiency investigates the information transmission rates of 17 languages with various linguistic structures (Coupé et al., 2019). Despite significant differences in speech rate (the number of syllables spoken per second) and information density (how much information is conveyed per syllable), the overall information rate across languages remains similar, averaging around 39 bits per second (Coupé et al., 2019). This balance suggests a universal tendency where languages compensate for lower information density by increasing speech rate and vice versa (Coupé et al., 2019).

Regarding how different languages might impact a developer's coding ability, the study suggests that the cognitive load associated with processing different languages may differ based on these information rates (Coupé et al., 2019). For example, languages with high syllabic information density might impose a more significant cognitive load per unit of time. Still, they may require fewer syllables (and possibly less time) to convey a concept, affecting information

processing efficiency (Coupé et al., 2019). While this study does not directly relate to coding, the differences in linguistic encoding could imply varying cognitive demands across languages, potentially influencing how easily developers in multilingual contexts process and comprehend code or programming concepts across different spoken languages (Coupé et al., 2019).

As measured by patents, population growth is crucial in shaping sustainable economic growth and technological innovation (Coccia, 2014). Research examining OECD countries suggests an inverted-U relationship between population growth rates and technological output. This indicates that moderate population growth fosters higher innovation levels, whereas low/negative growth and high growth (above 1% annually) hinder technological performance (Coccia, 2014). The findings suggest that increasing population growth beyond a certain threshold may lead to diminishing returns in research productivity as the balance between income per capita and population growth becomes strained (Coccia, 2014).

Based on the reviewed literature, this study will explore how factors related to the respondent's country of residence may influence the use of AI code-generation tools and the time developers spend searching for answers to programming questions. Given the wide range of underlying differences across countries, such as technological infrastructure, educational systems, and economic factors, the country will be examined through exploratory analysis rather than as a formal moderating hypothesis.

Developer Experience

Developer experience is a critical factor influencing various aspects of software development, including code quality, maintainability, and productivity (Baltes & Diehl, 2018; Banker et al., 1998; Beaver & Schiavone, 2006; Feigenspan et al., 2012). Understanding how

years of experience affect these outcomes is essential for optimizing software development processes. This section reviews empirical studies that explore the relationship between developer experience and software development, emphasizing its role as a moderator in the conceptual model.

Feigenspan et al. (2012) highlight that experience is critical to developer performance. Their study demonstrates a significant correlation between the years developers have been programming professionally and their efficiency in completing programming tasks (Feigenspan et al., 2012). Experienced developers solve programming tasks more accurately and quickly. They also identify and resolve errors quickly, reducing the time spent on tasks. The study highlights experience as a vital factor in assessing developer performance (Feigenspan et al., 2012).

Experienced developers can be skilled at solving problems quickly, but they may rely too heavily on established patterns (Feigenspan et al., 2012). This reliance can sometimes create tunnel vision and prevent innovation, so experienced developers may not explore novel approaches (Feigenspan et al., 2012). Thus, understanding how years of experience interact with other factors, such as creativity and risk-taking, is critical for optimizing productivity.

Banker et al. (1993) conducted a study exploring the impact of experience on software maintenance performance. They found that project team experience is inversely related to the time required for software enhancement projects. Experienced teams can comprehend and modify complex software more efficiently, reducing project effort. This efficiency stems from their ability to process complex informational cues and execute maintenance tasks effectively (Banker et al., 1998).

In addition, the Banker et al. (1993) study investigated tools designed to automate the creation of standard code routines. The study revealed that while code generators can improve initial productivity, they could also make maintenance more challenging (Banker et al., 1998). However, experienced developers are better at managing the complexities created by code generators, mitigating their adverse effects on maintenance performance (Banker et al., 1998). The ability to handle such complexities ensures code maintainability and highlights the critical role of experience in balancing productivity and long-term code quality.

The Beaver & Schiavone (2005) paper provides empirical evidence from NASA's Kennedy Space Center, demonstrating that the skill and experience of the software development team significantly affect project timelines. The study shows that more experienced developers positively influence project timelines by reducing the time required to complete tasks (Beaver & Schiavone, 2006). Conversely, less experienced developers often struggle with responsibilities beyond their skill level, leading to increased errors and delays (Beaver & Schiavone, 2006). This finding emphasizes the critical role of experience in managing project timelines and ensuring the timely completion of software development projects (Beaver & Schiavone, 2006).

Many people think experience and expertise are the same, but the study by Baltes and Diehl (2018) provides a nuanced perspective. Their research suggests that experience, measured in years, does not necessarily correlate strongly with self-assessed expertise, especially across diverse developer populations (Baltes & Diehl, 2018). They argue that expertise in software development is highly task-specific and influenced by deliberate practice and individual differences (Baltes & Diehl, 2018). Deliberate practice involves targeted activities to improve performance. Developers who engage in deliberate practice, monitoring, and self-reflection enhance their task efficiency and reduce the time required to complete programming tasks

(Baltes & Diehl, 2018). This insight highlights that while years of experience are essential, they do not automatically translate into expertise (Baltes & Diehl, 2018).

Simply relying on experience without expertise can lead developers to overlook the importance of continuous learning. In fields like software development, expertise is often driven by the ability to acquire new skills, especially in response to emerging technologies. Developers who are not lifelong learners could fail to develop new skills and are relegated to maintenance roles. These studies all underscore the importance of considering work experience as a moderator when using AI code generation to reduce time searching for answers to programming questions. While experience generally enhances efficiency and project timelines, its role is nuanced and context-dependent. Expertise, developed through continuous learning and practice, is important for optimizing performance. Mentorship and collaboration also play a crucial role in ensuring that experienced developers contribute to team efficiency, reducing errors, and improving project outcomes.

Based on the reviewed literature, this study hypothesizes:

H2: Developer experience (years of professional software development) will moderate the relationship between AI code-generation tool use and time spent searching for answers, such that developers with fewer years of experience will experience a stronger reduction in search time.

Multiple Programming Languages

Research indicates that while mastering multiple programming languages can present challenges, it also brings substantial advantages such as enhanced flexibility, problem-solving

skills, and software interoperability in projects (Bissyande et al., 2013; Kochhar et al., 2016; Shrestha et al., 2022). However, the heightened likelihood of defects in multi-language projects underscores the importance of robust quality assurance practices. Simultaneously, working with different languages can foster collaboration and project success. Understanding these trade-offs is crucial for developers and managers striving to optimize software development processes (Bissyande et al., 2013; Kochhar et al., 2016; Shrestha et al., 2022).

Bissyande et al. (2013) examined different programming languages based on popularity, interoperability, and impact across 100,000 open-source projects on GitHub. Popularity was gauged based on the number of projects, lines of code, and developer involvement (Bissyande et al., 2013). Interoperability was assessed by analyzing projects employing multiple languages. The research revealed that scripting languages like JavaScript, Ruby, and Python are widely utilized across numerous projects and by various developers, often combined with other languages (Bissyande et al., 2013). The study also identified robust interoperability in languages like C and its derivatives such as C++ and Java. The findings suggested that developers proficient in multiple languages are better equipped to contribute to diverse projects (Bissyande et al., 2013).

Kochhar et al. (2017) studied the impact of using multiple programming languages on the number of defects in software development. The research used a dataset of GitHub projects written in 17 programming languages. The findings indicated that projects employing multiple languages exhibited higher defect proneness (Kochhar et al., 2016). Also, specific languages such as C++, Objective-C, and Java had a higher error rate in multi-language settings. The study employed negative binomial regression, accounting for project age, team size, and the number of commits (Kochhar et al., 2016). The results suggested that while knowledge of multiple

languages can offer flexibility and broader problem-solving approaches, it also introduces complexity that may result in more bugs(Kochhar et al., 2016).

Another study delved into developers' cognitive challenges when learning additional programming languages (Shrestha et al., 2022). The study shed light on interference theory, where existing programming knowledge interferes with acquiring new languages. Empirical evidence was retrieved from Stack Overflow questions and interviews with professional programmers (Shrestha et al., 2022). Furthermore, the study revealed that erroneous assumptions stemming from prior language knowledge could lead to significant learning difficulties (Shrestha et al., 2022). This interference can impede developers' efficiency and prolong the time needed to attain proficiency in new languages. Thus, while multi-language knowledge is advantageous, it can also introduce cognitive load that hinders swift adaptation and learning (Shrestha et al., 2022).

The challenge of projects written in multiple programming languages goes beyond the individual developer (Shrestha et al., 2022). As new developers begin working on a multi-language project, they may need longer ramp-up times, affecting team performance and project timelines. Thus, while multi-language proficiency can broaden a developer's skill set, teams must be mindful of the potential cognitive burdens it introduces and how they might affect overall productivity (Shrestha et al., 2022).

Mastering multiple programming languages gives developers an advantage in terms of flexibility. However, the increased defect proneness and cognitive challenges introduced by multi-language environments necessitate robust quality assurance practices and careful project management (Shrestha et al., 2022). If teams can balance the advantages of flexibility with the complexities of multi-language integration, software development teams can optimize individual

performance and project outcomes (Shrestha et al., 2022). The collaborative advantages of multi-language fluency must be weighed against the potential for cognitive overload and the need for more extensive testing and quality assurance in complex multi-language environments.

Based on the reviewed literature, this study hypothesizes:

H3: The number of programming languages a developer uses will moderate the relationship between AI code-generation tool use and time spent searching for answers, such that developers familiar with more programming languages will experience a stronger reduction in search time.

How This Study Differs

This study adds to the existing research on AI code generation by exploring the impact on professional developer productivity. Specifically, the study focuses on how generative AI tools influence developers' time searching for programming solutions. This study uses archival data from the Stack Overflow Developer Surveys of 2023 and 2024, enabling a robust, data-driven analysis of real-world developer interactions with AI tools. This study provides insights into the practical implications of AI-assisted coding in natural developer environments rather than controlled experimental settings using archival data. Figure 1 illustrates where this research is situated compared to other studies.

Chapter Three – Data Preparation and Transformation

This quantitative research examines the impact of AI code generation tools on developer productivity using archival data from the Stack Overflow Developer Surveys. These annual datasets, comprising responses from thousands of developers worldwide, are a rich source of information, providing valuable insights into developer behaviors and trends (*Stack Overflow Developer Survey 2023*, 2023; *Stack Overflow Developer Survey 2024*, 2024). Therefore, these datasets are ideal for evaluating the practical effects of AI-powered development tools on productivity. However, since the survey evolves year-to-year, the datasets need data manipulation before merging. In addition, some transformed and computed fields are created to undertake the analyses in this study.

Research Approach

This quantitative, causal-comparative study examines the relationship between AI code generation usage and time spent searching for programming solutions among professional software developers. Causal-comparative research involves using pre-existing groups to explore the differences between dependent variables (Schenker & Rumrill, Jr., 2004). The analysis includes confirmatory, exploratory, and predictive components, utilizing archival data from the Stack Overflow Developer Surveys. Filters are applied to the datasets so that only professional developers are analyzed. In addition, filters ensure complete data for all variables, emphasizing workflows where productivity is critical and reassuring the audience about the study's applicability to real-world scenarios.

Stack Overflow Developer Survey Archival Datasets

The Stack Overflow Developer Surveys for 2023 and 2024 provide valuable datasets for analyzing developer behavior and demographics, including time spent searching for programming solutions. These surveys enable a focused examination of professional developers, offering insights into their practices and preferences (*Stack Overflow Developer Survey 2023*, 2023; *Stack Overflow Developer Survey 2024*, 2024). The Stack Overflow Developer Surveys are commonly used in research. For example, the ADBU Journal of Engineering Technology published an article analyzing big data analytics worldwide (Beeharry & Ganoo, 2018).

Due to U.S. transport/export sanctions, potential respondents in Crimea, Cuba, Iran, North Korea, and Syria could not access the surveys. While some participants used VPNs to bypass restrictions, this limitation may have led to the underrepresentation of specific demographics and technical perspectives, particularly from emerging markets. This factor should be considered when interpreting results (*Stack Overflow Developer Survey 2023*, 2023; *Stack Overflow Developer Survey 2024*, 2024).

The datasets are publicly available under the Open Database License (ODbL) (*Open Data Commons Open Database License (ODbL) v1.0 — Open Data Commons: Legal Tools for Open Data*, 2023). This licensing promotes transparency, reproducibility, and collaboration, enabling researchers to build on these findings. Each dataset is provided as a compressed zip file containing several key components:

- Survey Results (CSV): Raw responses, each row representing a respondent and each column representing a survey question.
- Schema File: Detailed descriptions of survey questions and response options for accurate interpretation.

- README File: An overview of dataset contents, survey methodology, and citation guidelines.
- PDF Survey Copy: A complete record of survey questions and answers.

These comprehensive resources, with their detailed descriptions, complete records, and clear guidelines, ensure that researchers can effectively analyze and interpret the data to generate meaningful insights, instilling confidence in the study's thoroughness.

2023 Stack Overflow Developer Survey Dataset

The 2023 Stack Overflow Developer Survey, conducted from May 8 to May 19, gathered responses from 89,184 qualified participants (*Stack Overflow Developer Survey 2023*, 2023). Recruitment was primarily through Stack Overflow's communication channels, including onsite notifications, blog posts, emails, newsletters, advertisements, and social media, targeting the platform's most engaged users. An essential addition to the 2023 survey was question QID328, which asked respondents about their use of AI-powered development tools over the past year, providing a list of ten tools via checkboxes (*Stack Overflow Developer Survey 2023*, 2023). This question is central to the statistical analysis in this study. According to Kaggle, the dataset has been viewed 7,751 times (*Kaggle Stack Overflow Developer Survey 2023*, n.d.).

2024 Stack Overflow Developer Survey Dataset

The 2024 Stack Overflow Developer Survey was conducted from May 19 to June 20, and 65,437 qualified responses were received from 185 countries (*Stack Overflow Developer Survey 2024*, 2024). Recruitment methods mirrored those of 2023, relying on Stack Overflow's platforms and targeting highly engaged users. The survey's median completion time was

approximately 21 minutes, reflecting the inclusion of additional questions. Question QID327 in the 2024 survey consolidated QID327 and QID328 from 2023, asking respondents about their use of AI-powered development and search tools over the past year (*Stack Overflow Developer Survey 2024*, 2024). The 2024 dataset has been viewed 2,882 times on Kaggle (*Kaggle Stack Overflow Developer Survey 2024*, n.d.).

R Studio

All analyses in this study were conducted using R Studio, a versatile integrated development environment (IDE) for R statistical software (R Core Team, 2024). R Studio's wide-ranging capabilities suit various analyses, including descriptive statistics, correlation matrices, Linear Regression, and Random Forest models (R Core Team, 2024).

The R Studio project is organized into a main file and function files. This architecture improves readability and maintainability. These files are `main`, `create_dataset`, `column_functions`, `filter_functions`, `misc_functions`, `confirmatory_functions`, and `predictive_functions`. Each analysis includes details of the specific R packages used, ensuring a clear and structured approach to the work.

Combine Datasets

This study combined the 2023 and 2024 Stack Overflow Developer Survey datasets into a single dataset of 154,621 respondents. Combining the datasets allowed the study to analyze AI code-generation data over two consecutive years. A file named `columns.csv` was created to standardize the dataset with a list of the column names for the analyses. The `create_dataset`

function initialized an empty data frame with these columns, ensuring consistency across datasets. The `read_and_select` function was then applied to each survey year to:

- Read the dataset for that year.
- Select columns matching those in `columns.csv`.
- Add missing columns as NA values.
- Reorder columns to match the standardized structure.
- Append a `SurveyYear` column indicating the year of each dataset.

The datasets were merged using the `bind_rows` function from the `dplyr` package (Wickham et al., 2023). Duplicate columns, such as "Country.1," were removed to maintain a concise dataset.

Time Searching

The dependent variable in the conceptual model is the time spent searching for answers to programming questions. Question QID291 from the Stack Overflow Developer Survey (SODS) categorizes search time into groupings ranging from less than 15 minutes to over 120 minutes daily. A derived variable, `TSX`, was created for statistical analysis, assigning numeric values to these groupings: 15, 30, 60, 120, and 150. The `dplyr` package facilitated the creation of this derived field (Wickham et al., 2023).

AI Code Generation Usage

For the confirmatory study, the conceptual model's independent variable is the respondent's use of an AI code-generating tool. Question QID328 from the Stack Overflow Developer Survey (SODS) asked respondents to indicate which AI-powered development tools

they used regularly over the past year, providing a list of tools represented as checkboxes. Respondents could select none, one, or multiple tools.

For the confirmatory analysis, a new boolean variable named AID was created. This variable is TRUE if a respondent selected any AI tool and FALSE if no tools were selected. Additionally, a new boolean column was created for each tool. If a respondent selected a tool, its boolean value was TRUE. Tracking AI code-generation usage tools allows more granular analysis.

Number of Programming Languages

For analysis of the moderating variable number of programming languages used, the study uses question QID233 from the SODS. The question asks what programming languages the respondent used over the last year and offers a list of programming languages to select. The respondent can choose none, one, or more than one of the languages in the list. The study sums the number of programming languages and stores that count in a new PLC variable. In addition, each programming language gets a new bit column in the dataset. The bit column is one if the respondent selected that programming language and zero if they did not. This step allows a more detailed examination of individual programming languages' effect on search time.

Developer Country

For analysis of the moderating variable developer country, the study uses question QID6 in the SODS. This question asks the respondent where they live and has a list of countries from which to choose. The respondent may only choose one Country or NA from the list. The country

does not require any data preparation or new variables. However, the study created a new field named CTY_INT to store an integer representation of the country for linear regression.

The dataset has been given additional context with the add_world_data function. This function, leveraging the rnatrualearth package, has added global data such as population estimates, GDP, economy classification, and income group (Massicotte & South, 2024). The data was merged with the primary dataset using a standardized ISO country code column (CTY_ISO3A). These enhancements have empowered the analyses to factor in regional disparities by providing valuable socioeconomic context.

Developer Experience

For analysis of the moderating variable developer experience, the study uses question QID288 in the SODS. The question asks the respondents how many years of professional experience they have. The answers are either a numeric value selected from a dropdown or "Less than 1 year," "NA," or "More than 50 years." The study created a new variable for the non-numeric values to hold a numerical value. The value used in the variable is the number selected from the dropdown. For "Less than 1 year," .5 was used as the value, and for "More than 50 years," 55 was used as the value. The data in the 2023 and 2024 datasets did not include respondents with less than 1 year or more than 50 years of experience. In addition, the study filters out any NA values for developer experience.

Frequency of Outside Interactions

The frequency that developers seek help or interact with individuals outside their immediate team, as well as their experiences with knowledge silos, provides insights into

collaboration and potential challenges in adopting AI code-generating tools. Question QID290 from the Stack Overflow Developer Survey includes three Likert-scale questions:

1. "How often do you need help from someone outside your team?"
2. "How often do you interact with someone outside your team?"
3. "How often do you encounter knowledge silos in your organization?"

Responses were transformed into integer values for quantitative analysis:

- "Never" = 0
- "1-2 times a week" = 2
- "3-5 times a week" = 5
- "6-10 times a week" = 10
- "10+ times a week" = 15

This numeric scale reflects the extent of collaboration and knowledge silos within an organization. The derived columns enable the study to analyze patterns in developer interactions better. Missing responses were marked as NA to maintain data integrity.

Developer Age

The age of a developer could influence their approach to adopting AI code-generating tools. Question QID127 in the Stack Overflow Developer Survey asks respondents to indicate their age group, with options such as "Under 18 years old," "18-24 years old," "25-34 years old," and so on, up to "65 or older" and "Prefer not to say." The study created a new variable, *Age_Int*, which assigns a numerical value to each age group to enable quantitative analysis. In this variable, the midpoint value of each age range is used as a representative integer: for instance, 18 is assigned for "Under 18 years old," 24 for "18-24 years old," and 68 for "65 or older."

Respondents who selected "Prefer not to say" or provided other non-numeric responses are assigned an NA value to exclude them from age-based analyses. This transformation ensures that age can be evaluated continuously, facilitating comparisons regarding AI tool adoption and developer productivity.

Education Level

The education level of a developer may influence their approach to and comfort with AI code-generating tools. Question QID25 in the Stack Overflow Developer Survey asks respondents about their highest level of formal education, with options ranging from "Primary/elementary school" to "Professional degree (JD, MD, Ph.D., Ed.D., etc.)." For analysis purposes, a new variable, `EdLevel_Int`, was created to assign a numerical value to each educational category, allowing it to be treated as a continuous variable. For example, 5 represents "Primary/elementary school," 12 for "Secondary school," 16 for "Bachelor's degree," and 21 for "Professional degree." This conversion uses typical education levels as numerical proxies, enabling comparative analysis of respondents' educational backgrounds. Responses categorized as "Something else" are assigned a value of 8, while any unspecified responses are marked as NA. This approach standardizes education levels, supporting insights into how different educational backgrounds may correlate with AI tool adoption and productivity in development.

Organization Size

Question QID29 asks, "How many people are employed in your company?" and offers a list of answers such as "2-9 employees", "10-19 employees", and "20-99 employees", etc. A new

column was created based on the organization size and the upper value of the category. For example, “2-9 employees” is represented as 9, “10-19 employees” is transformed to 19, etc. These numeric representations allow organization size to be used in linear regression models.

Work Location

Question QID308 asks the respondents if they work in person, fully remote, or on a hybrid schedule. The answers are converted into numeric values and stored in a new field. In-person employees are transformed into the number one. Hybrid workers are given a value of two, and fully remote workers are given a value of three. These numbers were selected so that one has more time in the office and three have the most time away from the office. By using 1-3 in this manner, linear regression can be used to determine the effect of work location on search time.

Filters - Professional Developer Criteria

Stack Overflow defines a developer as anyone who writes code (*Stack Overflow Developer Survey 2023, 2023; Stack Overflow Developer Survey 2024, 2024*). However, this study narrows the focus to professional developers, individuals employed full-time in a technical role within an organization comprising more than one person.

The Stack Overflow Developer Survey dataset contains data points that enable filtering of the respondents down to those professional developers working at organizations with more than one person. Table B1 outlines the survey questions and responses used for this filtering and shows the resulting counts of developers included in the 2023 and 2024 datasets (Wickham, 2016).

Individual Contributor

Question QID287 asks, "Are you an individual contributor, or do you manage people?" The two possible answers are "I am an individual contributor" or "I manage people." This study only includes respondents who are individual contributors, as it focuses on developers engaged directly in coding activities without managerial responsibilities. Including only individual contributors ensures the analysis reflects the experiences of developers using AI code generation tools. The distinction is crucial for this research because it seeks to understand the impact of AI code generation tools on the coding process itself, and managers might not directly engage with these tools in their day-to-day work as much as individual contributors do.

Developer Type

Question QID2 asks, "Which of the following options best describes you today?" Only respondents who selected "I am a developer by profession" were considered for this study. This question helps to differentiate between individuals who program as a profession and those who code for other purposes, such as hobbies or education. By concentrating on professional developers, the study aims to provide a more specific assessment of how AI code-generation tools impact developers more likely to use these technologies in a professional setting.

Employment Status

This study focuses on professional developers who are employed full-time, as this group is most likely to regularly use and benefit from AI code generation tools in a consistent work environment. Question QID296 asks, "What is your current employment status?" This study only uses respondents who answered "Employed full time" to this question. This question aims to

categorize respondents based on their employment status to identify the working context of professional software developers. Understanding the employment status helps analyze how different work environments might influence the use of AI tools and the search behaviors for programming solutions.

Developer Role

Software developers can perform a wide range of roles, and question QID31 in the SODS states, "Which of the following best describes the type of developer work you do?" Only respondents selecting "Back-end developer," "Front-end developer," "Full-stack developer," and "Desktop or enterprise applications developer" were used. This study focuses on respondents engaged in these specific roles as they represent core areas of software development that are most likely to integrate and benefit from AI code generation technologies. This question is designed to identify the respondents' specific areas of software development. Understanding the development roles allows the study to analyze how AI code generation impacts software developers' roles.

Organization Size

This research focuses on developers working in organizations with more than one person. It excludes freelancers or sole proprietors, focusing on professional developers working for companies. This filter is important because it allows the examination of the impact of AI code generation within larger teams and development environments. Collaborative coding and team dynamics are more relevant than on small teams. Question QID29 asks, "How many people are employed in your company?" and offers a list of answers such as "2-5 people", "6-25 people",

and "5000 or more". The study filters out any respondents who answered Exclude "I don't know" and "Just me – I am a freelancer, sole proprietor, etc." This question aims to gauge the size of the respondent's workplace, which can influence the development environment, team collaboration, and the adoption of tools like AI code generators. The study targets a demographic likely to benefit from and interact with collaborative coding tools and AI technologies in a team setting by excluding single-person operations and unknowns.

Other Filters

Time Searching (QID291), Years of Experience (QID288), Languages (QID233), Age (QID127), and Work Location (QID308) are also essential questions for this study. These questions provide data on the respondents' experience, search behaviors, age, and geographical context, and any records with missing values for any of these questions were excluded to maintain data completeness. The Frequency of Interruption questions (QID290) are designed to help developers understand the collaboration. These questions all require responses with no NA values allowed. Requiring answers to these questions ensures a complete understanding of respondents' collaboration and communication patterns. This understanding is essential for analyzing team dynamics and knowledge sharing in development environments.

Results

The dataset initially contained 152,621 records but was reduced after applying the filtering criteria. These filters excluded non-professional developers, respondents without essential answers, and cases with ambiguous locations. After using these filters, the dataset was

narrowed to 32,981 records for analysis. Table 1 presents the filter results in tabular format, while Figure 3 visualizes the filtering process.

This filtering process is essential to maintain data quality and focuses on professional developers who will most likely engage with AI tools in meaningful, productivity-related contexts. By analyzing this clean and well-matched dataset, the study can draw more accurate and generalizable insights into the impact of AI code-generating tools on developer productivity within a professional setting.

Chapter Four – Confirmatory Analysis

This quantitative study examines the effect of AI code generation on the time spent searching for answers to programming questions and, in turn, developer productivity. Research question one asks, "How does AI code generation impact developer productivity?" Research question two asks, "Do any demographic characteristics moderate the effect of AI code generation on time spent searching for answers to programming questions?" This chapter discusses the confirmatory analysis designed to answer those two questions. Figure 2 visually represents the conceptual model of this confirmatory analysis.

Analysis

Main Effect

H1 states: *Developers using AI code generation tools will spend less time searching for answers to programming questions than those not using those tools.* This study employs a combination of descriptive statistics, a correlation matrix, and linear regression analysis to confirm H1. These methods provide a high-level overview of patterns through aggregate metrics, identify relationships between variables, and offer a quantitative assessment of the relationship between AI tool usage and search times. This section provides evidence to evaluate the validity of H1 and explores the broader implications of AI tool usage on developer productivity.

Descriptive Statistics include aggregate metrics such as means, medians, and standard deviations. These statistics are generated with the entire combined dataset and split by AI tool usage (AID = 1 vs. AID = 0), allowing the study to compare the impact of AI tools on search times. Descriptive statistics were expressed as numbers with a sub-group comparison. R Studio

is used for descriptive statistics, employing the dplyr package for data manipulation and summarization (Wickham et al., 2023).

To understand further the underlying structure of the relationships among the study variables, an exploratory factor analysis (EFA) was conducted. Factor analysis is a data reduction technique that identifies latent constructs by grouping variables based on shared variance. This approach is beneficial when dealing with multiple correlated variables, as it simplifies complex datasets and uncovers meaningful ones. Given this study's number of independent, dependent, and control variables, factor analysis was employed to determine whether certain variables loaded onto common factors, which could then inform subsequent analyses.

The principal axis factoring method was employed to extract factors, followed by an oblique rotation to allow for potential correlation between factors. The Kaiser-Meyer-Olkin (KMO) measure of sampling adequacy and Bartlett's test of sphericity were used to assess the appropriateness of the dataset for factor analysis. The psyche package in R was used for the factor analysis (Revelle, 2007). Factors were retained based on eigenvalues greater than 1.0 and were evaluated using a 0.40 threshold, meaning variables above this value were considered strong contributors to a given. After testing multiple factor solutions, a six-factor model was selected as the best fit for the data. This model effectively captured the relationships among key study variables while maintaining interpretability.

Since programming languages often share similarities, factor analysis was performed to categorize them. Once again, the Kaiser-Meyer-Olkin (KMO) measure of sampling adequacy and Bartlett's test of sphericity were employed to evaluate the dataset's suitability for factor

analysis. After testing various factor solutions, a 10-factor model was chosen as the best fit for programming language data.

A correlation matrix was created to assess the relationships between the primary study variables. The analysis examined the correlations among key predictors used in the confirmatory study: AI code-generation usage, developer experience, the number of programming languages used, the developer's country, and the time spent searching for answers to programming questions. In addition, the correlation matrix included control variables such as age, work location, organization size, education level, frequency of interruptions, and country population.

To better capture the broader context of developer behavior, variables representing the usage of specific AI code-generation tools (e.g., Codeium, GitHub Copilot, Tabnine) and programming language groups (e.g., Frontend JavaScript languages, Systems languages, Mobile languages) were also included. This extended matrix provided a more comprehensive view of the potential interactions among technical, demographic, and geographic factors.

Descriptive statistics, including mean and standard deviation, were presented alongside the correlations. Significance was assessed at the 5%, with p-values below 0.05 indicating statistically significant relationships. The analysis used the `dplyr` and `Hmisc` packages in R (Harrell Jr, 2003; Wickham et al., 2023). This correlation analysis provided essential insights into the strength and direction of associations among the study variables, helping to inform the subsequent linear regression modeling.

Linear regression was used to identify correlations between AI code generation (AID) usage and the time spent searching for answers to programming questions (TSX). Additionally, linear regression evaluates the relationship among covariates. This methodology offers a predictive model that estimates how AI tool usage impacts search time while interpreting AI's

influence on developer productivity. The R stats package supported the linear regression analysis (R Core Team, 2024).

The model was tested for assumptions to ensure the validity of the regression results, including linearity, independence of errors, and homoscedasticity (Raza et al., 2023). The Breusch-Pagan test indicated significant heteroscedasticity in the residuals, which violates the assumption of constant variance. The regression coefficients were recalculated using robust standard errors with the `coeftest` function from the `lmtest` and `sandwich` packages (Hyndman & Khandakar, 2008; Zeileis, 2004). This adjustment ensures that the standard errors are robust against heteroscedasticity, making the p-values and confidence intervals more reliable. Using robust standard errors was necessary because heteroscedasticity can lead to underestimated or overestimated standard errors, potentially invalidating hypothesis tests and the interpretation of results.

Moderating Effects

This study analyzes the main effect of AI code generation on search time. It examines how specific moderating variables- Country of Respondent, Years of Experience, and Number of Programming Languages Used- affect this relationship. By exploring these moderating effects, the study aims to understand how contextual factors may influence the impact of AI tool adoption on developer productivity. The following hypotheses are tested for moderation:

- H2: *Developer experience (years of professional software development) will moderate the relationship between AI code-generation tool use and time spent searching for answers, such that developers with fewer years of experience will experience a stronger reduction in search time.*

- H3: *The number of programming languages a developer uses will moderate the relationship between AI code-generation tool use and time spent searching for answers, such that developers familiar with more programming languages will experience a stronger reduction in search time.*

Multiple linear regression models analyzed the interactions between AI code generation usage (AID) and each moderator. The analysis evaluates whether developers with more experience, those who use more programming languages, or those from specific countries derive greater or lesser time savings from AI tools. By quantifying how each predictor variable and interaction term influences the outcome variable, the regression analysis offers more profound insights into the nuances of these relationships.

This approach is critical for interpreting how AI code generation and associated factors impact developer productivity. It enables the study to capture the complexities of real-world developer experiences, shedding light on whether the benefits of AI tools are universally consistent or contextually dependent on demographic or experiential factors.

Control Variables

This section extends the confirmatory analyses by examining additional factors that may influence developers' productivity and search behaviors. These analyses provide deeper insights into the complex interplay of technical, demographic, and organizational variables, offering a broader understanding of factors affecting AI tool adoption and developer efficiency.

A series of linear regression models were created to explore further factors influencing AI code-generation tool usage and its effect on time spent searching for answers to programming

questions. These models assess the relationships between demographics, AI tool usage, programming languages, and their interactions to identify key predictors of developers' search behavior. These models include:

- Confirmatory – Measures the predictors for the main and moderating effects without interactions.
- Demographic Information – Adds control variables to the model.
- Demographic Information and AI Tools – Builds on the previous linear regression model, adding the AI code-generation tools to the model.
- Demographic Information, AI Tools, and Programming Languages—This section adds programming language groups and individual languages to the model.
- Demographic Information with Interactions – This model uses the variables for the main and moderating effects, as well as control variables. Interactions are added for AI code-generation usage.
- Demographic Information, AI Tools, and Interactions – Builds on the previous model by adding each AI code-generating tool interaction.
- Demographic Information, AI Tools, Programming Languages, and Interactions – Builds on the previous model by adding programming language groups, individual programming languages, and interactions with AI code-generation usage to the model.
- Demographic Information, AI Tools, Programming Languages, and Interactions – This final model adds interactions between AI code-generation usage, AI tools, and programming languages. This model aims to determine if certain combinations of AI tools and programming languages significantly affect search time.

Each of these regression models contributes to a more granular understanding of the conditions under which AI tools influence search behavior, helping to identify whether certain demographic groups, programming language preferences, or AI tool interactions significantly affect the time developers spend searching for answers.

Additional Demographic Analysis

Demographic analysis builds upon the confirmatory findings by investigating additional factors influencing developers' productivity and search behaviors. Research question 2 asks, "Do any demographic characteristics moderate the effect of AI code generation on time searching for answers to programming questions?" The previous correlation matrix and linear regression models investigate the conceptual model's relationships and additional demographic features. These analyses seek to provide deeper insights into the complex interplay of technical, demographic, and organizational variables, offering actionable recommendations for optimizing AI tool adoption in development contexts.

Results

Descriptive Statistics

Table 2 summarizes the descriptive statistics for time spent searching for programming solutions with and without AI code-generation tools. Developers using AI tools spent 67.3 minutes searching, compared to 66.1 minutes for those not using these tools. The dataset contained 11,364 respondents who used AI code generation and 21,617 who did not. Standard deviations were 44.8 for both groups. Both groups had a minimum of 15 minutes, 180 minutes,

and a median of 60 minutes spent searching for answers to programming questions. Figure 4 represents the box plot for the descriptive statistics.

Moderator and Control Variable Factor Analysis

Factor analysis was conducted using principal axis factoring with varimax rotation. The Kaiser-Meyer-Olkin (KMO) measure of sampling adequacy was 0.54, below the standard recommended threshold of 0.6 (Shrestha, 2021). However, with a sample size of over 30,000, the KMO measure of 0.54 is acceptable (Shrestha, 2021). Bartlett's test of sphericity was significant ($p < 0.001$), indicating sufficient correlations among variables for factor analysis (Shrestha, 2021). A six-factor solution was selected, explaining 37.3% of the variance. While the MSA is lower than ideal, removing weaker variables was not feasible, as they are essential to the study's hypotheses and conceptual model. Given the interpretability of the extracted factors and theoretical alignment with prior research, the factor analysis was deemed suitable for further analysis. Table 3 includes the factor analysis for confirmatory and control variables.

The first factor, Experience and age, includes strong loadings for developer experience and age, reflecting the relationship between a developer's years of experience and their age. This factor aligns with the expectation that older developers tend to have more experience. Due to this factor, experience is used in the linear regression models, and age is not used.

The second factor, Economic and country Influence, captures the effects of external economic and regional factors, with the country of origin and economic conditions loading onto this factor. These variables suggest that a developer's location and economic environment may shape their professional behavior. Even though Economy had a slightly higher factor, Country is used in linear regression in place of Economy since Country is part of the conceptual model.

The third factor is dominated by AI Code-Generation Usage at 0.987 and will be in its group. The fourth factor indicates that the population is higher than the organization size and country at 0.570 and will, therefore, be in its group. The fifth factor will include only Interaction Frequency since it had a factor of 0.522.

Programming Language Factor Analysis

Factor analysis was also conducted on the programming languages in the survey using principal axis factoring with varimax rotation. The Kaiser-Meyer-Olkin (KMO) measure of sampling adequacy was 0.56, which is acceptable based on the number of respondents (Shrestha, 2021). Bartlett's test of sphericity was significant ($p < 0.001$), indicating sufficient correlations among variables for factor analysis (Shrestha, 2021). A ten-factor solution was selected, explaining 22.2% of the variance. Given the interpretability of the extracted factors with programming language usage, the factor analysis was deemed suitable for further analysis. Table 4 includes the factor analysis for programming languages.

The first factor includes strong loadings for JavaScript and TypeScript, which are often used in web development. Therefore, they will be grouped in Frontend JavaScript Languages. The second factor saw a surprisingly strong relationship between Rust and R. Therefore, those languages will be grouped as the Rust & R group. In addition, factor seven indicated a strong relationship between Elixir and Erlang. These two languages are combined into a factor named Functional Languages. Based on factor eight, Swift and Kotlin are combined in a Mobile Languages group. PowerShell and Assembly have high loadings in similar factors and are grouped in Scripting Languages. Finally, C and C++ are grouped into System Languages.

Grouping languages into factors simplifies the interpretation of linear regression results by reducing the number of individual predictors, which helps avoid multicollinearity and overfitting. Instead of analyzing dozens of languages separately, factor groupings allow for a more generalizable and meaningful analysis of trends in programming language usage. This approach also improves the statistical power of the model by focusing on broader categories rather than granular differences that may not significantly impact the outcome.

Correlation Matrix

The correlation matrix, summarized in Tables 5-7, explored relationships between key variables, including time searching, AI code-generation usage, developer experience, number of programming languages used, country of the developer, AI code-generation tools, and programming languages. Results indicated a minimal but significant correlation between AI code-generation usage and time searching ($r = 0.012$, $p = 0.029$). In addition, developer experience ($r = -0.124$, $p < 0.001$) was significantly correlated with time searching. Furthermore, the number of programming languages used ($r = 0.026$, $p < 0.001$) was significantly correlated with time searching, indicating that these factors may play a more substantial role in search time. Interestingly, AI usage showed a weak but moderately significant negative correlation with the country ($r = -0.009$, $p = 0.085$), suggesting some contextual variations in the adoption or effectiveness of AI tools. This interaction will be analyzed further with linear regression for potential moderating effects. Respondents had a median of 10.76 years of work experience and used a median of 5.21 languages.

Linear Regression Models

A series of linear regression models were conducted to investigate the relationship between AI code-generation tool usage and developers' time searching for programming solutions. Standardized beta coefficients (β) and robust standard errors (SE) are reported.

The initial set of models (Table 8) focused on core variables without interaction terms. Across all models, AI code-generation usage was not a statistically significant predictor of search time. Developer experience consistently showed a strong negative association with search time ($\beta \approx -0.125^{***}$), indicating that as developers gain professional experience, they spend less time searching for answers. The number of programming languages used was a significant positive predictor in early models but became less consistent once control variables were introduced.

Control variables, including organization size ($\beta \approx 0.036^{***}$), work location ($\beta \approx -0.017^{**}$), frequency of interruptions ($\beta \approx 0.136^{***}$), and country population estimate ($\beta \approx -0.016^{**}$), emerged as significant predictors. These findings suggest that larger organizations, more frequent interruptions, and smaller country populations are associated with variations in developer search behavior.

When specific AI code-generation tools were included, Tabnine usage was a significant positive predictor ($\beta \approx 0.018^{**}$), indicating that developers using Tabnine spent more time searching for answers than developers using other tools.

The following models expanded by including programming language categories. Developers who worked primarily with Frontend JavaScript languages ($\beta \approx -0.023^{**}$), Functional languages ($\beta \approx -0.025^{***}$), Mobile languages ($\beta \approx -0.014^{*}$), PHP ($\beta \approx -0.012^{*}$), and Go ($\beta \approx -0.013^{*}$) spent significantly less time searching. Conversely, developers using Python ($\beta \approx 0.026^{***}$) and Ruby ($\beta \approx 0.014^{*}$) spent considerably more time searching for answers.

Table 9 presents the results of models incorporating interaction terms. Significant interactions were observed between AI code-generation usage and:

- Developer Experience ($\beta \approx 0.022^*$ to 0.024^*),
- Country ($\beta \approx 0.031^{**}$ to 0.033^{**}),
- Country Population ($\beta \approx 0.020^*$ to 0.022^{**}).

These interactions suggest that the effect of AI code-generation usage on search time varies depending on the developer's experience level and country context.

Further interactions between AI code-generation usage and AI tools revealed that Codeium usage was associated with greater search time efficiency when paired with certain language groups. Specifically, a significant negative interaction was observed between Codeium and Systems languages ($\beta \approx -0.055^{***}$), suggesting developers using Codeium alongside Systems languages (e.g., C, C++) experienced reduced search time.

Additional significant interactions included:

- Codeium \times SQL ($\beta \approx 0.021^*$),
- GitHub Copilot \times Rust and R ($\beta \approx 0.061^{**}$),
- GitHub Copilot \times Ruby ($\beta \approx -0.059^*$),
- GitHub Copilot \times Go ($\beta \approx -0.039^*$),
- Replit Ghostwriter \times Ruby ($\beta \approx -0.013^*$).

No significant interactions were found between Tabnine and Whispr AI across programming language groups.

Overall, the findings suggest that while AI code-generation tools do not universally reduce developer search time, their effectiveness can vary depending on the developer's experience, geographic location, AI tool used, and primary programming languages.

Curvilinear Relationship

To examine the possibility of a curvilinear relationship between AI code-generation usage and time spent searching for programming solutions, a regression model was estimated including a squared term for AI usage (AID^2). However, because AI usage was measured as a binary variable (0 = no usage, 1 = usage), the squared term was perfectly collinear with the original variable and could not be estimated. Thus, a curvilinear relationship could not be assessed in this case, and the relationship between AI usage and time searching appears to be adequately captured as linear.

Research Questions and Hypotheses Discussion

Research Question One

How does AI code generation impact the time developers spend searching for programming solutions?

The analysis revealed that AI code-generation tools, while designed to improve developer productivity, do not significantly impact the time developers spend searching for answers to programming questions based on this dataset. Descriptive statistics, correlation analysis, and linear regression consistently showed no statistically significant reduction in search time associated with AI code-generation tool usage.

Hypothesis One

Developers using AI code-generation tools will spend less time searching for answers to programming questions.

The AI code-generation tool usage coefficient was not statistically significant across the regression models. As a result, the study fails to reject the null hypothesis for Hypothesis One. These findings suggest that AI tools may provide other forms of assistance but do not directly reduce the time spent searching for programming solutions.

Research Question Two

Do developer characteristics moderate the relationship between AI code-generation tool usage and time searching for programming solutions?

Developer experience and the number of programming languages used during the survey were explored as potential moderators. Developer experience consistently demonstrated a strong negative association with search time, implying that more experienced developers benefit from accumulated knowledge and require less time to find programming solutions. In contrast, while the number of programming languages used showed some significance individually, it did not moderate the effect of AI code-generation tool usage on search time.

Additionally, country-level factors emerged as necessary. Developers from smaller-population countries experienced greater benefits from AI code-generation tools regarding reduced search time, while developers from larger-population countries did not show the same gains.

Hypothesis Two

Developer experience will moderate the relationship between AI code-generation tool usage and time spent searching for answers, such that developers with fewer years of experience will experience a more substantial reduction in search time.

Years of professional experience emerged as a significant moderator in the multiple linear regression models. Less experienced developers benefited more from AI code-generation tool usage than experienced developers. Therefore, the study rejects the null hypothesis for Hypothesis Two.

Hypothesis Three

The number of programming languages developers use will moderate the relationship between AI code-generation tool usage and time spent searching for answers.

No statistically significant moderating effect was observed for the number of programming languages used during the survey. Therefore, the study fails to reject the null hypothesis for Hypothesis Three.

Chapter Five – Exploratory Analysis

Chapter 5 builds upon the confirmatory findings in Chapter 4, delving into exploratory analyses to investigate additional factors influencing developers' productivity and search behaviors. Research question 2 asks, "Do any demographic characteristics moderate the effect of AI code generation on time searching for answers to programming questions?" These analyses seek to provide deeper insights into the complex interplay of technical, demographic, and organizational variables, offering actionable recommendations for optimizing AI tool adoption in development contexts.

Furthermore, research question 3 asks, "What other factors contribute to the effect of AI code generation on time spent searching for answers to programming questions?" Random Forest is employed to predict the effect of various features on time spent searching for answers to programming questions. Random forest is a machine learning method known for its ability to handle nonlinear relationships and is particularly good at regression (Wright & Ziegler, 2017). By using behavioral, demographic, and organizational features, this analysis looks for patterns to predict the effect of AI code generation on developer productivity.

Lastly, country, population, and economic status were significant modifiers in Chapter 4. This chapter explores the relationship between population and its moderating effect on AI code generation for time searching. Specifically, how does changing the population affect the effectiveness of AI code generation for time searching?

Analysis

Predictors of AI Code Generation Effectiveness

This study employs Random Forest modeling to predict developers' time searching for programming solutions (TSX) based on demographic, organizational, and behavioral factors (Wright & Ziegler, 2017). Random Forest is well-suited for modeling complex, nonlinear relationships and is adequate for classification and regression tasks.

Based on the linear regression models from Chapter 4, the predictor variables in the model include the confirmatory variables of developer experience in years and number of programming languages used. Since population showed more significance in linear regression, the random forest uses it over the respondent's country. In addition, control variables of work location, organization size, education level, and frequency of interruptions are used. Each AI code generation tool is used as a predictor in the random forest. Finally, programming language groups and individual programming languages based on factor analysis are also predictors in the random forest.

AI Code-Generation Tools

This analysis evaluates the performance of specific AI code-generation tools based on data from the 2023 and 2024 Stack Overflow Developer Surveys. By calculating and comparing the mean time spent searching for programming solutions (TSX) among users of each tool, this analysis identifies tools associated with the most significant efficiency gains.

This tool-level analysis supplements the findings from linear regression and Random Forest models by:

- Highlighting tools that consistently reduce search time.

- Identifying significant variations in tool performance.
- Providing insights into adoption trends and tool-specific usage patterns.

Descriptive statistics summarize tool effectiveness, offering a practical understanding of which AI tools deliver the most value in reducing developers' search times. The analysis supports strategic tool selection and highlights opportunities for improving AI tools to meet developer needs.

Country

This section also explores the impact of AI code generation at the country and regional levels to uncover variations in developer productivity across geographic locations. By calculating average time savings with and without AI tools for each country, the study provides a nuanced view of how regional contexts influence the effectiveness of AI tools.

The steps for the analysis are as follows:

1. **Filter by Country:** Countries with fewer than 30 respondents are excluded to ensure the reliability of the results. This threshold, based on statistical power considerations (VanVoorhis & Morgan, 2007), improves the validity of the analysis.
2. **Calculate Average Time Savings:** For each remaining country, the average time spent searching for solutions (TSX) is calculated separately for AI tool users ($AID = 1$) and non-users ($AID = 0$). This process allows for direct comparison of search times between the two groups.

3. **Compute Time Savings:** The average time savings for each country is computed by subtracting the mean TSX for AI users from the mean TSX for non-users. Positive values indicate that AI tools reduce search time.
4. **Top 25 Countries by Time Savings:** The top 25 countries with the highest average time savings are identified and visualized to highlight regions where AI tools significantly impact productivity.
5. **World Map Visualization:** A world map is created using the `sf` and `rnatrualearth` packages (Massicotte & South, 2024; Pebesma, 2018), with a color scale from the `Viridis` package (Garnier et al., 2023) to illustrate geographic variation in time savings. Countries with insufficient data are grayed out, while brighter colors indicate more significant time savings.

This approach offers a global perspective on AI tools' productivity impact, identifying regions where adoption strategies can be optimized and highlighting potential barriers to effectiveness.

Results

Predictors of AI Code Generation Effectiveness

The model uses a training dataset (70%) and a test dataset (30%), with 1,000 trees trained for stability in variable importance results. The `ranger` package was used to implement Random Forest quickly and efficiently (Wright & Ziegler, 2017). After training, variable importance scores were plotted to identify the most influential factors in predicting TSX. Higher scores reflect more substantial predictive power, highlighting key contributors to developers' search

time. Figure 6 presents these results in a bar chart, illustrating the relative importance of each predictor variable.

The Random Forest analysis identified years of professional experience (4.8m) as the most influential predictor of time spent searching for answers to programming questions. Interaction frequency (4.5m) and country population (4.4m) were second and third in importance, respectively. The number of programming languages used (3.2m) and organization size (2.7m) finish the top five. Education level (1.9m) and work location (1m) are the following two factors of importance.

SQL was the most influential programming language (829k), and GitHub Copilot was the most influential AI code-generation tool (827k). Several programming languages follow in order of importance, including Rust/R (666k), system languages (662k), PHP (653k), Go (634k), frontend JavaScript languages (597k), scripting languages (585k), Java (564k), mobile languages (553k), and Python (538k). AI tools Tabnine (359k) and Codeium (255k) are next. Functional programming languages (252k) are the final language group. The AI tools Whispr AI (65k) and Replit Ghostwriter (10k) are the final factors in order of importance.

AI Code-Generation Tools

Table 10 summarizes the counts and mean TSX for commonly used AI code-generating tools, and Figure 7 visualizes the mean TSX broken down by year. With 9,928 users, GitHub Copilot has the lowest mean TSX, at approximately 66.7 minutes, reinforcing its popularity and potential productivity benefits. Other tools like Codeium and Tabnine show considerable usage with slightly higher mean TSX values. These results indicate moderate efficiency gains.

Conversely, Replit Ghostwriter and Whispr AI have smaller user bases and higher mean TSX values (around 72 minutes), which may indicate lower effectiveness or limited adoption. However, smaller sample sizes for these tools warrant caution when drawing firm conclusions. These findings reveal that GitHub Copilot leads in both adoption and search time reduction, while other tools like Codeium and Tabnine offer meaningful, albeit less pronounced, productivity improvements.

Country

Linear regression indicated that the country is a significant moderator of AI code generation and time searching. Therefore, the country was analyzed in more depth by calculating the average time spent searching (TSX) for respondents using AI tools (AID = 1) and those not using them (AID = 0) across countries. The difference in mean TSX between these groups represents the average time savings attributed to AI tool usage within each country. Only countries with at least 30 respondents were included to ensure reliable results. The analysis identified the top 25 countries with the highest time savings, highlighting variations in AI tool efficiency across national contexts.

Table 11 summarizes these results, showing Morocco, Belgium, Croatia, and Indonesia as the top four countries with the most significant time savings. The table reveals a wide range of time savings across countries, from nearly 12 minutes in Morocco to just under 2 minutes in Ireland. These findings indicate substantial variability in how developers from different countries benefit from AI code generation tools. Figure 8 presents a bar chart of the top 25 countries in the table.

Figure 9 provides a world map visualization of these differences to better visualize each country's time-saving region. The world map offers a clear geographic perspective on the global impact of AI tools on developer productivity. Countries colored in blue have the highest time savings, and countries highlighted in red lose time using AI code generation. One interesting discovery is that countries in the Malay Archipelago are on extreme ends of time savings based on AI code-generation usage. Indonesia is among the countries with the highest time savings, but the bordering country, Malaysia, is among the lowest.

Multiple linear regression models indicated that the population was a significant moderator of AI code generation and time searching. Therefore, more analysis was performed on the population. This interaction between AI code generation and country population provides insight into how the benefits of AI-assisted coding vary by geographic and economic context. The plot in Figure 10 demonstrates that developers in smaller-population countries see the most significant reduction in search time when using AI tools. In contrast, those in larger population countries do not experience the same level of efficiency gains. One possible explanation is that developers in smaller countries rely more on AI-generated solutions due to fewer available local resources, such as in-person mentorship, collaborative development environments, or extensive developer communities. In contrast, larger populated countries may have greater access to knowledge-sharing opportunities, structured learning programs, and developer networks, potentially reducing the relative impact of AI assistance.

Additionally, the diminishing effect of AI in more populated countries could be influenced by factors such as education systems, workplace AI adoption policies, and cultural attitudes toward AI-driven development. Developers may already have access to robust information repositories in nations with established technology industries, making AI less

impactful. In contrast, AI might be a more valuable tool in emerging markets with smaller developer communities in bridging knowledge gaps and accelerating learning. These findings suggest that the adoption and effectiveness of AI tools in programming may not be uniform globally and that population size may play a crucial role in shaping the return on investment for AI-assisted coding.

Chapter Six - Discussion

This study examined how AI code-generation tools impact professional developers' time searching for programming solutions and how factors such as developer experience and the number of programming language used by the developer moderate that relationship. Using archival data from the 2023 and 2024 Stack Overflow Developer Surveys, the study found that while AI code-generation tools alone do not significantly reduce search time, experience and other developer characteristics strongly influence outcomes. Developers with more experience spent less time searching overall, and AI tools offered greater time-saving benefits to less experienced developers. Exploratory and predictive analyses also identified the country population, frequency of interruptions, and the number of programming languages used as key factors contributing to search time. These findings suggest that AI code-generation tools affect developer productivity in nuanced ways, shaped by demographic, organizational, and technical variables.

Interpretation of Confirmatory Hypotheses

The results provided mixed support for the study's confirmatory hypotheses. Hypothesis 1 predicted that developers using AI code-generation tools would spend less time searching for programming solutions than those not using such tools. However, this hypothesis was not supported. AI code-generation usage alone did not significantly reduce search time among professional developers. Hypothesis 2 proposed that developer experience would moderate the relationship between AI tool use and search time, with less experienced developers benefiting more. This hypothesis was supported providing evidence that less experienced developers showed greater reductions in search time when using AI tools. Hypothesis 3 suggested that the

number of programming languages used would moderate the relationship between AI tool use and search time, with developers familiar with more languages experiencing greater time savings. Contrary to expectations, greater language familiarity was associated with increased search time, possibly due to more complex problem-solving demands. Overall, these findings suggest that the effects of AI code-generation tools on developer productivity are moderated by individual differences rather than being uniformly beneficial.

Exploratory Findings

Exploratory analyses revealed additional factors influencing developers' time searching for programming solutions. Work location emerged as a significant factor, with developers who spent less time working onsite experiencing reduced search times. Increased frequency of workplace interactions and higher education levels were associated with longer search times, suggesting that greater collaboration or academic training may introduce more complex problem-solving demands. Specific programming languages also showed differences in search behaviors: Python usage was associated with longer search times, while familiarity with Frontend JavaScript languages, Rust, R, and PHP was linked to shorter search times. Additionally, organization size demonstrated a moderate relationship with search behaviors, indicating that developers in larger, more structured environments may search differently than those in smaller organizations.

Comparison to Prior Research

The findings of this study align with and extend previous research on developer productivity and AI tool adoption. Prior studies have suggested that AI code-generation tools,

such as GitHub Copilot and ChatGPT, can enhance developer efficiency by reducing the time spent searching for solutions (Bird et al., 2022; Kalamvakou, 2023). However, the results of this study indicate that AI tool usage alone does not significantly reduce search time for professional developers, highlighting a more complex relationship than initially anticipated. These results are consistent with prior research noting that the adoption and effectiveness of new technologies can be moderated by individual and contextual factors (Besker et al., 2020; Rifat & Viitanen, 2021).

In particular, the finding that less experienced developers benefit more from AI tools aligns with technology adoption theories, suggesting that perceived usefulness varies by user characteristics (Morris & Venkatesh, 2000). Additionally, the exploratory results regarding work location and organization size reflect broader trends in the literature emphasizing the influence of organizational environment on developer workflows (Forsgren et al., 2021). Overall, this study reinforces the growing consensus that the impact of AI code-generation tools on productivity is multifaceted, dependent on demographic, technical, and contextual variables rather than being universally beneficial across all users.

Limitations of Study

Several limitations should be considered when interpreting the results of this study. First, the use of archival data from the 2023 and 2024 Stack Overflow Developer Surveys imposed constraints on the scope and depth of the analysis. Because the survey questions and response options were not explicitly designed for this research, some relevant factors may not have been captured, limiting the ability to examine certain variables of interest. Additionally, reliance on self-reported data introduces the potential for respondent bias.

Second, although the study controlled for key demographic and technical variables such as experience, programming languages, and geographic context, unmeasured factors, including organizational culture, team workflows, and specific work environments, may have influenced the observed relationships. These unmeasured variables could affect the generalizability of the findings across different developer populations.

Third, ordinary least squares (OLS) regression was employed as the primary method of hypothesis testing. While OLS offers broad applicability and ease of interpretation, alternative approaches such as ordinal logistic regression, generalized estimating equations (GEE), or structural equation modeling (SEM) could enhance model robustness in future research. OLS remained an appropriate choice in this study, given the research questions and study design.

Finally, although the analyses revealed associations between AI tool use, developer characteristics, and time spent searching for programming solutions, causal inferences cannot be drawn. Future research employing experimental designs, real-time usage tracking, and broader sampling across underrepresented groups and tools is necessary to deepen understanding of AI's evolving impact on developer productivity.

Practical Implications

The findings of this study have important implications for developers, organizations, and AI tool designers. First, organizations should tailor AI tool adoption and training initiatives based on developer experience levels. While less experienced developers may benefit significantly from AI code-generation tools, experienced developers may derive greater value from tools optimized for advanced workflows. Targeted deployment and training strategies can help maximize productivity gains across diverse developer populations.

Second, the influence of geographic factors suggests that organizations operating globally should account for regional differences in infrastructure, education, and access to resources when deploying AI tools. Local support, customized training, and region-specific enhancements may help close adoption and productivity gaps across international teams.

Third, beyond focusing solely on performance metrics such as time savings, organizations should also consider broader dimensions of developer productivity, including satisfaction and collaboration. AI tool integration strategies emphasizing ease of use, intuitive interfaces, and support for collaborative workflows are likely to yield more sustainable improvements.

Finally, tool developers should prioritize expanding language support, improving tool capabilities across diverse programming environments, and designing AI features that accommodate varying levels of developer expertise. Addressing these practical considerations can help ensure that AI code-generation tools better align with the complex realities of modern software development environments.

Theoretical Implications

Although this study was not grounded in a formal theoretical framework, it contributes empirical insights to the emerging literature on AI code-generation tool adoption and developer productivity. The study provides foundational evidence that can inform future theory development by examining real-world patterns in professional developers' behaviors across diverse contexts.

The findings highlight the importance of considering individual, organizational, and contextual moderators when evaluating the impact of AI tools on developer workflows. Future

research could build formal theoretical models by exploring factors such as team dynamics, organizational culture, and task complexity to understand better the conditions under which AI tools enhance or inhibit productivity outcomes. In doing so, scholars can develop more nuanced theories reflecting AI adoption's multifaceted realities in software development environments.

Recommendations For Future Research

Building on the findings and limitations of this study, several areas warrant further investigation to deepen understanding of AI code-generation tools and their impact on developer productivity. First, future research should prioritize greater geographic diversity by including developers from underrepresented regions. Exploring variations based on education systems, infrastructure, and resource availability could uncover significant differences in AI tool effectiveness across global contexts. Understanding why developers in some countries benefit more than others from AI code-generation tools remains a critical question.

Second, longitudinal studies are needed to examine how the impacts of AI tool usage evolve. Investigating whether productivity gains plateau, accelerate, or diminish with sustained AI adoption would offer valuable insights for developers, organizations, and tool designers.

Third, future studies should expand beyond individual developer outcomes to explore team-based dynamics. Examining how AI tools influence collaboration, knowledge sharing, and overall team productivity would help capture the broader organizational impacts of AI integration.

Additionally, research should examine tool-specific features to identify which AI functionalities most significantly enhance productivity. Such insights could inform better tool design, training programs, and deployment strategies.

Finally, expanding the scope of productivity measurement remains essential. Future studies should incorporate broader outcomes such as code quality, maintainability, and developer satisfaction, aligning with comprehensive productivity frameworks like SPACE to better understand AI's influence on software development work. By addressing these areas, future research can help maximize the value of AI code-generation tools across diverse technical, demographic, and organizational settings.

Conclusion

This study advances the understanding of how AI code-generation tools influence developer productivity, highlighting the importance of individual, technical, and contextual factors in shaping outcomes. While AI tools offer promise, their impact is not uniform, and their benefits are moderated by characteristics such as developer experience, number programming languages used, and geographic context. By combining confirmatory, exploratory, and predictive analyses, this research provides a foundation for future scholarly work and practical innovation at the intersection of AI and software development. Continued investigation into the evolving role of AI tools will be critical for maximizing their potential and ensuring they contribute meaningfully to the productivity and satisfaction of developers worldwide.

Broader Reflections on AI, Creativity, and Human Ingenuity

Ada Lovelace, often regarded as the world's first programmer, envisioned computing not as a replacement for human thought but as a tool for amplifying human understanding and creativity (Lovelace & Menebrea, 1843). In many ways, AI code-generation tools today extend that vision. They automate routine aspects of programming, freeing developers to engage more

deeply with complex, creative challenges, such as augmenting rather than replacing human expertise. This study's findings highlight the ongoing relevance of Lovelace's insight: when thoughtfully deployed, technology enhances the developer's role rather than diminishes it.

Lord Byron, Lovelace's father and one of the leading voices of the Romantic era, famously wrote of finding joy and meaning in the untamed landscapes of nature: "There is pleasure in the pathless woods, there is a rapture on the lonely shore..." (Byron, 1818). His words resonate with the evolving landscape of AI in software development. Just as Byron celebrated the exploration of the unknown, modern developers and organizations must navigate the unfamiliar terrain of AI with curiosity, resilience, and a commitment to preserving human creativity at the center. As AI continues to reshape professional landscapes, embracing its potential and limitations will be critical to ensuring technology remains a catalyst for innovation rather than a replacement for it.

References

- Albrecht, A. J., & Gaffney, J. E. (1983). Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, *SE-9*(6), 639–648. <https://doi.org/10.1109/TSE.1983.235271>
- Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2019). A Survey of Machine Learning for Big Code and Naturalness. *ACM Computing Surveys*, *51*(4), 1–37. <https://doi.org/10.1145/3212695>
- Arora, A., & Gambardella, A. (2005). The Globalization of the Software Industry: Perspectives and Opportunities for Developed and Developing Countries. *Innovation Policy and the Economy*, *5*, 1–32. <https://doi.org/10.1086/ipe.5.25056169>
- Asare, O., Nagappan, M., & Asokan, N. (2024). *Is GitHub's Copilot as Bad as Humans at Introducing Vulnerabilities in Code?* (arXiv:2204.04741). arXiv. <http://arxiv.org/abs/2204.04741>
- Backus, J. W., Stern, H., Ziller, I., Hughes, R. A., Nutt, R., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., & Sheridan, P. B. (1957). The FORTRAN automatic coding system. *Papers Presented at the February 26-28, 1957, Western Joint Computer Conference: Techniques for Reliability on - IRE-AIEE-ACM '57 (Western)*, 188–198. <https://doi.org/10.1145/1455567.1455599>
- Baltes, S., & Diehl, S. (2018). Towards a theory of software development expertise. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 187–200. <https://doi.org/10.1145/3236024.3236061>

- Banker, R. D., Davis, G. B., & Slaughter, S. A. (1998). Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study. *Management Science*, 44(4), 433–450. <https://doi.org/10.1287/mnsc.44.4.433>
- Barenkamp, M., Rebstadt, J., & Thomas, O. (2020). Applications of AI in classical software engineering. *AI Perspectives*, 2(1), 1. <https://doi.org/10.1186/s42467-020-00005-4>
- Barke, S., James, M. B., & Polikarpova, N. (2022). *Grounded Copilot: How Programmers Interact with Code-Generating Models* (arXiv:2206.15000). arXiv. <http://arxiv.org/abs/2206.15000>
- Beaver, J. M., & Schiavone, G. A. (2006). The effects of development team skill on software product quality. *ACM SIGSOFT Software Engineering Notes*, 31(3), 1–5. <https://doi.org/10.1145/1127878.1127882>
- Beeharry, Y., & Ganoo, M. (2018). Analysis of data from the survey with developers on Stack Overflow: A Case Study. *Journal of Engineering Technology*, 7(2).
- Besker, T., Ghanbari, H., Martini, A., & Bosch, J. (2020). The influence of Technical Debt on software developer morale. *Journal of Systems and Software*, 167, 110586. <https://doi.org/10.1016/j.jss.2020.110586>
- Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., & Gazit, I. (2022). Taking Flight with Copilot: Early insights and opportunities of AI-powered pair-programming tools. *Queue*, 20(6), 35–57. <https://doi.org/10.1145/3582083>
- Bissyande, T. F., Thung, F., Lo, D., Jiang, L., & Reveillere, L. (2013). Popularity, Interoperability, and Impact of Programming Languages in 100,000 Open Source Projects. *2013 IEEE 37th Annual Computer Software and Applications Conference*, 303–312. <https://doi.org/10.1109/COMPSAC.2013.55>

- Byron, G. G. (1818). *Childe Harold's Pilgrimage* [Website]. <https://poets.org/poem/childe-harolds-pilgrimage-there-pleasure-pathless-woods>
- Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., ... Zaremba, W. (2021). *Evaluating Large Language Models Trained on Code* (arXiv:2107.03374). arXiv. <http://arxiv.org/abs/2107.03374>
- Coccia, M. (2014). Driving forces of technological change: The relation between population growth and technological innovation. *Technological Forecasting and Social Change*, 82, 52–65. <https://doi.org/10.1016/j.techfore.2013.06.001>
- Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). *Natural Language Processing (almost) from Scratch* (arXiv:1103.0398). arXiv. <https://doi.org/10.48550/arXiv.1103.0398>
- Coupé, C., Oh, Y. M., Dediu, D., & Pellegrino, F. (2019). Different languages, similar encoding efficiency: Comparable information rates across the human communicative niche. *Science Advances*, 5(9), eaaw2594. <https://doi.org/10.1126/sciadv.aaw2594>
- Coutinho, M., Marques, L., Santos, A., Dahia, M., França, C., & De Souza Santos, R. (2024). The Role of Generative AI in Software Development Productivity: A Pilot Case Study. *Proceedings of the 1st ACM International Conference on AI-Powered Software*, 131–138. <https://doi.org/10.1145/3664646.3664773>
- Crossley, T. F., Krishna Pendakur, & Pendakur, K. (2006). *The social cost-of-living: Welfare foundations and estimation*. <https://doi.org/10.1920/wp.ifs.2006.0610>

- Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., Ming, Z., & Jiang. (2023). *GitHub Copilot AI pair programmer: Asset or Liability?* (arXiv:2206.15331). arXiv. <http://arxiv.org/abs/2206.15331>
- Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding* (arXiv:1810.04805). arXiv. <https://doi.org/10.48550/arXiv.1810.04805>
- Dohmke, T., Iansiti, M., & Richards, G. (2023). *Sea Change in Software Development: Economic and Productivity Analysis of the AI-Powered Developer Lifecycle* (arXiv:2306.15033). arXiv. <https://doi.org/10.48550/arXiv.2306.15033>
- Ebert, C., & Louridas, P. (2023). Generative AI for Software Practitioners. *IEEE Software*, 40(4), 30–38. <https://doi.org/10.1109/MS.2023.3265877>
- Ernst, N. A., & Bavota, G. (2022). AI-Driven Development Is Here: Should You Worry? *IEEE Software*, 39(2), 106–110. <https://doi.org/10.1109/MS.2021.3133805>
- Evtikhiev, M., Bogomolov, E., Sokolov, Y., & Bryksin, T. (2023). Out of the BLEU: How should we assess quality of the Code Generation models? *Journal of Systems and Software*, 203, 111741. <https://doi.org/10.1016/j.jss.2023.111741>
- Feigenspan, J., Kastner, C., Liebig, J., Apel, S., & Hanenberg, S. (2012). Measuring programming experience. *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, 73–82. <https://doi.org/10.1109/ICPC.2012.6240511>
- Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). *CodeBERT: A Pre-Trained Model for Programming and Natural Languages* (arXiv:2002.08155). arXiv. <http://arxiv.org/abs/2002.08155>

- Finnie-Ansley, J., Denny, P., Becker, B. A., Luxton-Reilly, A., & Prather, J. (2022). The Robots Are Coming: Exploring the Implications of OpenAI Codex on Introductory Programming. *Proceedings of the 24th Australasian Computing Education Conference*, 10–19. <https://doi.org/10.1145/3511861.3511863>
- Fisher, J., Alvarez, J., & Taylor, R. (1978). A survey of how practicing programmers keep up-to-date first results including their implications for computer science education. *ACM SIGCSE Bulletin*, 10(3), 67–72. <https://doi.org/10.1145/953028.804236>
- Forsgren, N., Kalliamvakou, E., Noda, A., Greiler, M., Houck, B., & Storey, M.-A. (2023). DevEx in Action: A study of its tangible impacts. *Queue*, 21(6), 47–77. <https://doi.org/10.1145/3639443>
- Forsgren, N., Storey, M.-A., Maddila, C., Zimmermann, T., Houck, B., & Butler, J. (2021). The SPACE of developer productivity. *Communications of the ACM*, 64(6), 46–53. <https://doi.org/10.1145/3453928>
- Fuegi, J., & Francis, J. (2003). Lovelace & babbage and the creation of the 1843 “notes.” *IEEE Annals of the History of Computing*, 25(4), 16–26. <https://doi.org/10.1109/MAHC.2003.1253887>
- Gonzalez-Barahona, J. M., Robles, G., Andradas-Izquierdo, R., & Ghosh, R. A. (2008). Geographic origin of libre software developers. *Information Economics and Policy*, 20(4), 356–363. <https://doi.org/10.1016/j.infoecopol.2008.07.001>
- Harrell Jr, F. E. (2003). *Hmisc: Harrell Miscellaneous* (p. 5.2-2) [Dataset]. <https://doi.org/10.32614/CRAN.package.Hmisc>
- Harrison Oke Ekpobimi, Regina Coelis Kandekere, & Adebamigbe Alex Fasanmade. (2024). The future of software development: Integrating AI and machine learning into front-end

- technologies. *Global Journal of Advanced Research and Reviews*, 2(1), 069–077.
<https://doi.org/10.58175/gjarr.2024.2.1.0031>
- Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., & Steinhardt, J. (2021). *Measuring Coding Challenge Competence With APPS* (arXiv:2105.09938). arXiv. <http://arxiv.org/abs/2105.09938>
- Hopper, G. M. (1952). The education of a computer. *Proceedings of the 1952 ACM National Meeting (Pittsburgh) on - ACM '52*, 243–249. <https://doi.org/10.1145/609784.609818>
- Horne, D. (2023). PwnPilot: Reflections on Trusting Trust in the Age of Large Language Models and AI Code Assistants. *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*, 2457–2464.
<https://doi.org/10.1109/CSCE60160.2023.00396>
- Hou, D., & Wang, Y. (2009). An empirical analysis of the evolution of user-visible features in an integrated development environment. *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research - CASCON '09*, 122.
<https://doi.org/10.1145/1723028.1723044>
- Hucka, M., & Graham, M. J. (2018). Software search is not a science, even among scientists: A survey of how scientists and engineers find software. *Journal of Systems and Software*, 141, 171–191. <https://doi.org/10.1016/j.jss.2018.03.047>
- Hyndman, R. J., & Khandakar, Y. (2008). Automatic Time Series Forecasting: The **forecast** Package for R. *Journal of Statistical Software*, 27(3).
<https://doi.org/10.18637/jss.v027.i03>

Kaggle Stack Overflow Developer Survey 2023. (n.d.). Kaggle. Retrieved March 3, 2025, from <https://www.kaggle.com/datasets/berkayalan/stack-overflow-annual-developer-survey-2023/data>

Kaggle Stack Overflow Developer Survey 2024. (n.d.). Kaggle. Retrieved March 3, 2025, from <https://www.kaggle.com/datasets/berkayalan/stack-overflow-annual-developer-survey-2024/data>

Kaliamvakou, E. (2023, February 17). Research: Quantifying GitHub Copilot's impact on developer productivity and happiness. *The GitHub Blog*. <https://github.blog/2022-09-07-research-quantifying-github-copilots-impact-on-developer-productivity-and-happiness/>

Kim, S., Zhao, J., Tian, Y., & Chandra, S. (2021). Code Prediction by Feeding Trees to Transformers. *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 150–162. <https://doi.org/10.1109/ICSE43902.2021.00026>

Kochhar, P. S., Wijedasa, D., & Lo, D. (2016). A Large Scale Study of Multiple Programming Languages and Code Quality. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 563–573. <https://doi.org/10.1109/SANER.2016.112>

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. Burges, L. Bottou, & K. Q. Weinberger (Eds.), *Advances in Neural Information Processing Systems* (Vol. 25). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

- Lakhanpal, B. (1993). Understanding the factors influencing the performance of software development groups: An exploratory group-level analysis. *Information and Software Technology*, 35(8), 468–473. [https://doi.org/10.1016/0950-5849\(93\)90044-4](https://doi.org/10.1016/0950-5849(93)90044-4)
- Li, M. M., Dickhaut, E., Bruhin, O., Wache, H., & Weritz, P. (n.d.). *More Than Just Efficiency: Impact of Generative AI on Developer Productivity*.
- Liu, J., Tang, X., Li, L., Chen, P., & Liu, Y. (2023). ChatGPT vs. Stack Overflow: An Exploratory Comparison of Programming Assistance Tools. *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, 364–373. <https://doi.org/10.1109/QRS-C60940.2023.00105>
- Liu, Y., He, H., Han, T., Zhang, X., Liu, M., Tian, J., Zhang, Y., Wang, J., Gao, X., Zhong, T., Pan, Y., Xu, S., Wu, Z., Liu, Z., Zhang, X., Zhang, S., Hu, X., Zhang, T., Qiang, N., ... Ge, B. (2024). *Understanding LLMs: A Comprehensive Overview from Training to Inference* (arXiv:2401.02038). arXiv. <http://arxiv.org/abs/2401.02038>
- Lovelace, A., & Menebrea, L. (1843). Sketch of the Analytical Engine Invented by Charles Babbage. *Bibliothèque Universelle de Genève, October 1842*(82), 666–731.
- Massicotte, P., & South, A. (2024). *rnaturalearth: World Map Data from Natural Earth* (Version 1.0.1.9000) [Computer software]. <https://docs.ropensci.org/rnaturalearth/>
- Mastropaolo, A., Pascarella, L., Guglielmi, E., Ciniselli, M., Scalabrino, S., Oliveto, R., & Bavota, G. (2023). On the Robustness of Code Generation Techniques: An Empirical Study on GitHub Copilot. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2149–2160. <https://doi.org/10.1109/ICSE48619.2023.00181>

- McCarthy, J., Minsky, M., Rochester, N., & Shannon, C. (1955). *A proposal for the Dartmouth summer research project on artificial intelligence*. Stanford University.
<http://jmc.stanford.edu/articles/dartmouth/dartmouth.pdf>
- Meyer, A. N., Barr, E. T., Bird, C., & Zimmermann, T. (2021). Today Was a Good Day: The Daily Life of Software Developers. *IEEE Transactions on Software Engineering*, 47(5), 863–880. <https://doi.org/10.1109/TSE.2019.2904957>
- Meyer, A. N., Satterfield, C., Zuger, M., Kevic, K., Murphy, G. C., Zimmermann, T., & Fritz, T. (2022). Detecting Developers' Task Switches and Types. *IEEE Transactions on Software Engineering*, 48(1), 225–240. <https://doi.org/10.1109/TSE.2020.2984086>
- Míšek, J. (2017). *IntelliSense implementation of a dynamic language*.
https://dspace.cuni.cz/bitstream/handle/20.500.11956/94052/RPTX_2017_1_11320_0_564579_0_196610.pdf?sequence=1
- Mitchell, T. M. (2013). *Machine learning* (Nachdr.). McGraw-Hill.
- Moor, J. (2006). The Dartmouth College artificial intelligence conference: The next fifty years. *AI Magazine*, 27(4), 87–87.
- Morris, M. G., & Venkatesh, V. (2000). AGE DIFFERENCES IN TECHNOLOGY ADOPTION DECISIONS: IMPLICATIONS FOR A CHANGING WORK FORCE. *Personnel Psychology*, 53(2), 375–403. <https://doi.org/10.1111/j.1744-6570.2000.tb00206.x>
- Mozannar, H., Bansal, G., Fournay, A., & Horvitz, E. (2024). *Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming* (arXiv:2210.14306). arXiv. <http://arxiv.org/abs/2210.14306>

Open Data Commons Open Database License (ODbL) v1.0—Open Data Commons: Legal tools for open data. (2023). Open Data Commons.

<http://opendatacommons.org/licenses/odbl/1.0/>

Parnin, C., & Rugaber, S. (2011). Resumption strategies for interrupted programming tasks.

Software Quality Journal, 19(1), 5–34. <https://doi.org/10.1007/s11219-010-9104-9>

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., & Karri, R. (2022). Asleep at the Keyboard?

Assessing the Security of GitHub Copilot’s Code Contributions. *2022 IEEE Symposium on Security and Privacy (SP)*, 754–768. <https://doi.org/10.1109/SP46214.2022.9833571>

Project Management Institute (Ed.). (2017). *A guide to the project management body of*

knowledge / Project Management Institute (Sixth edition). Project Management Institute.

R Core Team. (2024). *_R: A Language and Environment for Statistical Computing_ [R]*. R

Foundation for Statistical Computing. <http://www.r-project.org>

Raza, M., Ahmed, M., Razzaque, S., & Hina, H. (2023). Testing for Heteroskedasticity in The Presence of Outliers. *Journal of Education and Social Studies*, 4(2), 313–329.

<https://doi.org/10.52223/jess.2023.4209>

Revelle, W. (2007). *psych: Procedures for Psychological, Psychometric, and Personality*

Research (p. 2.4.12) [Dataset]. <https://doi.org/10.32614/CRAN.package.psych>

Rifat, A., & Viitanen, E. (2021). *How to educate and train software developers to new*

processes? <https://doi.org/10.13140/RG.2.2.30481.35688>

Sadowski, C., Stolee, K. T., & Elbaum, S. (2015). How developers search for code: A case study. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software*

Engineering, 191–201. <https://doi.org/10.1145/2786805.2786855>

Sammet, J. E. (2000). The real creators of Cobol. *IEEE Software*, 17(2), 30–32.

<https://doi.org/10.1109/52.841602>

Sarkar, A., Gordon, A. D., Negreanu, C., Poelitz, C., Ragavan, S. S., & Zorn, B. (2022). *What is it like to program with artificial intelligence?* (arXiv:2208.06213). arXiv.

<http://arxiv.org/abs/2208.06213>

Sarsa, S., Denny, P., Hellas, A., & Leinonen, J. (2022). Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models. *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*, 27–43. <https://doi.org/10.1145/3501385.3543957>

Schenker, J. D., & Rumrill, Jr., P. D. (2004). Causal-comparative research designs. *Journal of Vocational Rehabilitation*, 21(3), 117–121. <https://doi.org/10.3233/JVR-2004-00260>

Schulz, P. (1991, August 19). Microsoft Windows gets back to ‘basics.’ *Info World*.

Shrestha, N. (2021). Factor Analysis as a Tool for Survey Analysis. *American Journal of Applied Mathematics and Statistics*, 9(1), 4–11. <https://doi.org/10.12691/ajams-9-1-2>

Shrestha, N., Botta, C., Barik, T., & Parnin, C. (2022). Here we go again: Why is it difficult for developers to learn another programming language? *Communications of the ACM*, 65(3), 91–99. <https://doi.org/10.1145/3511062>

Sobania, D., Briesch, M., & Rothlauf, F. (2022). Choose your programming copilot: A comparison of the program synthesis performance of github copilot and genetic programming. *Proceedings of the Genetic and Evolutionary Computation Conference*, 1019–1027. <https://doi.org/10.1145/3512290.3528700>

Stack Overflow Developer Survey 2023. (2023). 2023 Developer Survey.

<https://survey.stackoverflow.co/2023/>

Stack Overflow Developer Survey 2024. (2024). 2024 Developer Survey.

<https://survey.stackoverflow.co/2024/>

Storey, M.-A., Houck, B., & Zimmermann, T. (2022). How developers and managers define and trade productivity for quality. *Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering*, 26–35.

<https://doi.org/10.1145/3528579.3529177>

Sun, Z., Du, X., Song, F., Wang, S., Ni, M., & Li, L. (2023). Don't Complete It! Preventing Unhelpful Code Completion for Productive and Sustainable Neural Code Completion Systems. *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 324–325. <https://doi.org/10.1109/ICSE-Companion58688.2023.00089>

Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., & Zhang, L. (2020). TreeGen: A Tree-Based Transformer Architecture for Code Generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(05), 8984–8991. <https://doi.org/10.1609/aaai.v34i05.6430>

Tregubov, A., Boehm, B., Rodchenko, N., & Lane, J. A. (2017). Impact of task switching and work interruptions on software development processes. *Proceedings of the 2017 International Conference on Software and System Process*, 134–138.

<https://doi.org/10.1145/3084100.3084116>

Treude, C., Barzilay, O., & Storey, M.-A. (2011). How do programmers ask and answer questions on the web? (NIER track). *Proceedings of the 33rd International Conference on Software Engineering*, 804–807. <https://doi.org/10.1145/1985793.1985907>

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022). Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. *CHI*

- Conference on Human Factors in Computing Systems Extended Abstracts*, 1–7.
<https://doi.org/10.1145/3491101.3519665>
- Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). *Attention Is All You Need* (arXiv:1706.03762). arXiv.
<https://doi.org/10.48550/arXiv.1706.03762>
- Wang, R., Cheng, R., Ford, D., & Zimmermann, T. (2024). Investigating and Designing for Trust in AI-powered Code Generation Tools. *The 2024 ACM Conference on Fairness, Accountability, and Transparency*, 1475–1493. <https://doi.org/10.1145/3630106.3658984>
- Weber, T., Brandmaier, M., Schmidt, A., & Mayer, S. (2024). Significant Productivity Gains through Programming with Large Language Models. *Proceedings of the ACM on Human-Computer Interaction*, 8(EICS), 1–29. <https://doi.org/10.1145/3661145>
- Weisz, J. D., Muller, M., Ross, S. I., Martinez, F., Houde, S., Agarwal, M., Talamadupula, K., & Richards, J. T. (2022). Better Together? An Evaluation of AI-Supported Code Translation. *27th International Conference on Intelligent User Interfaces*, 369–391.
<https://doi.org/10.1145/3490099.3511157>
- Wickham, H., Francois, R., Henry, L., & Vaughn, K. (2023). *dplyr: A Grammar of Data Manipulation* (Version 1.1.4) [Computer software]. <https://dplyr.tidyverse.org>
- Wickham, H. (with Sievert, C.). (2016). *ggplot2: Elegant graphics for data analysis* (Second edition). Springer international publishing.
- Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., Von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., ... Rush, A. (2020). Transformers: State-of-the-Art Natural Language Processing. *Proceedings of the 2020 Conference on Empirical*

Methods in Natural Language Processing: System Demonstrations, 38–45.

<https://doi.org/10.18653/v1/2020.emnlp-demos.6>

Wright, M. N., & Ziegler, A. (2017). **ranger**: A Fast Implementation of Random Forests for High Dimensional Data in C++ and R. *Journal of Statistical Software*, 77(1).

<https://doi.org/10.18637/jss.v077.i01>

Xu, F. F., Vasilescu, B., & Neubig, G. (2022). In-IDE Code Generation from Natural Language: Promise and Challenges. *ACM Transactions on Software Engineering and Methodology*, 31(2), 1–47. <https://doi.org/10.1145/3487569>

Yildirim, N., & Ansal, H. (2011). Foresighting FLOSS (free/libre/open source software) from a developing country perspective: The case of Turkey. *Technovation*, S0166497211001052. <https://doi.org/10.1016/j.technovation.2011.07.004>

Zeileis, A. (2004). Econometric Computing with HC and HAC Covariance Matrix Estimators. *Journal of Statistical Software*, 11(10). <https://doi.org/10.18637/jss.v011.i10>

Zhang, B., Liang, P., Zhou, X., Ahmad, A., & Waseem, M. (2023). *Practices and Challenges of Using GitHub Copilot: An Empirical Study*. 124–129. <https://doi.org/10.18293/SEKE2023-077>

Ziegler, A., Kalliamvakou, E., Li, X. A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., & Aftandilian, E. (2022). Productivity assessment of neural code completion. *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 21–29. <https://doi.org/10.1145/3520312.3534864>

Figures

Figure 1

How this study differs

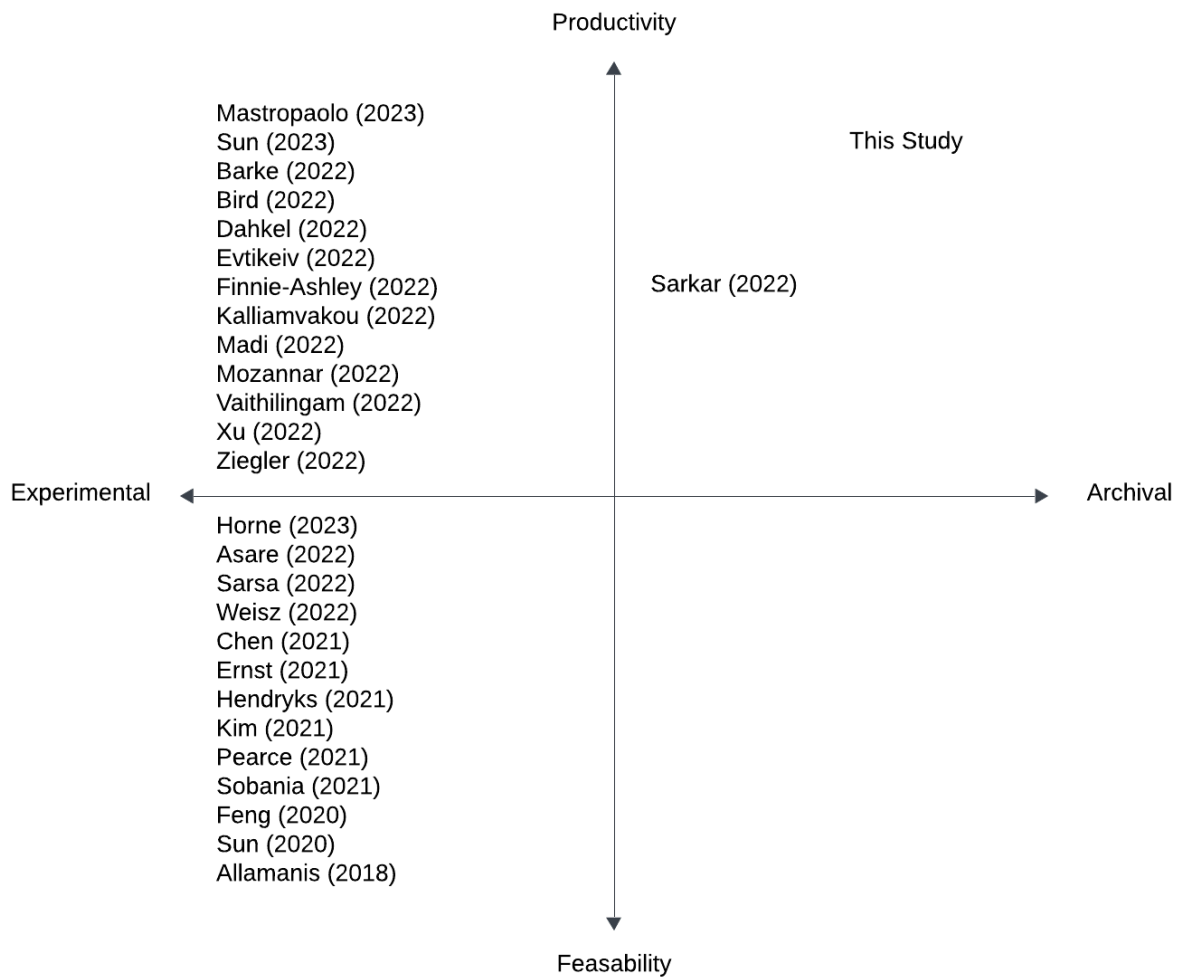


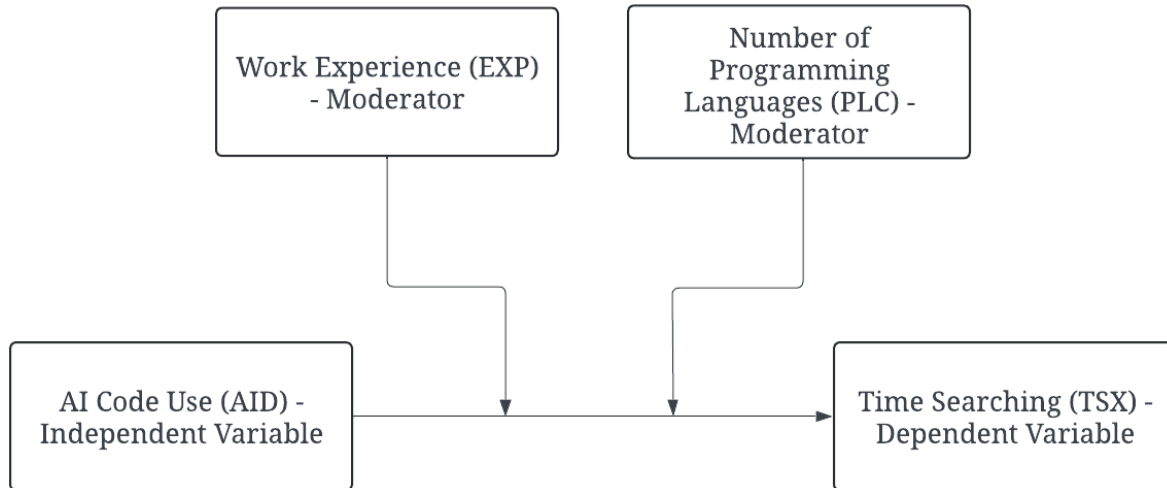
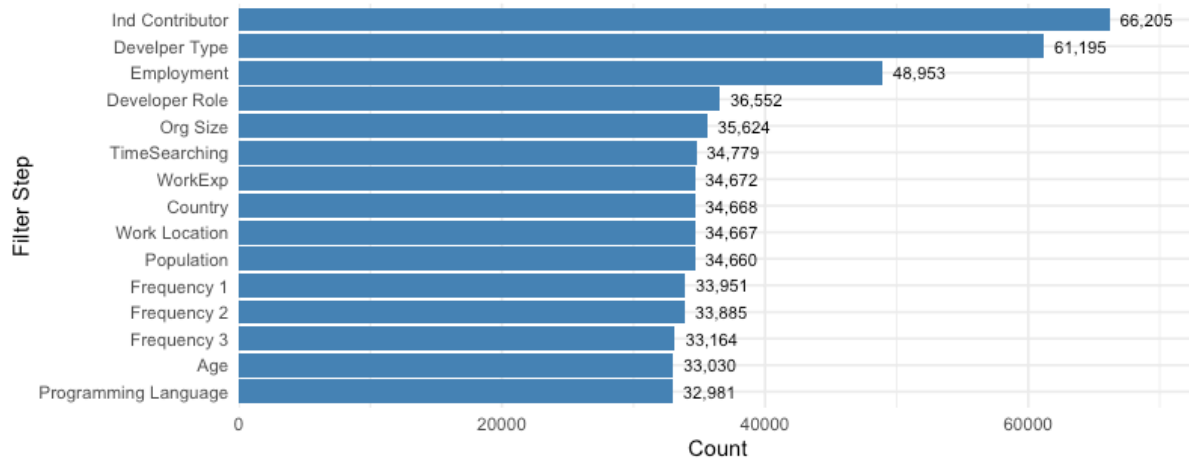
Figure 2*Conceptual Model***Figure 3***Filter Process Results*

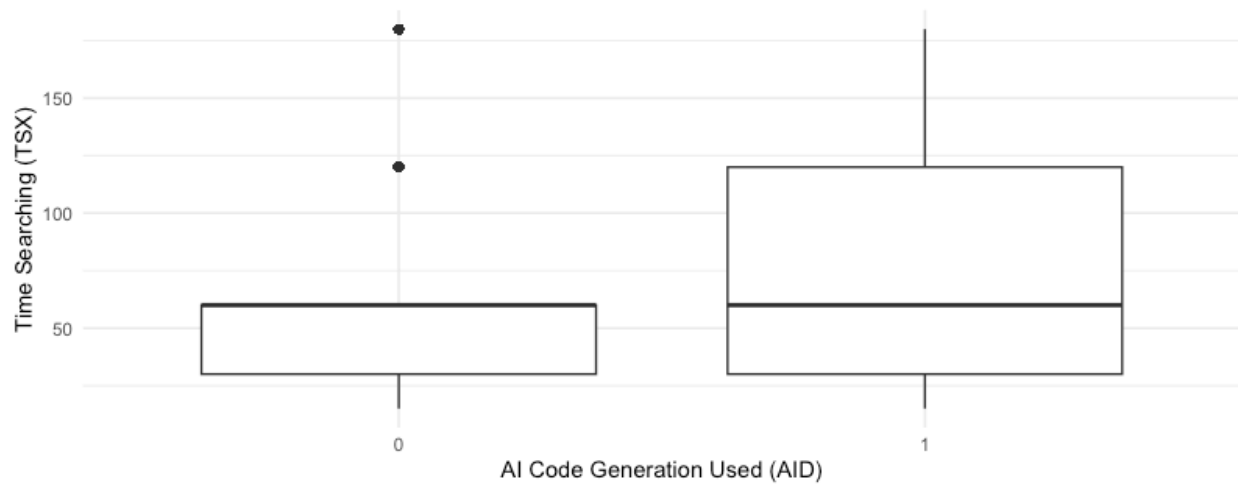
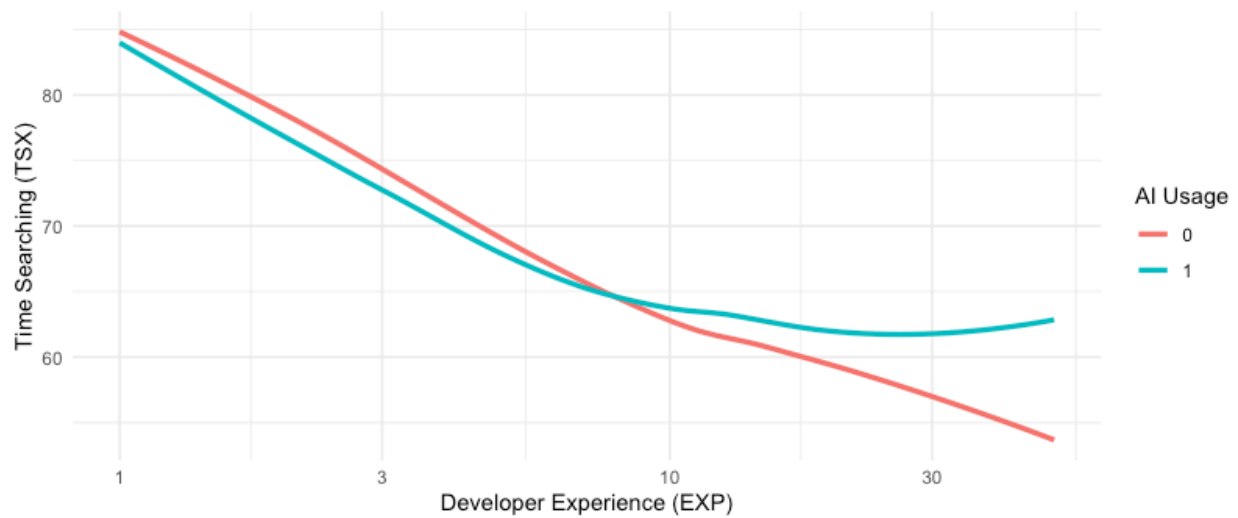
Figure 4*Box Plot for Descriptive Statistics***Figure 5***Impact of Experience and AI Code-Generation on Search Time*

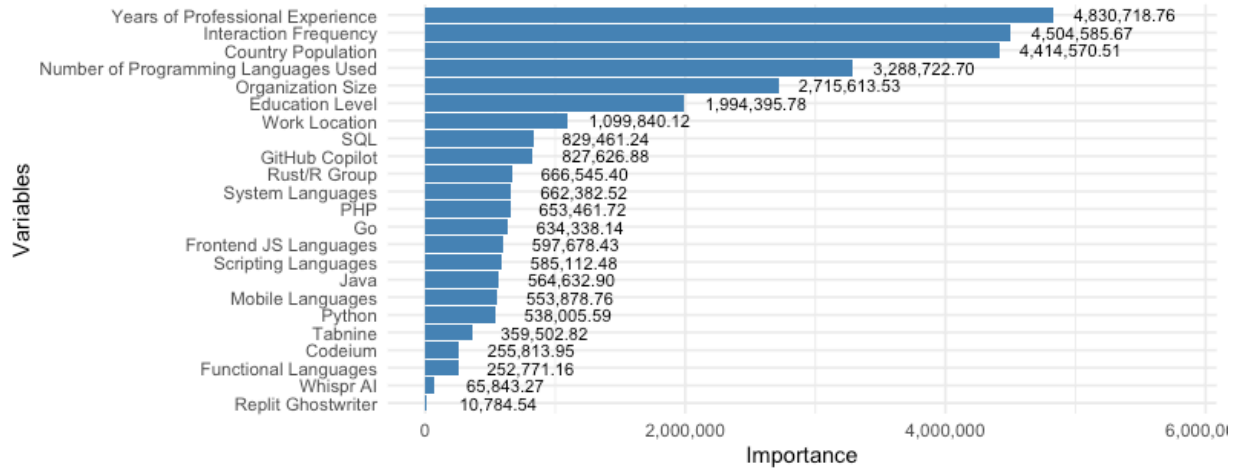
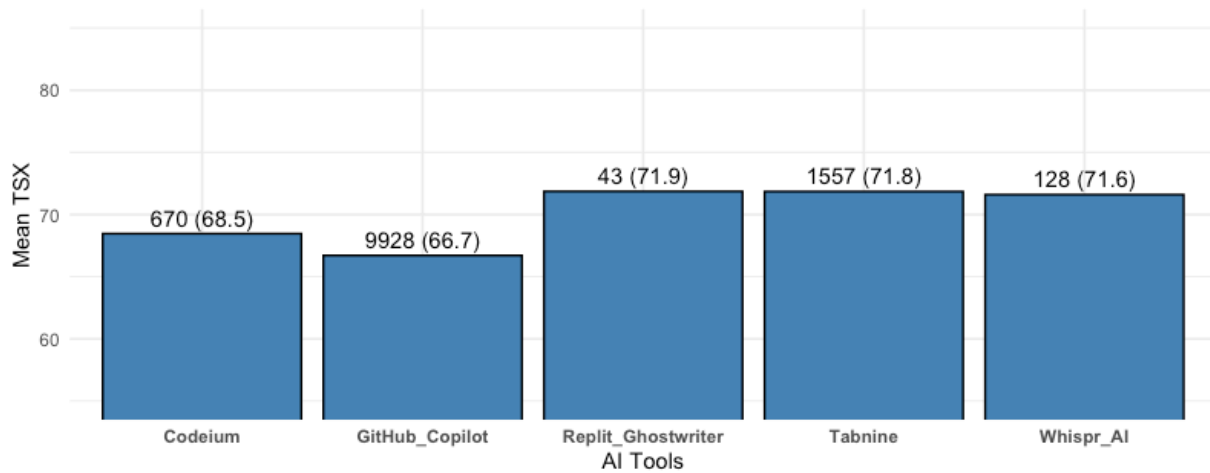
Figure 6*Predictive Variable Importance Plot***Figure 7***Mean Search Time by AI Tool*

Figure 8

Difference in Time Searching by Country

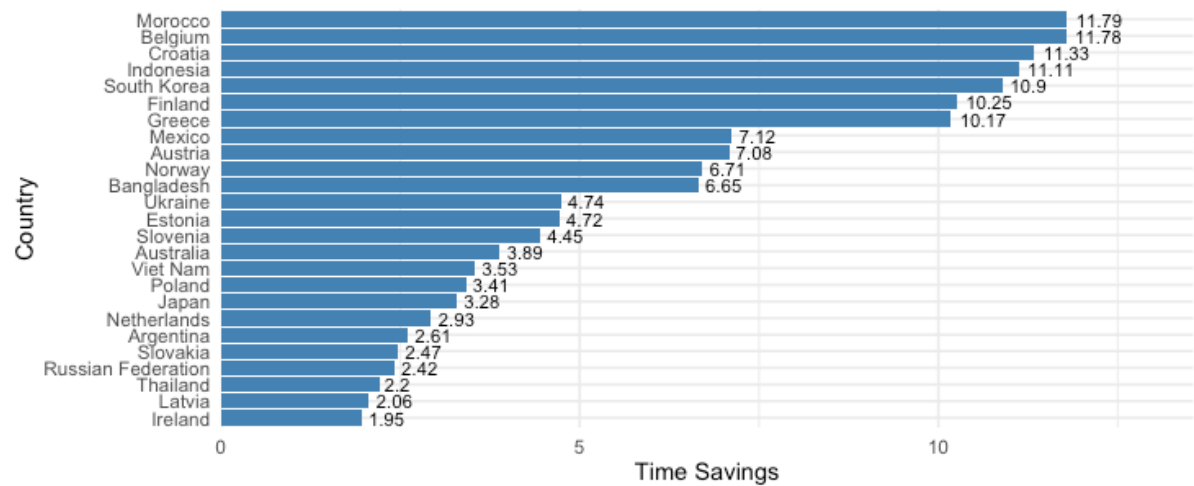


Figure 9

Average Time Savings by Country World Map

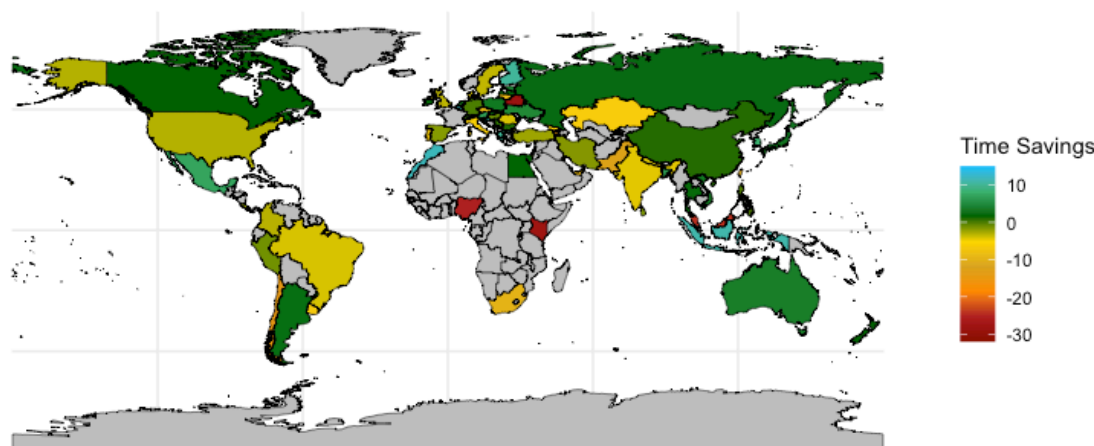
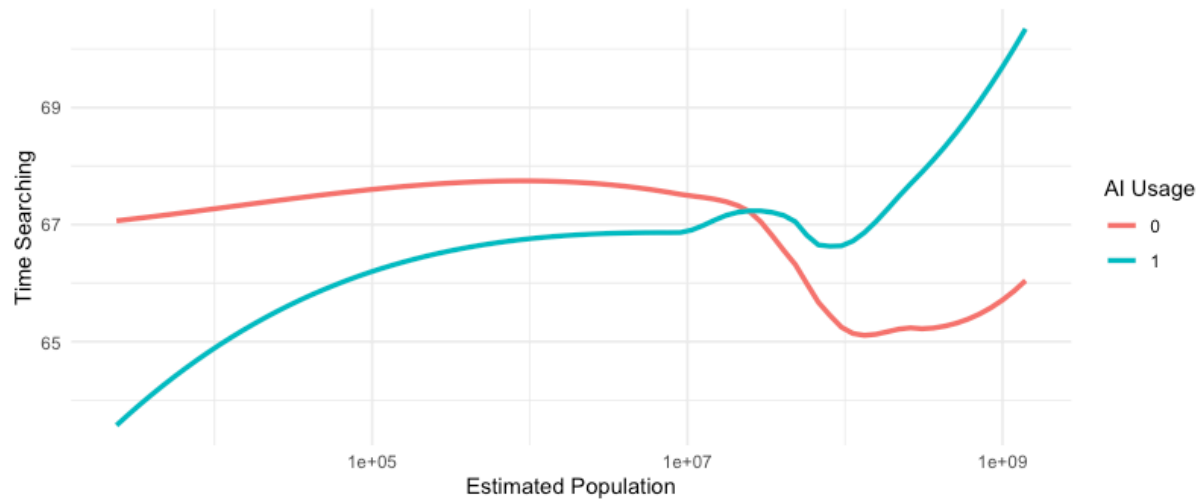


Figure 10

Interaction Effect of AI Code-Generation Usage and Country Population on Search Time



Tables

Table 1

Developer Inclusion Criteria

Question	Question Text	Answer	Recs
QID287	ICorPM	Individual Contributor	66,205
QID2	MainBranch	I am a developer by profession	61,195
QID296	Employment	Employed, Full Time	48,953
QID31	DevType	Back-end, Front-end, Full-stack, Ent Apps	36,552
QID29	OrgSize	More than one person in the company	35,624
QID291	TimeSearching	Must be answered	34,779
QID288	WorkExp	Must be answered	34,672
QID6	Country	Filter out any Nomadic respondents	34,668
QID308	RemoteWork	Must be answered	34,667
NA	Population	No NA values	34,660
QID290	Frequency 1	No NA values	33,951
QID290	Frequency 2	No NA values	33,885
QID290	Frequency 3	No NA values	33,164
QID127	Age	No NA values	33,030
QID233	Language	No NA values	32,981

Table 2

Number of records, Mean, and Standard Deviation with and without AI Code-Generation

AI Code-Generation Used	Yes	No
Mean (M)	67.3	66.1
Count (N)	11,387	21,728

Note. Standard Deviation = 44.8, Min = 15, Max = 180, and Median = 60 for both groups.

Table 3*Demographic Factor Analysis*

Dataset Item	Factor loading					
	1	2	3	4	5	6
Experience	0.984*	0.114			-0.101	
Age	0.837*	0.156				0.229
Economy	0.111	0.550*				
Country		0.512*		0.133		
Work Location	0.114	0.191			-0.171	
Organization Size		0.123		0.267		
AI Code-Generation Usage			0.987*			-0.142
Time Searching					0.271	
Interruption Frequency					0.522	0.286
Education Level						
Number of Programming Languages					0.144	-0.237
Population				0.570*		

Note. N = 32,981.

* Indicates factor loadings above 0.30.

Table 4*Programming Language Factor Analysis*

Language	Factor loading									
	1	2	3	4	5	6	7	8	9	10
Python										0.405*
JavaScript	0.986*									
TypeScript	0.360*			0.142		-0.106				
Java	0.789*									
C	0.179			0.861*	0.293	0.111			0.348*	
C#					0.935*	0.102				-0.182
C++	-0.130					0.269			0.434*	0.257
SQL	0.158			0.169	0.202	0.154			-0.220	0.140
Ruby			0.983*							
PHP	0.182			0.122		0.188				
Go										0.283
Rust		0.947*				0.296				
R		0.709*	0.524*			0.290				
Elixir							0.636*			
Erlang							0.630*			
Swift								0.649*		
Kotlin								0.220		0.188
PowerShell				0.109	0.376*	0.157			-0.117	0.111
Assembly						0.217			0.128	0.270

Note. N = 32,981.

* Indicates factor loadings above 0.30.

Table 5*Correlations Among AI Code-Generation Usage, Answer Search Time, Moderators, and Other Fields with Descriptive Statistics*

	M	SD	1	2	3	4	5	6	7	8	9	10	11	12
1. Time Searching	66.54	44.81	—	—	—	—	—	—	—	—	—	—	—	—
2. AI Code-Gen Usage	0.35	0.48	0.012 *	—	—	—	—	—	—	—	—	—	—	—
3. Number of Languages	5.21	2.71	0.026 ***	0.101 ***	—	—	—	—	—	—	—	—	—	—
4. Experience	10.76	8.51	-0.124 ***	-0.054 ***	0.005	—	—	—	—	—	—	—	—	—
5. Country	—	—	-0.009 .	0.002	-0.023 ***	0.083 ***	—	—	—	—	—	—	—	—
6. Age	38.18	9.24	-0.101 ***	-0.068 ***	-0.056 ***	0.836 ***	0.084 ***	—	—	—	—	—	—	—
7. Population	2.1E+08	3.5E+08	0.004	0	-0.030 ***	-0.113 ***	0.027 ***	-0.131 ***	—	—	—	—	—	—
8. Education	15.92	2.14	0.005	-0.058 ***	-0.051 ***	0.031 ***	0.023 ***	0.103 ***	0.035 ***	—	—	—	—	—
9. Organization Size	3,881.66	6,667.28	0.037 ***	-0.037 ***	-0.005	0.034 ***	0.079 ***	0.028 ***	0.139 ***	0.072 ***	—	—	—	—
10. Work Location	2.25	0.71	0.045 ***	0.031 ***	-0.043 ***	0.148 ***	0.115 ***	0.147 ***	-0.063 ***	-0.005	0.035 ***	—	—	—
11. Interaction Frequency	2.87	2.19	0.138 ***	0.034 ***	0.096 ***	0.023 ***	0.043 ***	0	-0.004	-0.014 *	0.061 ***	-0.075 ***	—	—
12. Economy	94.29	21.95	-0.020 ***	0.029 ***	0.067 ***	0.168 ***	0.271 ***	0.175 ***	-0.125 ***	-0.008	0.063 ***	0.085 ***	0.085 ***	—

Note. . $p < .1$, * $p < .05$, ** $p < .01$, *** $p < .001$

N = 32,981 for all fields. Country is categorical, therefore Mean and Standard Deviation are reported

Table 6|*Correlations Among AI Code-Generation Usage, Answer Search Time, Moderators, and Programming Languages*

	M	SD	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1. Time Searching	66.54	44.81	—	—	—	—	—	—	—	—	—	—	—	—	—	—
2. AI Code-Gen Usage	0.35	0.48	0.012 *	—	—	—	—	—	—	—	—	—	—	—	—	—
3. Frontend Langs	0.779	0.415	0	0.115 ***	—	—	—	—	—	—	—	—	—	—	—	—
4. Systems Langs	0.765	0.424	0.004	0.017 **	0.272 ***	—	—	—	—	—	—	—	—	—	—	—
5. Rust/R Group	0.179	.0383	-0.005	0.080 ***	0.039 ***	-0.013 *	—	—	—	—	—	—	—	—	—	—
6. Functional Langs	0.030	0.169	-0.019 **	0.024 ***	-0.025 ***	-0.045 ***	0.117 ***	—	—	—	—	—	—	—	—	—
7. Mobile Langs	0.106	0.308	0.007	0.035 ***	0.018 **	-0.046 ***	0.061 ***	0.002	—	—	—	—	—	—	—	—
8. Scripting Langs	0.167	0.373	0.013 *	0.017 **	0.070 ***	0.199 ***	0.003	-0.032 ***	0	—	—	—	—	—	—	—
9. Python	0.394	0.489	0.051 ***	0.058 ***	-0.007	0	0.121 *	-0.012 *	0.055 ***	0.052 ***	—	—	—	—	—	—
10. Java	0.801	0.399	0.014 **	0.055 ***	0.644 ***	0.133 ***	0	-0.043 ***	0.076 ***	0.045 ***	-0.004	—	—	—	—	—
11. PHP	0.171	0.377	-0.009	0.021 ***	0.161 ***	0.108 ***	-0.021 ***	-0.027 ***	0.004	0.026 ***	0.008	0.150 ***	—	—	—	—
12. C#	0.330	0.470	-0.001	-0.020 ***	0.021 ***	0.388 ***	-0.090 ***	-0.056 ***	-0.040 ***	0.340 ***	-0.089 ***	-0.074 ***	-0.062 ***	—	—	—
13. SQL	0.565	0.496	0.002	0.015 **	0.141 ***	0.168 ***	0.029 ***	-0.008	0.012 *	0.177 ***	0.081 ***	0.154 ***	0.165 ***	0.177 ***	—	—
14. Ruby	0.072	0.258	-0.005	0.061 ***	0.049 ***	-0.024 ***	0.595 ***	0.133 ***	0.028 ***	-0.034 ***	0.012 **	0.036 ***	-0.008	-0.098 ***	0.040 ***	—
15. Go	0.145	0.352	-0.001	0.076 ***	-0.008	-0.070 ***	0.163 ***	0.043 ***	0.074 ***	-0.018 **	0.130 ***	-0.024 ***	0.010	0.010	0.013 *	0.070 ***

Note. * $p < .05$, ** $p < .01$, *** $p < .001$. $N = 32,981$ for all fields, therefore Mean and Standard Deviation are reported.

Table 7

Correlations Among AI Code-Generation Usage, Answer Search Time, and AI Code-Generation Tools

	M	SD	1	2	3	4	5	6
1. Time Searching	66.54	44.81	—	—	—	—	—	—
2. AI Code-Gen Usage	0.35	0.48	0.012 *	—	—	—	—	—
3. Codeium	0.02	0.141	0.006	0.199 ***	—	—	—	—
4. GitHub Copilot	0.301	0.459	0.003	0.905 ***	0.060 ***	—	—	—
5. Replit Ghostwriter	0.001	0.036	0.004	0.050 ***	0.036 ***	0.033 ***	—	—
6. Tabnine	0.047	0.212	0.026 ***	0.307 ***	0.081 ***	0.114 ***	0.043 ***	—
7. Whisper AI	0.004	0.062	0.007	0.086 ***	0.043 ***	0.057 ***	0.052 ***	0.044 ***

Note. * $p < .05$, ** $p < .01$, *** $p < .001$. $N = 32,981$ for all fields, therefore Mean and Standard Deviation are reported.

Table 8*Time Searching Linear Regression Models with No Interactions*

Predictor	Model 1	Model 2	Model 3	Model 4
AI Code-Gen Usage	0.003 (0.006)	0.002 (0.005)	0.018 (0.016)	0.019 (0.016)
Experience	-0.125*** (0.005)	-0.128*** (0.005)	-0.127*** (0.005)	-0.128*** (0.006)
# of Programming Langs	0.026*** (0.006)	0.013* (0.006)	0.012* (0.006)	0.02 (0.011)
Country	< 0.001 (0.005)	-0.006 (0.005)	-0.005 (0.005)	-0.006 (0.005)
Organization Size	—	0.036*** (0.006)	0.036*** (0.006)	0.032*** (0.006)
Work Location	—	-0.017** (0.006)	-0.016** (0.006)	-0.013* (0.006)
Education Level	—	0.009 (0.005)	0.009 (0.005)	0.006 (0.005)
Frequency of Interaction	—	0.136*** (0.006)	0.137*** (0.006)	0.136*** (0.006)
Country Pop Estimate	—	-0.016** (0.006)	-0.017** (0.006)	-0.018** (0.006)
Codeium	—	—	< 0.001 (0.006)	0.001 (0.006)
GitHub Copilot	—	—	-0.025 (0.015)	-0.023 (0.015)
Replit Ghostwriter	—	—	0.002 (0.006)	0.002 (0.006)
Tabnine	—	—	0.018** (0.006)	0.018** (0.006)
Whispr AI	—	—	0.005 (0.006)	0.005 (0.006)
Frontend JS Langs	—	—	—	-0.023** (0.008)
Systems Langs	—	—	—	-0.003 (0.007)
Functional Langs	—	—	—	-0.025*** (0.007)
Mobile Langs	—	—	—	-0.014* (0.005)
Scripting Langs	—	—	—	0.006 (0.006)
Python	—	—	—	0.026*** (0.007)
Java	—	—	—	0.013 (0.008)
PHP	—	—	—	-0.012* (0.006)
C#	—	—	—	0.005 (.007)
SQL	—	—	—	-0.002 (0.006)
Ruby	—	—	—	0.014* (0.007)
Go	—	—	—	-0.013* (0.006)

Note. Standardized beta coefficients (β) and robust standard errors (SE) are reported in parentheses.

* $p < .05$, ** $p < .01$, *** $p < .001$

Table 9*Time Searching Linear Regression Models with Interactions*

Predictor	Model 5	Model 6	Model 7	Model 8
AI Code-Gen Usage (AID)	0.019 (0.047)	0.027 (0.050)	0.055 (0.052)	0.007 (0.077)
Experience (EXP)	-0.136*** (0.006)	-0.136*** (0.006)	-0.136*** (0.006)	-0.136*** (0.007)
Num of Pgm Langs (PLC)	0.012 (0.007)	0.012 (0.007)	0.012 (0.007)	0.019 (0.014)
Country (CTY)	-0.016* (0.007)	-0.016* (0.007)	-0.016* (0.007)	-0.017 (0.007)
Organization Size (ORG)	0.034*** (0.007)	0.034*** (0.007)	0.034*** (0.007)	0.029*** (0.007)
Work Location (LOC)	-0.012 (0.007)	-0.012 (0.007)	-0.012 (0.007)	-0.010 (0.007)
Education Level (EDL)	0.013* (0.007)	0.013* (0.007)	0.013* (0.007)	0.009 (0.007)
Freq of Interaction (FRQ)	0.138*** (0.008)	0.138*** (0.008)	0.138*** (0.008)	0.137*** (0.008)
Country Pop. (POP)	-0.027 (0.007)	-0.027 (0.007)	-0.027 (0.007)	-0.029*** (0.007)
AID * EXP	0.022*(0.009)	0.024*(0.009)	0.022*(0.010)	0.022* (0.010)
AID * PLC	0.002 (0.014)	0.001 (0.014)	0.016 (0.024)	0.006 (0.028)
AID * CTY	0.031** (0.012)	0.033** (0.012)	0.032** (0.012)	0.031** (0.012)
AID * ORG	0.005 (0.008)	0.005 (0.008)	0.003 (0.008)	0.006 (0.008)
AID * LOC	-0.027 (0.020)	-0.023 (0.020)	-0.016 (0.020)	-0.019 (0.020)
AID * EDL	-0.044 (0.040)	-0.043 (0.040)	-0.052 (0.040)	-0.035 (0.041)
AID * FRQ	-0.004 (0.011)	-0.002 (0.011)	-0.002 (0.011)	-0.001 (0.011)
AID * POP	0.022** (0.008)	0.02** (0.008)	0.019** (0.008)	0.020* (0.008)
AID * Codeium	—	0 (0.006)	0.001 (0.006)	0.048* (0.022)
AID * GitHub Copilot	—	-0.023 (0.015)	-0.022 (0.010)	0.010 (0.057)
AID * Replit Ghostwriter	—	0.001 (0.006)	0.001 (0.006)	-0.016 (0.018)
AID * Tabnine	—	0.018** (0.006)	0.019** (0.006)	0.034 (0.025)
AID * Whispr AI	—	0.005 (0.006)	0.004 (0.006)	0.003 (0.023)
AID * Frontend JS Langs	—	—	0.016 (0.054)	-0.084 (0.053)
AID * Systems Langs	—	—	0.012 (0.209)	0.051 (0.041)
AID * Rust/R Group	—	—	0.008 (0.002)	-0.067** (0.025)
AID * Functional Langs	—	—	0.005 (0.053)	-0.020 (0.021)
AID * Mobile Langs	—	—	0.006 (0.270)	0.000 (0.020)
AID * Scripting Langs	—	—	0.007 (0.062)	0.012 (0.024)
AID * Python	—	—	0.008 (0.032)	0.004 (0.025)
AID * Java	—	—	0.015 (0.989)	0.059 (0.051)
AID * PHP	—	—	0.007 (0.062)	0.009 (0.022)
AID * C#	—	—	0.007 (0.764)	0.015 (0.027)
AID * SQL	—	—	0.009 (0.989)	-0.040 (0.029)
AID * Ruby	—	—	0.007 (0.044)	0.061* (0.027)
AID * Go	—	—	0.006 (0.028)	0.035 (0.022)

Predictor	Model 5	Model 6	Model 7	Model 8
AID * Codeium * Frontend	—	—	—	0.027 (0.023)
AID * Codeium * Systems	—	—	—	-0.055*** (0.016)
AID * Codeium * Rust/R	—	—	—	-0.006 (0.007)
AID * Codeium * Func	—	—	—	-0.008 (0.006)
AID * Codeium * Mobile	—	—	—	-0.005 (0.006)
AID * Codeium * Scripting	—	—	—	0.000 (0.007)
AID * Codeium * Python	—	—	—	-0.010 (0.008)
AID * Codeium * Java	—	—	—	-0.026 (0.022)
AID * Codeium * PHP	—	—	—	0.004 (0.008)
AID * Codeium * C#	—	—	—	-0.011 (0.008)
AID * Codeium * SQL	—	—	—	0.021* (0.010)
AID * Codeium * Ruby	—	—	—	0.007 (0.008)
AID * Codeium * Go	—	—	—	-0.006 (0.007)
AID * Copilot * Frontend	—	—	—	0.079 (0.050)
AID * Copilot * Systems	—	—	—	-0.061 (0.038)
AID * Copilot * Rust/R	—	—	—	0.061** (0.023)
AID * Copilot * Func	—	—	—	0.021 (0.019)
AID * Copilot * Mobile	—	—	—	-0.010 (0.018)
AID * Copilot * Scripting	—	—	—	0.004 (0.022)
AID * Copilot * Python	—	—	—	-0.003 (0.023)
AID * Copilot * Java	—	—	—	-0.078 (0.049)
AID * Copilot * PHP	—	—	—	0.009 (0.019)
AID * Copilot * C#	—	—	—	-0.015 (0.025)
AID * Copilot * SQL	—	—	—	0.040 (0.027)
AID * Copilot * Ruby	—	—	—	-0.059* (0.025)
AID * Copilot * Go	—	—	—	-0.039. (0.020)
AID * Ghostwriter * Frontend	—	—	—	0.024 (0.016)
AID * Ghostwriter * Systems	—	—	—	0.005 (0.013)
AID * Ghostwriter * Rust/R	—	—	—	0.006 (0.009)
AID * Ghostwriter * Func	—	—	—	0.004 (0.005)
AID * Ghostwriter * Mobile	—	—	—	0.009 (0.006)
AID * Ghostwriter * Script	—	—	—	0.006 (0.010)
AID * Ghostwriter * Python	—	—	—	-0.014 (0.011)
AID * Ghostwriter * Java	—	—	—	0.000 (0.017)
AID * Ghostwriter * PHP	—	—	—	0.007 (0.009)
AID * Ghostwriter * C#	—	—	—	0.000 (0.010)

Predictor	Model 5	Model 6	Model 7	Model 8
AID * Ghostwriter * SQL	—	—	—	-0.009 (0.016)
AID * Ghostwriter * Ruby	—	—	—	-0.13** (0.005)
AID * Ghostwriter * Go	—	—	—	-0.004 (0.008)
AID * Tabnine * Frontend	—	—	—	0.010 (0.025)
AID * Tabnine * Systems	—	—	—	-0.012 (0.015)
AID * Tabnine * Rust/R	—	—	—	0.009 (0.009)
AID * Tabnine * Func	—	—	—	-0.003 (0.006)
AID * Tabnine * Mobile	—	—	—	0.012. (0.007)
AID * Tabnine * Scripting	—	—	—	0.002 (0.007)
AID * Tabnine * Python	—	—	—	-0.002 (0.009)
AID * Tabnine * Java	—	—	—	-0.024 (0.024)
AID * Tabnine * PHP	—	—	—	-0.002 (0.008)
AID * Tabnine * C#	—	—	—	-0.001 (0.008)
AID * Tabnine * SQL	—	—	—	0.013 (0.011)
AID * Tabnine * Ruby	—	—	—	-0.003 (0.008)
AID * Tabnine * Go	—	—	—	-0.015 (0.008)
AID * Whispr * Frontend	—	—	—	-0.020 (0.027)
AID * Whispr * Systems	—	—	—	0.006 (0.017)
AID * Whispr * Rust/R	—	—	—	-0.009 (0.007)
AID * Whispr * Func	—	—	—	0.004 (0.006)
AID * Whispr * Mobile	—	—	—	-0.003 (0.006)
AID * Whispr * Systems	—	—	—	-0.003 (0.006)
AID * Whispr * Python	—	—	—	-0.003 (0.009)
AID * Whispr * Java	—	—	—	0.030 (0.028)
AID * Whispr * PHP	—	—	—	0.009 (0.008)
AID * Whispr * C#	—	—	—	0.007 (0.007)
AID * Whispr * SQL	—	—	—	-0.016 (0.010)
AID * Whispr * Ruby	—	—	—	0.009 (0.007)
AID * Whispr * Go	—	—	—	-0.007 (0.005)

Note. Standardized beta coefficients (β) and robust standard errors (SE) are reported in parentheses.

* $p < .05$, ** $p < .01$, *** $p < .001$

Table 10*Time Searching by AI Code Generation Tool*

Tool	Count	Mean Time Searching
GitHub Copilot	9,928	66.710
Codeium	670	68.500
Whispr AI	128	71.602
Replit Ghostwriter	43	71.860
Tabnine	1,557	71.865

Table 11*Time Savings Using AI Code Generation for the Top 25 Countries*

Country	No AI Code Generation		AI Code Generation		Savings
	N	Mean	N	Mean	
Morocco	28	64.3	16	52.5	11.8
Belgium	192	73.1	89	61.3	11.8
Croatia	78	61.2	28	49.8	11.3
Indonesia	86	79.5	57	68.4	11.1
South Korea	41	86.3	34	75.4	10.9
Finland	164	73.4	94	63.2	10.3
Greece	154	70.7	55	60.5	10.2
Mexico	141	73.1	78	66	7.12
Austria	231	71.3	96	64.2	7.08
Norway	179	72.9	138	66.2	6.71
Bangladesh	95	73.3	59	66.6	6.65
Ukraine	408	69.9	220	65.1	4.74
Estonia	43	78.1	38	73.4	4.72
Slovenia	69	67.2	22	62.7	4.45
Australia	1335	67.2	744	63.3	3.89
Viet Nam	63	81.4	57	77.9	3.53
Poland	453	76.8	217	73.4	3.41
Japan	87	66.4	63	63.1	3.28
Netherlands	493	65.1	283	62.1	2.93
Argentina	89	69.6	60	67.0	2.61
Slovakia	49	58.5	30	56	2.47
Russian Federation	318	67	69	64.6	2.42
Thailand	24	60.6	38	58.4	2.20
Latvia	38	80.5	13	78.5	2.06
Ireland	117	64.6	62	62.7	1.95

Note. Minimum of 30 respondents.

Appendix A

OSF Project Home

https://osf.io/dxg4r/?view_only=a25c6791d4f047ebb7c08054f42dd1d7