# THE IMPACT OF AI-POWERED DEVELOPMENT AGENTS ON SOFTWARE DEVELOPER PRODUCTIVITY

**A THESIS SUBMITTED TO**
**THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES OF**
**ANKARA UNIVERSITY**

**by**

**Mohammad SHABIB**

**IN PARTIAL FULFILMENT OF THE REQUIREMENTS**
**FOR THE DEGREE OF**
**MASTER OF SCIENCE IN**
**ARTIFICIAL INTELLIGENCE TECHNOLOGIES**

**ANKARA**
**2026**

## ABSTRACT

Master's Thesis

THE IMPACT OF AI-POWERED DEVELOPMENT AGENTS ON SOFTWARE DEVELOPER PRODUCTIVITY

Mohammad SHABIB

Ankara University
Graduate School of Natural and Applied Science
Department of Artificial Intelligence Technologies

Supervisor: Prof. Dr. Asım Egemen YILMAZ

AI-powered development agents are now widely adopted, and this thesis asks what they actually do to software developer productivity, through a mixed-methods case study at a mid-size software company. A survey of 54 AI tool users across backend, frontend, and quality assurance roles is supplemented by project management metrics from five anonymized development teams. Perceived productivity gains are consistently positive and moderate, with a statistically significant positive correlation between usage frequency and perceived productivity. Effectiveness depends heavily on the task: repetitive work and unit test generation rate highest, while team collaboration and architectural guidance return little. No statistically significant differences appear across professional roles or experience levels, though descriptive trends point to slightly larger gains for junior developers and quality assurance engineers. Thematic analysis of open-ended responses identifies prompt engineering difficulty, limited context awareness, and hallucinations as the primary challenges. Supplementary project management data shows cross-team throughput increases and cycle time reductions after adoption, consistent in direction with what respondents reported. Together these findings support a division of labor model of human-AI collaboration, in which AI handles automatable pattern-based work while humans retain the advantage in contextual reasoning, creative problem-solving, and collaboration. For organizations adopting these tools, the evidence supports expecting moderate rather than revolutionary gains.



**July 2026, 89 pages**

**Keywords:** AI-Powered Development Agents, GitHub Copilot, Software Developer Productivity, Code Quality, Developer Satisfaction

# ÖZET

Yüksek Lisans Tezi

YAPAY ZEKA DESTEKLİ GELİŞTİRME AJANLARININ YAZILIM GELİŞTİRİCİ VERİMLİLİĞİ ÜZERİNDEKİ ETKİSİ

Mohammad SHABIB

Ankara Üniversitesi
Fen Bilimleri Enstitüsü
Yapay Zeka Teknolojileri Anabilim Dalı

Danışman: Prof. Dr. Asım Egemen YILMAZ

Yapay zeka destekli geliştirme ajanları bugün yaygın biçimde benimsenmiştir; bu tez, bu araçların yazılım geliştirici verimliliği üzerinde gerçekte ne yaptığını orta ölçekli bir yazılım şirketinde yürütülen karma yöntemli bir vaka çalışmasıyla incelemektedir. Backend, frontend ve kalite güvence rollerinde çalışan 54 yapay zeka aracı kullanıcısına uygulanan anket, beş anonimleştirilmiş geliştirme ekibinden elde edilen proje yönetim metrikleriyle desteklenmiştir. Algılanan verimlilik kazanımları tutarlı biçimde olumlu ve ılımlıdır; kullanım sıklığı ile algılanan verimlilik arasında istatistiksel olarak anlamlı bir pozitif korelasyon bulunmuştur. Etkinlik büyük ölçüde göreve bağlıdır: tekrarlayan işler ve birim test üretimi en yüksek puanı alırken, ekip işbirliği ve mimari rehberlik az değer sunmaktadır. Mesleki roller veya deneyim seviyeleri arasında istatistiksel olarak anlamlı bir fark ortaya çıkmamıştır; ancak betimsel eğilimler, alt düzey geliştiriciler ve kalite güvence mühendisleri için biraz daha yüksek kazanımlara işaret etmektedir. Açık uçlu yanıtların tematik analizi, başlıca zorluklar olarak prompt mühendisliği güçlüğünü, sınırlı bağlam farkındalığını ve halüsinasyonları belirlemiştir. Ek proje yönetimi verileri, benimseme sonrasında ekipler arası iş çıktısında artış ve döngü süresinde azalma göstermekte olup bu yön katılımcıların bildirdikleriyle tutarlıdır. Bu bulgular birlikte, yapay zekanın otomatikleştirilebilir kalıp tabanlı işleri üstlendiği, insanların ise bağlamsal akıl yürütme, yaratıcı problem çözme ve işbirliğinde üstünlüğünü koruduğu bir işbölümü modelini desteklemektedir. Bu araçları benimseyen kuruluşlar için bulgular, devrimsel değil ılımlı kazanımlar beklenmesini desteklemektedir.

**Temmuz 2026, 89 sayfa**

**Anahtar Kelimeler:** Yapay Zeka Destekli Geliştirme Ajanları, GitHub Copilot, Yazılım Geliştirici Verimliliği, Kod Kalitesi, Geliştirici Memnuniyeti

## FOREWORD AND ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Prof. Dr. Asım Egemen YILMAZ, for his guidance, support, and valuable insights throughout this thesis study.

I owe my deepest thanks to my family. To my mother, Pharm. Eman MOHSIN, who gave me everything before I knew how to ask for it, for sacrifices I noticed and sacrifices I never will. Whatever is good in this work began with her; this page is too small for the rest. To my father, Eng. Azzam SHABIB, for his calm guidance, and for believing in the destination even when I was lost on the road. To my siblings, all of them engineers, for setting the bar high and then helping me climb over it. And to my parents' genes, which did most of the heavy lifting.

To my friends, who were there at my lowest and celebrated at my highest, thank you for making the distance between those two points bearable. If I named you all, this thesis would need a second volume.

I am grateful to the faculty and staff of the Department of Artificial Intelligence Technologies at Ankara University for providing an enriching academic environment, and to all the developers who participated in this research and generously contributed their time and experiences.

I would also like to acknowledge the assistance of Claude for its utility as an AI assistant in brainstorming and refining the text during the drafting process of this thesis.

Finally, I would like to thank myself. For the perseverance, the late nights, and the sheer dedication it took to see this research through to completion. This journey was challenging, but the hard work was entirely worth it.

Mohammad SHABIB
Ankara, July 2026

## TABLE OF CONTENTS

## SYMBOLS AND ABBREVIATIONS

| | |
|---|---|
| AI | Artificial Intelligence |
| ANOVA | Analysis of Variance |
| API | Application Programming Interface |
| BERT | Bidirectional Encoder Representations from Transformers |
| CASE | Computer-Aided Software Engineering |
| CI/CD | Continuous Integration/Continuous Deployment |
| CRUD | Create, Read, Update, Delete |
| CSV | Comma-Separated Values |
| DevEx | Developer Experience |
| DevOps | Development and Operations |
| DORA | DevOps Research and Assessment |
| FAANG | Facebook, Amazon, Apple, Netflix, Google |
| GPT | Generative Pre-trained Transformer |
| HTTP | Hypertext Transfer Protocol |
| IDE | Integrated Development Environment |
| IRB | Institutional Review Board |
| KPI | Key Performance Indicator |
| LLM | Large Language Model |
| LOC | Lines of Code |
| METR | Model Evaluation and Threat Research |
| NLP | Natural Language Processing |
| PR | Pull Request |
| QA | Quality Assurance |
| RAG | Retrieval-Augmented Generation |
| REST | Representational State Transfer |
| ROI | Return on Investment |
| RQ | Research Question |
| SD | Standard Deviation |
| SDLC | Software Development Life Cycle |
| SP | Story Points |
| SPACE | Satisfaction, Performance, Activity, Communication and Collaboration, Efficiency and Flow |
| % | Percentage |

## LIST OF FIGURES

## LIST OF TABLES

# 1. INTRODUCTION

Writing software changed when large language models learned to write code. OpenAI released GPT-3 in 2020, GitHub Copilot followed in 2021, ChatGPT in 2022, and within roughly three years AI-powered development agents went from research demonstrations to tools that millions of developers open every morning. The models behind them are trained on code repositories (Chen et al. 2021; Rozière et al. 2024), and they now cover generation, debugging assistance, documentation, and translation from a plain-language description into working code (Huynh and Lin 2025).

How fast this happened is easiest to see in the survey data. Stack Overflow's 2025 Developer Survey, a self-selected practitioner poll drawing over 49,000 respondents across 177 countries, puts adoption at roughly 84% of developers currently using or planning to use AI coding assistants, with 51% using them daily (Stack Overflow 2025). In Rogers's (2003) terms that is well past the early adopters and into the early majority, reached in under three years. Commercial figures track the same curve: GitHub Copilot passed 4.7 million paid subscribers by early 2026 (Microsoft 2026) and was running inside 90% of Fortune 100 companies as of mid-2025 (Microsoft 2025). Sonar (2026) puts the share of committed code that is AI-generated or AI-assisted at 42%.

Claims about productivity arrived alongside the tools. Vendors and early studies reported time savings of up to 55% on specific development tasks (Peng et al. 2023; Ziegler et al. 2024) and a 26% rise in completed tasks across large-scale field experiments (Cui et al. 2025). The evidence supporting those numbers is thinner than the numbers suggest, especially for mid-size organizations outside the technology sector's elite tier. Most of it comes from controlled laboratory tasks (Peng et al. 2023; Vaithilingam et al. 2022) or from large technology companies (Ziegler et al. 2024; Cui et al. 2025). The ordinary enterprise case is left open: developers maintaining legacy systems under real deadlines. Where researchers have looked outside those settings, the picture is less uniform. Becker et al. (2025) measured a 19% *slowdown* among experienced open-source developers. Stray et al. (2026) recorded perceived improvements that commit-level metrics failed to confirm.

This thesis argues that AI-powered development agents deliver moderate, task-dependent productivity benefits in mid-size software organizations, concentrated in automatable pattern-based work and showing no statistically significant variation across developer

roles or experience levels. The argument rests on a mixed-methods case study: a survey of 54 AI tool users, supplemented by JIRA project management metrics from five anonymized development teams. The organization in question resembles most software companies considerably more closely than the elite firms that dominate the existing literature (Ziegler et al. 2024; Cui et al. 2025).

## 1.1 Problem Statement

Wide adoption and confident vendor claims have not been matched by evidence covering ordinary enterprise environments. The gaps set out below motivate this research.

First, most empirical work has studied large technology companies: firms with exceptional engineering talent, mature infrastructure, and cultures built for fast technology adoption. GitHub's internal Copilot study (Ziegler et al. 2024) and Microsoft's randomized field experiments across 4,867 developers (Cui et al. 2025) are methodologically strong, but the environments they describe differ from a typical mid-size software company in ways that matter. Developers at these firms tend to be more experienced, work on newer stacks, and carry fewer legacy constraints. Some recent work has moved outside that setting: Chen et al. (2026) surveyed 2,989 developers at a financial services firm, and Brandebusemeyer et al. (2026) studied an enterprise software vendor. Both organizations are nonetheless large, with dedicated tooling teams and formal governance. Whether any of this generalizes to companies that adopt the same tools without comparable support structures remains an open question.

Second, studies tend to pick one methodology and stay there. Some measure specific metrics under controlled conditions (Peng et al. 2023; Erhabor et al. 2025); others capture developer perception through observation and interviews (Barke et al. 2023; Wang et al. 2024). Few combine the two closely enough to connect a measurable productivity change to the subjective experience that decides whether developers keep using a tool. The separation matters because the two kinds of evidence can point in opposite directions. Stray et al. (2026) found perceived gains with no movement in commit-level metrics, and Chen et al. (2026), working from a large industrial sample, conclude that no single measure characterises AI's productivity impact. Storey et al. (2025) and Schoonenboom and Johnson (2017) make the methodological case directly: surveys cannot recover the "why" behind usage patterns, and telemetry cannot recover the "how" of subjective experience.

Third, variation across developer roles and experience levels has received limited attention. A backend engineer, a frontend developer, and a QA engineer work on different problems with different tools, yet studies that examine role usually compare adjacent functions rather than divisions inside the engineering team itself; Ulloa et al. (2026), for instance, look at how product managers delegate work to generative AI. Experience level has drawn more interest, with conflicting results. Cui et al. (2025) report higher adoption and larger gains among less experienced developers. Moradi Dakhel et al. (2023) argue close to the opposite, that Copilot is an asset to experts and a liability to novices who cannot filter out buggy output, and Qian and Wexler (2024) observe experts rejecting suggestions more readily than novices do. An organization deciding where to direct its AI tool budget has little settled evidence on either question.

Fourth, almost nothing is known about how these effects behave over time. Most studies measure a single task or a few weeks (Peng et al. 2023; Becker et al. 2025). Whether gains fade as novelty wears off, grow as developers work out better strategies, or settle somewhere in between is left open. Studies that follow teams for longer are rare (Stray et al. 2026; Tomaz et al. 2026), and there is reason to doubt that a single measurement holds: Shah et al. (2025) found trust rising between one hour and ten days of Copilot use.

Finally, the literature on practical difficulties is scattered across individual risks rather than assembled into an account of what a working developer actually runs into. The individual risks are well documented: insecure generated code (Pearce et al. 2022; Zhao et al. 2025), trouble reading and debugging long generated blocks (Vaithilingam et al. 2022), suggestions that miss project context (Davila et al. 2024), weak performance on complex, multi-file, and proprietary work (Pandey et al. 2024), and the pull toward excessive dependency (Cavalcante et al. 2025). What is missing is a working population ranking them against one another. Vendors cannot prioritise frustrations nobody has weighted, and organizations cannot train around barriers nobody has named.

## 1.2 Research Questions

**RQ1: How do AI-powered development agents impact software developer productivity?**

Productivity is the claim that sells these tools, so it is the first thing worth testing. The question stays broad on purpose, because productivity spans completion speed, output quality,

cognitive load, and the ability to sustain all of them at once, a multidimensional view formalized in the SPACE framework (Forsgren et al. 2021) and supported by work showing that single-dimensional metrics misrepresent knowledge work (Storey et al. 2021). Answering it here draws on quantitative measures (throughput, cycle time, task completion rates) alongside qualitative ones (perceived efficiency, shifts in how time is spent, workflow effects).

**RQ2: How does the impact vary across developer roles (Backend, Frontend, QA) and experience levels (Junior, Mid-level, Senior)?**

Roles differ in what they build and where they get stuck, so there is little reason to expect AI assistance to land evenly across them. Experience may moderate the effect as well, and could do so in either direction: juniors may lean on generated code, while seniors may value the clearing of routine work more than the code itself. Published findings conflict here. Cui et al. (2025) report larger gains for less experienced developers, while Moradi Dakhel et al. (2023) argue that novices are the least equipped to catch what AI gets wrong. Which pattern holds matters to any organization deciding who to train first, and to theories of human-AI collaboration.

**RQ3: What are developers' perceptions of AI tool effectiveness across different task types (code generation, debugging, testing, documentation)?**

Effectiveness almost certainly varies by task. Pandey et al. (2024) report up to 50% time saved on documentation and autocompletion and 30–40% on repetitive coding, unit test generation, and debugging, together with considerable trouble on complex tasks, large functions, multi-file changes, and proprietary contexts. A spread that wide means an aggregate productivity figure hides more than it shows. This question asks where the tools earn their keep and where they do not, which bears on how organizations deploy them and on what researchers should measure next.

**RQ4: What challenges, concerns, and limitations do developers face when using AI tools?**

Productivity gains are only half of what adoption produces. Developers also run into code that needs heavy modification, suggestions that are wrong or insecure (Pearce et al.

2022), skill development eroded by over-reliance (Anthropic 2026), reasoning errors such as automation bias and anchoring (Zhou et al. 2026), intellectual property questions, and friction with tooling already in place. Naming and ranking those obstacles is what makes a realistic assessment of the tools possible at all.

### 1.3 Scope and Limitations

**Scope:** One mid-size software company, 500+ employees and roughly 200 engineers. A survey (N=54, 54% response rate) ran in November 2025, supplemented by JIRA metrics covering seven months either side of widespread AI tool adoption across five anonymized teams. The tools in question are the mainstream ones adopted between 2023 and 2024 (GitHub Copilot, ChatGPT, Claude, OpenCode). Respondents therefore have 6-24 months of experience with them.

**Limitations:** A single-company case study supports analytical rather than statistical generalization (Yin 2018). The cross-sectional design cannot establish causation. Self-reported data carries response and common-method bias (Podsakoff et al. 2003), which the JIRA metrics partially offset. With N=54 AI users, statistical power for fine-grained subgroup analysis is limited (Cohen 1988); smaller role categories were consolidated into Backend, Frontend, and QA to make comparison possible, and even then the smallest group (QA, n=8) supports only modest power. The JIRA side covers five teams (N=5), too few for inferential testing, and the changes it shows cannot be attributed to AI adoption alone. Everything reported here reflects tool capabilities as they stood between 2023 and 2025.

## 2. LITERATURE REVIEW

This review draws on targeted searches of Google Scholar, IEEE Xplore, the ACM Digital Library, and Scopus, with arXiv preprints added for recent work not yet formally published. The search terms combined "AI coding assistant," "GitHub Copilot," "large language model code generation," "developer productivity," "software engineering AI tools," and "human-AI collaboration in software development." Coverage centres on publications from 2020 onward, the LLM era, with earlier foundational work included where it still frames the argument. Industry reports, practitioner blogs, and company research documents appear where they carry evidence the academic literature does not, and are identified as non-peer-reviewed wherever they are cited.

### 2.1 AI in Software Development

AI has changed how software gets written and maintained. The ambition itself is old, going back to the CASE tools of the 1980s, but LLMs have moved practice on a scale that invites comparison with high-level programming languages or the arrival of the IDE.

Productivity has been a preoccupation of the discipline for as long as it has existed. Brooks (1975) and DeMarco and Lister (1987) established that software development is a human activity before it is a technical one, bounded more by communication and cognition than by hardware. Brooks's "No Silver Bullet" argument held that no single technology would deliver a ten-fold productivity improvement within a decade, because the complexity of software is essential rather than accidental. Generative AI has revived the question of whether that ceiling still holds.

The modern era began with OpenAI's GPT-3 in 2020, which turned out to write code better than anyone had designed it to. Chen et al. (2021) introduced Codex, a GPT model fine-tuned on publicly available code, alongside the HumanEval benchmark that became the standard evaluation frame for code generation; Codex solved 28.8% of the benchmark problems on a first attempt. GitHub Copilot's 2021 launch turned the prototype into a commercial "AI pair programmer." By 2023 conversational agents such as ChatGPT and Claude had joined it, leaving developers with two distinct ways to work: completion inside the editor, and dialogue in a chat window. Rogers (2003) supplies the vocabulary for how quickly this spread, from the innovator phase to the early majority in under three years. Stack Overflow's 2025 Developer Survey, a self-selected practitioner poll of over 49,000

respondents, found 84% of developers using or planning to use AI tools and 51% using them daily (Stack Overflow 2025).

## 2.2 AI-Powered Code Generation Tools

The agents in use today run on LLMs trained across large bodies of public source code. They use the transformer architecture (Vaswani et al. 2017) to predict the next token in a sequence, which is enough to produce syntactically valid and often semantically appropriate snippets, functions, or whole classes from a natural language prompt or the surrounding code.

### 2.2.1 Technical capabilities

Capability has widened quickly. Early versions completed single lines, working as a more capable autocomplete. Huynh and Lin (2025) survey how LLM-based code generation extended from completion into generation and code search, then into debugging and cross-language translation. Rozière et al. (2024) introduced Code Llama, a family of open models reaching 67% on HumanEval and 65% on MBPP under a licence permitting commercial use, which put capable code models within reach of anyone unwilling or unable to depend on a proprietary API.

### 2.2.2 Available tools

The market has split into two interaction modalities:

1. **In-IDE Assistants:** GitHub Copilot and Amazon CodeWhisperer sit inside the editor, reading the active file and its context to suggest code as it is typed. Ziegler et al. (2024) locate their value in fewer interruptions to flow, since boilerplate and API lookups no longer break concentration.
2. **Conversational Agents:** ChatGPT, Claude, and Gemini work as chatbots, taking pasted code and returning explanations or architectural advice. Pendyala and Thakur (2025) evaluate such models on code translation drawn from Rosetta Code, finding marked prompt sensitivity under interpretability analysis and some open-source models outperforming expectations on generation.

Correctness and security remain unsettled. Ni et al. (2023) built L2CEval to evaluate language-to-code generation across seven tasks spanning semantic parsing, math reasoning, and Python programming, tracing how model size, pretraining data, instruction tuning, and prompting affect performance, and how well calibrated the models are about their own output. Evaluation practice has itself come under scrutiny. Crupi et al. (2025) asked whether LLMs can judge code reliably, putting eight models to work assessing the correctness of more than 2,600 Java and Python snippets and five models to rating roughly 1,200 code summaries against human judgement. Even the strongest model misjudged correctness often, and the smaller ones did considerably worse, which is a warning against reading benchmark scores as a proxy for what a developer will experience. Amazon Web Services (2024) documents the serving side, deploying Code Llama 70B and Mixtral 8x7B on managed cloud infrastructure. Princis et al. (2025) come at output quality from another angle with TreeCoder, treating decoding strategies and constraints as tunable parts of a tree search over candidate programs instead of leaning on prompt engineering, and report gains over unconstrained baselines on MBPP and SQL-Spider.

## 2.3 Developer Productivity Measurement

What productivity means, and how to measure it, remains among the most contested questions in software engineering. Lines of code and velocity points have drawn criticism for decades on the grounds that they miss the creative and knowledge-intensive character of the work (Storey et al. 2021). Applied to AI tools, a speed-only measure ignores whether the software produced is any good.

### 2.3.1 The SPACE framework

Forsgren et al. (2021) proposed SPACE as an answer to single-dimensional metrics, splitting productivity into satisfaction and well-being, performance, activity, communication and collaboration, and efficiency and flow. The distinction earns its keep when evaluating AI tools, where a rise in activity (more code written) may arrive with a fall in satisfaction (more time debugging what the model produced) or in communication (less discussion during review). Noda et al. (2023) reduce developer experience to three measurable dimensions, feedback loops, cognitive load, and flow state, arguing that improving how the work feels produces better and more durable outcomes than optimising output measures that are easy to game.

### 2.3.2 DORA metrics

DORA (DevOps Research and Assessment) approaches the same problem from the delivery side, tracking deployment frequency, lead time for changes, time to restore service, and change failure rate (Forsgren et al. 2018). Practitioner guidance treats the two frameworks as complements rather than alternatives: CodePulse Team (2026) and Codex (2026) both recommend starting with DORA, whose four metrics come out of version control and deployment data without a survey, then adding SPACE for the satisfaction and collaboration dimensions DORA cannot see. Neither source addresses AI tooling. Extending their reasoning: if AI agents make developers faster without costing quality, that should surface as shorter lead time for changes with no rise in change failure rate.

### 2.3.3 Challenges in measurement

Isolating AI's contribution to any of these metrics is hard. Storey et al. (2021) warn against the "streetlight effect," measuring what is easy such as suggestion acceptance rate instead of what matters such as delivered feature value. Brannon (2026), writing for practitioners, argues that pairing DORA with SPACE gives the balanced view, and this thesis adopts that pairing. Dong (2026) proposes "queue health" as a review metric on the argument that review throughput is a question of the quality of attention rather than its speed: as queues lengthen, reviewers skim and feedback goes perfunctory. The concern sharpens wherever AI tools push more code into review.

## 2.4 Empirical Studies on AI Coding Assistants

Empirical work has begun to put numbers on the impact, and the numbers disagree with each other depending on where they were collected.

### 2.4.1 Controlled experiments

The early controlled studies reported dramatic gains. Peng et al. (2023) found developers using GitHub Copilot completing a standardized HTTP server implementation 55.8% faster than a control group. Ziegler et al. (2024) at GitHub found that developers who accepted Copilot suggestions felt more productive and reported spending less time searching for information. Between them they established the speed argument for adoption.

### 2.4.2 Field studies and real-world evidence

Away from controlled tasks the results scatter. Cui et al. (2025) ran randomized field experiments with 4,867 developers at Microsoft, Accenture, and an anonymous Fortune 100 company, conducted by those companies in the ordinary course of business. Pooled across the three, developers given access to an AI coding assistant completed 26.1% more tasks, and the less experienced among them showed both higher adoption and larger gains. The authors are careful to note that the effect runs considerably smaller in the field than in the laboratory, since only part of a developer's work is amenable to AI assistance in the first place. Bakal et al. (2025) add an enterprise view from a ZoomInfo deployment covering more than 400 developers: a 33% acceptance rate for Copilot suggestions, 20% for lines of code, 72% developer satisfaction, and performance that varied noticeably by programming language.

Later work has complicated the picture further. Becker et al. (2025), in a randomized controlled trial run by METR, found AI tools slowing experienced open-source developers by 19%, which is hard to reconcile with any claim of universal benefit. Stray et al. (2026) followed a government IT agency longitudinally and found developers perceiving improvement while objective commit-level metrics stayed flat, a perception-reality gap worth taking seriously. Chen et al. (2026) reach a compatible conclusion from a much larger base, pairing a survey of 2,989 developers at a financial services firm with eleven interviews. Developers there disagreed sharply about how much the tools helped, and the factors they associated with their own productivity ran well past delivery speed into skill development and ownership of the resulting code, neither of which commit-level telemetry can see. Their conclusion, that no single measure suffices, is the direct motivation for the triangulated design used in this thesis.

Davila et al. (2024) surveyed 72 employees of a Brazilian agroindustry company and found ChatGPT and GitHub Copilot adopted mainly to speed up online searching, typing, and syntax recall, with the recurring difficulty being suggestions that lack context, which practitioners work around by writing longer descriptions. Cavalcante et al. (2025) reach a compatible conclusion from a legacy maintenance case study, where automating repetitive work freed attention for harder problems but demanded constant review of what was generated and raised the risk of excessive dependency. Pandey et al. (2024) quantify the upside across 15 development tasks on large proprietary codebases: up to 50% time saved on documentation and autocompletion, 30–40% on repetitive coding, unit test

generation, and debugging, alongside clear trouble with complex tasks, large functions, multi-file changes, and proprietary contexts.

### 2.4.3 Impact on code quality

Quality is where the disagreement is sharpest. Standard implementations often come back clean; subtle logic errors are the problem. Moradi Dakhel et al. (2023) asked directly whether GitHub Copilot is an asset or a liability and found it solving almost every fundamental algorithmic problem they tested, though some solutions were buggy or non-reproducible and humans produced correct solutions at a higher rate. Their conclusion is conditional rather than general: an asset to expert developers, whose suggestions it matches, and a liability to novices who cannot filter buggy or non-optimal output. Maintainability is not the only exposure. Erhabor et al. (2025) measured the runtime performance of C++ code written with Copilot, drawing attention to something functional test suites never check, since code can pass every test and still run slower than what a developer would have written alone. Vaithilingam et al. (2022) found participants struggling to understand, edit, and debug long generated blocks, which limits their ability to catch the defects the model introduced. De La Cruz et al. (2025) raise the same worry at scale, warning that broad reliance on LLMs could spread insecure coding patterns unless verification practices keep up.

## 2.5 Trust and Adoption

Trust mediates adoption, and the evidence on how it develops does not converge. Shah et al. (2025) surveyed 71 upper-division computer science students working on a legacy code base and found average trust rising between one hour and ten days of Copilot use, even as those same participants concluded that some tasks still need a competent programmer working by hand. Qian and Wexler (2024) watched 76 software engineers during a programming exam and found attitude and behaviour diverging: reliance on the assistant grew as the task went on while reported trust fell, and experts rejected suggestions more readily than novices. Wang et al. (2024) interviewed 17 developers, then tested three sets of interface concepts with 12 more, covering suggestion quality indicators, usage statistics, and control mechanisms; their argument is that calibrated trust is a problem for the interface to solve rather than a trait of the individual developer. Rogers's (2003) diffusion theory points the same way, treating trialability and observability of benefits as what carries an innovation from early adopters into the mainstream.

## 2.6 Developer Experience and Human-AI Collaboration

AI agents turn a largely solitary activity into a collaborative one, which changes what developer experience (DevEx) even refers to.

### 2.6.1 Cognitive load and flow

Reduced cognitive load is the benefit developers cite most. Offloading syntax recall and boilerplate lets them hold onto flow and spend attention on the problem rather than the keyboard (Ziegler et al. 2024). Sabouri et al. (2025) qualify this, finding comprehensibility and perceived correctness driving those judgements and developers accepting only 52% of suggestions while frequently revising their initial decisions. Reading AI-written code, which may use unfamiliar patterns, can cost more mentally than writing it would have. Chukkala (2025) frames the arrangement as a synergistic division of labour, with AI absorbing routine work while humans keep creativity, ethical judgement, and the contextual understanding that hard problems require.

### 2.6.2 Interaction patterns

Barke et al. (2023), in their "Grounded Copilot" study, name two modes of interaction: acceleration, where the programmer knows what comes next and uses the tool to get there faster, and exploration, where the programmer does not and uses the tool to survey options. Developers switch between them constantly, which has consequences for both tool design and productivity measurement. Vaithilingam et al. (2022) had observed something related earlier with 24 developers, finding that Copilot improved neither task completion time nor success rate while most participants still preferred working with it, because it gave them a starting point and saved them searching online. Perceived and measured productivity were already coming apart. Prompt engineering is often described as the new competency separating those who can use these tools from those who cannot, but the size of that divide should not be overstated. Khojah et al. (2025) tested five prompt techniques across 7,072 prompts and three models, covering few-shot examples, personas, chain-of-thought, function signatures, and package lists, and found that combining techniques rarely produced statistically significant gains and that correctness improvements sometimes cost code quality. Supplying explicit type information had the clearest effect. On their evidence, prompt skill matters less as a source of differential productivity than the verification and review practices built around it.

### 2.6.3 Mixed methods research

No single instrument captures human factors of this kind. Storey et al. (2025) and Schoonenboom and Johnson (2017) argue that surveys alone (Likert 1932) cannot reach the "why" behind usage patterns while telemetry alone misses the "how" of subjective experience. This thesis follows the guidance of Storey et al. (2021, 2025) in combining quantitative productivity metrics with qualitative developer insight. Brandebusemeyer et al. (2026) show what the pairing yields in a mixed-methods field study at an enterprise software vendor, where professional developers completed Java tasks with and without Copilot in their own working environment and reported workload afterward using the NASA Task Load Index. Moderate use improved efficiency and lowered perceived workload; excessive use, or mixing several interaction types at once, gave those gains back.

## 2.7 Challenges and Concerns

Set against the optimism is a body of work on the difficulties, and on whether the value survives contact with them.

### 2.7.1 Security and quality risks

Introduced vulnerabilities are the first concern. Pearce et al. (2022) ran a security analysis across 89 security-relevant scenarios and found roughly 40% of the 1,689 programs Copilot produced carrying vulnerabilities, among them SQL injection, path traversal, and improper input validation. Models trained on large corpora of existing code reproduce the insecure patterns that corpus contains. Zhao et al. (2025) map the problem onto the Common Weakness Enumeration taxonomy, which turns "insecure" from one undifferentiated risk into a structured account of which weakness classes recur. Mitigation is an active front: Huang et al. (2026) apply contrastive decoding to steer generation away from vulnerable constructions. On that evidence security might be addressable during generation rather than only in review afterward. The opacity of these models compounds all of it, since being unable to explain why a particular solution was generated is a compliance problem in regulated industries (Pendyala and Thakur 2025).

### 2.7.2 Skill atrophy and over-reliance

Fear of skill atrophy runs strongest around junior developers. A novice who leans on AI for problems they do not understand may never build the mental models senior engineering requires, and a hollowed-out skill curve would take years to show up and years more to repair.

Zhou et al. (2026) investigate the cognitive biases that surface in LLM-assisted development, among them automation bias, anchoring, and weakened critical evaluation of generated code. Their finding is that the interaction itself introduces systematic reasoning errors, which traditional metrics have no way to detect. Anthropic (2026), in a company research report, ran a randomized controlled trial in which 52 mostly junior engineers learned an unfamiliar Python library; those working with AI assistance scored 17% lower, close to two letter grades, on a quiz covering concepts they had used minutes earlier, while the speed gain that came with it was not statistically significant. What that measures is skill formation during learning rather than long-term atrophy, which the study did not follow, but it puts evidence behind a concern that had been running on intuition.

## 2.8 Research Gap

The literature is growing quickly, and three gaps in it shape what this thesis sets out to do.

First, the empirical base sits in controlled laboratories or inside elite technology companies, Microsoft, GitHub, and Google among them (Peng et al. 2023; Ziegler et al. 2024). Mid-sized organizations, working under different constraints with legacy systems and a different talent profile, are barely represented, and there is no reason to assume Big Tech findings survive the move. Recent work has started to push outward, with Chen et al. (2026) studying a large financial services firm and Brandebusemeyer et al. (2026) an enterprise software vendor, but both remain large organizations with dedicated tooling teams and formal governance. The mid-size company that adopts identical tools without any of that support is still largely missing from the evidence.

Second, developers are usually treated as one undifferentiated group. Few studies disaggregate by role, frontend against backend against QA, or by experience level with enough granularity to see anything, and the value of AI assistance probably does differ across

those cohorts (Cui et al. 2025). Where role has been examined, the attention has gone to adjacent functions rather than to divisions inside the engineering team; Ulloa et al. (2026) study how product managers delegate work to generative AI and stay accountable for the result. The equivalent breakdown among backend, frontend, and QA engineers has not been done.

Third, studies measure either perceived productivity (Ziegler et al. 2024) or speed, and rarely triangulate the two against objective delivery metrics from DORA or JIRA in a real setting over time. Storey et al. (2025) argue that integrating both kinds of evidence is precisely what lets a study notice when a gain in one measure fails to reach outcomes. Chen et al. (2026) arrive at the same place from the other direction, showing that commit-level measurement alone misses the factors developers themselves treat as central.

This thesis addresses those gaps with a mixed-methods case study in a mid-size software company, analyzing how the impact of AI agents differs across developer roles and experience levels.

# 3. MATERIALS AND METHOD

## 3.1 Research Design

This study used a mixed-methods case study to investigate how AI-powered development agents affect software developer productivity. Quantitative survey data and qualitative open-ended responses were collected together, covering developer workflows, productivity metrics, code quality, and professional experience.

The case study design followed from the nature of the phenomenon. AI-powered development agents are recent and still moving, and case studies suit contemporary phenomena examined inside their real-world context (Yin 2018). The research questions go past whether productivity changes to how and why developers experience the change across roles, experience levels, and task types, which needs contextual detail that a narrower design would strip out. The organizational setting also draws a natural boundary around the study while still containing a varied developer population.

A multi-company design was considered. The intention was to include the researcher's previous employer alongside the current organization, allowing comparison across two sites with different cultures and technical stacks. That company was undergoing layoffs and organizational instability during the study period, which made it unsuitable: response rates would have suffered, and any productivity metric drawn from it would have been confounded by the disruption. The study proceeded as a single-company case as a result, which is the source of the external validity limitation discussed in Section 3.7.2.

Combining methods brings quantitative measurement of productivity perceptions, code quality assessments, and usage patterns together with qualitative exploration of what developers experience, struggle with, and adapt to. Triangulating this way strengthens validity by letting findings converge across sources (Creswell and Creswell 2018). The quantitative side gives breadth and supports statistical analysis of relationships among experience level, usage frequency, and perceived productivity. The qualitative side gives depth, reaching developer reasoning and contextual factors that numbers alone leave invisible.

## 3.2 Research Context

The research took place at a mid-size software development company employing over 500 people, roughly 200 of them software engineers spread across multiple product teams. The company name and identifying details are anonymized here to protect organizational privacy. The organization builds enterprise software with distributed teams organized around several product lines.

Participants came from across the engineering function. Backend engineers handle server-side logic, databases, and APIs; frontend engineers work on user interfaces and client-side applications; quality assurance engineers run testing and automation; and several more specialized roles sit alongside them. Experience runs from interns and junior developers with 0-2 years, through mid-level developers at 2-5 years, to senior developers with 5+ years and technical leads or architects with considerably more.

Adoption widened through 2024 to take in Claude, OpenCode, Gemini, and specialized tools including Cursor and Tabnine. By the time this survey ran in November 2025, AI coding tools were embedded in standard development workflows, though adoption remained voluntary and usage varied a great deal between individuals.

This is a typical mid-stage adoption environment: tools available and encouraged rather than mandated, developers free to experiment with their own workflows, enthusiasts and skeptics working side by side. That range of adoption patterns is what makes the organization a useful case study site, since it contains the variation the research questions are about.

### 3.2.1 Researcher positionality

The researcher is an employee of the case organization and a daily user of the tools under study, having held a personal ChatGPT subscription since 2023 and followed developments in the field closely since then. This carries both advantages and costs. Insider access made survey distribution, JIRA extraction, and the interpretation of team-level metrics possible in ways an external researcher could not have managed, and familiarity with the teams involved supported the reading of their delivery data. It also introduces bias: a researcher who already finds these tools valuable may frame questions and interpret re-

sponses in ways that favour a positive result. The organization itself actively encourages AI adoption, which makes it unrepresentative of organizations that are neutral or resistant toward these tools. The findings should be read with both factors in mind.

### 3.3 Survey Design

Data collection ran primarily through an online survey capturing quantitative measurements alongside qualitative insight into AI tool usage, productivity impacts, code quality perceptions, and developer experience. The instrument was developed following established practice for software engineering survey research (Kitchenham and Charters 2008) and human-computer interaction studies (Lazar et al. 2017).

Design began with a review of the literature on developer productivity measurement, AI coding assistant research, and survey methodology in software engineering. From that review came an initial question set covering demographics, tool usage patterns, productivity metrics, code quality assessments, cognitive load factors, and open-ended experience descriptions. Pilot testing used three colleagues outside the target sample, chosen for different vantage points: a senior backend engineer working as an architect, a QA engineer, and a senior manager with an organization-wide view. The manager could judge whether the questions would mean anything at organizational level, the architect whether the technical framing was right, and the QA engineer brought a reading of how developers would actually interpret the wording rather than how it was intended. They commented on question clarity, survey length, response option adequacy, and technical terminology, and their feedback led to several questions being reworded and response scales standardized.

The most consequential change to come out of the pilot was a suggestion to break the analysis down by demographic group, which shaped the role and experience-level comparisons that became RQ2.

The instrument also carried a conditional branch for developers who did not use AI tools, intended as a comparison group: reasons for not adopting, what might motivate them to try, their current workflow, and what an ideal assistant would do. Too few respondents were routed to it to support any comparison, so the branch is not analysed and the analytical sample consists entirely of AI tool users.

The final instrument holds 49 individual items across four sections, presented as 23 question prompts. Six of those prompts use a matrix (grid) format covering 32 individually rated items between them; the remaining 17 are single-response or open-ended. Survey length traded coverage against the time participants could reasonably give it. Pilot testing put average completion at 8-12 minutes, within accepted ranges for professional survey research (Dillman et al. 2014).

### 3.4 Data Collection

Data collection ran during November 2025 through an anonymous online survey distributed on the organization's internal communication channels. The instrument was implemented in Google Forms for accessibility, ease of completion, and secure collection. Distribution followed organizational research ethics protocols and was approved by the relevant stakeholders.

An invitation email went to approximately 100 software engineers across multiple product teams, explaining the research purpose, confirming that participation was voluntary, guaranteeing anonymity, and carrying the survey link. The email stated that no individual responses would reach management and that all data would be reported in aggregate. Nothing personally identifiable was collected: the survey captured role type, experience level, and usage patterns, and no names, employee IDs, or team affiliations.

The survey stayed open for two weeks, with a reminder email at the one-week mark. No incentives were offered, so the responses came from willingness to contribute rather than external motivation.

Fifty-four complete responses came back from approximately 100 invitations, a response rate near 54%. Every major engineering discipline and experience category is represented. That rate sits comfortably inside the 30% to 60% range typical of organizational surveys in software engineering (Dillman et al. 2014). Several things probably helped: the survey was short at 8-12 minutes, the topic mattered to the people being asked, and the organizational culture supports research participation. Non-response bias cannot be ruled out, since engineers who stayed silent may hold systematically different views of AI tools, but the spread of roles, experience levels, and attitudes among those who did respond suggests reasonable representativeness. The anonymity safeguards above were intended to make candid answers possible for those who took part.

## 3.5 Survey Instrument

The full instrument is reproduced in Appendix 1.

### 3.5.1 Section 1: General information

The opening section collected the demographic and background information needed for subgroup analysis. Question 1 asked participants to identify their professional background from five options: Backend, Frontend, DevOps, QA & Automation, or Other. Question 2 captured experience level using six categories: Intern, Junior, Mid-level, Senior, Team Lead/Technical Lead, or Product Owner/Manager. Question 3 asked for primary programming language, offering C#, JavaScript/TypeScript, Java, Python, PHP, C/C++, or Other.

Question 4 asked how long participants had been using AI coding tools, with response options ranging from Less than 1 month to More than 2 years.

### 3.5.2 Section 2: AI tool usage patterns

This section characterized how participants engage with AI tools in practice. Question 5 used a multiple-selection format to identify the type of AI tools used, distinguishing between web-based tools (e.g., ChatGPT, Claude, Gemini, not integrated with the codebase) and codebase-integrated tools (e.g., GitHub Copilot, integrated within the IDE). Question 6 identified the specific AI tools currently used via a checkbox list including GitHub Copilot, OpenCode, JetBrains AI Assistant, ChatGPT, Claude, Gemini, DeepSeek, Grok, Windsurf, Cursor, Kiro, and Other.

Question 7 measured usage frequency with four response options: Daily, Several times a week, Once a week, and Rarely. Question 8 captured license type using a multiple-selection format: Free, Paid Personal, and Business/Enterprise.

### 3.5.3 Section 3: Tool effectiveness

This section, the largest in the instrument, assessed how participants interact with and rely on AI-generated output. Question 9 asked how often participants check AI-generated code

before accepting it (Always, Usually, Sometimes, Rarely, Almost Never), and Question 10 asked how often AI tools provide the needed solution on the first try (Always, Usually, Sometimes, Rarely, Almost never). Question 11 asked how many attempts participants typically make when an AI tool does not provide the right solution initially, ranging from 1 attempt to more than 10. Question 12 asked how often AI tools suggest more code than needed (Always, Often, Sometimes, Rarely, Never), and Question 13 asked what participants check first when encountering a coding question (AI tools, Google search, Stack Overflow, official documentation, or colleagues).

The remaining six questions in this section used a five-point matrix (grid) format, in which participants rated multiple items on a single scale. Question 14 asked participants to rate the change, since adopting AI tools, in five areas: overall productivity, code quality, meeting deadlines, focus on complex tasks, and overall work experience (1 = Significantly decreased to 5 = Significantly increased). Question 15 asked participants to rate AI tool effectiveness across seven activities: design and architecture planning, technical documentation and knowledge management, team collaboration and communication, reducing time spent on repetitive coding tasks, creating comprehensive unit tests, bug diagnosis and troubleshooting, and discovering missed edge cases or business logic scenarios (1 = Not effective to 5 = Extremely effective). Question 16 asked participants to rate how their dependency on five alternative resources had changed since adopting AI tools: Stack Overflow, official documentation, technical blogs and articles, video tutorials, and colleague consultation (1 = Much less dependent to 5 = Much more dependent). Question 17 asked participants to rate their agreement with two statements: that AI tools improve their learning of new technologies, and that AI tools help them understand new codebases (1 = Strongly disagree to 5 = Strongly agree). Question 18 asked participants to rate AI tool effectiveness across eight specific coding task types: writing boilerplate code, implementing algorithms, API integration, database queries, unit test creation, code refactoring, documentation generation, and debugging assistance (1 = Not effective to 5 = Extremely effective). Question 19 asked participants to rate their level of concern regarding five potential impacts of AI tools on their professional development: becoming too dependent on AI assistance, losing fundamental programming skills, reduced problem-solving abilities, impact on career advancement, and job security concerns (1 = Not concerned at all to 5 = Extremely concerned). Collectively, these six matrix questions captured 32 individually rated items.

### 3.5.4 Section 4: Return on investment and open-ended questions

The final section paired one closed quantitative item with open-ended ones. Question 20 asked how participants quantify the return on investment from AI tools, using a multiple-selection format: time saved per day/week, increased project completion rate, reduced debugging time, improved code quality metrics, or "I don't measure ROI." Three open-ended questions closed the survey. Question 21 asked participants to describe the main challenges or disadvantages they had experienced with AI coding tools; Question 22 asked whether there were additional benefits not otherwise captured; and Question 23 asked what improvements they would like to see. These items gave participants room to raise concerns, benefits, or experiences the structured questions missed, and supplied the material for thematic analysis.

## 3.6 Data Analysis Methods

Table 3.1 presents the mapping between research questions, survey items, and analytical methods employed to address each question systematically.

Table 3.1 Research question to survey item and analysis method mapping.

| **Research Question** | **Survey Items** | **Analysis Methods** |
|---|---|---|
| RQ1: Productivity impact | Q14 (productivity matrix), Q7 (usage frequency) | Descriptive statistics, Spearman correlation |
| RQ2: Role & experience variation | Q1-Q2 (demographics) × Q14 | Kruskal-Wallis H-test, cross-tabulation |
| RQ3: Task-specific effectiveness | Q15, Q18 (effectiveness matrices) | Descriptive statistics, cross-tabulation by role |
| RQ4: Challenges & concerns | Q19 (concern matrix), Q21-Q23 (open-ended) | Thematic analysis, frequency counts |

### 3.6.1 Quantitative analysis

Descriptive statistics were computed for every quantitative variable to characterize the sample and summarize how responses distributed. Demographic and categorical responses were reported as frequency distributions and percentages. Likert items were summarized with means, medians, standard deviations, and five-number summaries

covering minimum, first quartile, median, third quartile, and maximum.

Cross-tabulation examined how demographic variables (role, experience level) relate to the key outcomes (productivity ratings, code quality perceptions, tool usage frequency), with contingency tables displaying joint distributions so response patterns could be compared across subgroups.

Inferential tests then assessed whether the differences between groups were statistically significant. The Kruskal-Wallis H-test served as the primary test for comparing distributions across three or more independent groups, chosen as the non-parametric alternative to one-way ANOVA because Likert data are ordinal rather than genuinely interval-level and the test assumes nothing about normality (Kruskal and Wallis 1952). It was applied to productivity ratings, code quality assessments, and other ordinal outcomes across experience levels (Intern, Junior, Mid-level, Senior, Lead) and professional roles (Backend, Frontend, QA).

Spearman rank correlation coefficients measured monotonic relationships between ordinal variables, again because Spearman's rho suits ordinal data and assumes neither linearity nor normality (Spearman 1904). It was used on usage frequency against productivity ratings, experience level against productivity ratings, usage frequency against code quality ratings, and code review frequency against code quality ratings.

Chi-square tests of independence covered the categorical variables, testing for association between professional role and usage frequency, between experience level and usage frequency, and between professional role and license type.

Significance was evaluated at alpha = 0.05. Effect sizes are reported alongside p-values so that practical significance is visible rather than inferred. All quantitative analysis ran in Python using pandas, numpy, scipy, and matplotlib.

Several alternative statistical approaches were considered and rejected, on grounds of the ordinal response data, the achieved sample size, and the cross-sectional design.

**Parametric tests** such as one-way ANOVA, independent-samples t-tests, and Pearson correlation assume interval-level measurement and approximately normal distributions.

Likert scale items are ordinal, and small subgroup sizes (Interns n=2, Team Leads n=2) preclude meaningful normality assessment. The selected non-parametric counterparts, the Kruskal-Wallis H-test and Spearman rank correlation, are the recognized alternatives for ordinal data and impose no distributional assumptions.

**Regression-based methods**, including linear regression and ordinal or multinomial logistic regression, would permit modeling productivity perceptions as a function of multiple predictors simultaneously. However, conventional guidelines require 10–20 observations per predictor for stable parameter estimation, a threshold the achieved sample (N=54) cannot satisfy for any reasonably specified model. The high inter-correlation among related Likert items (the five items of Q14 each addressing a facet of productivity) would further introduce multicollinearity, destabilizing coefficient estimates and inflating standard errors.

**Multivariate and latent-variable methods**, including MANOVA, exploratory and confirmatory factor analysis, and structural equation modeling, were not pursued because such techniques typically require sample sizes substantially larger than N=54 for reliable estimation, with structural equation modeling generally requiring $N \geq 200$. The research questions also concern group differences and bivariate associations rather than the structure of latent constructs, which puts latent-variable modeling at odds with the study's aims.

**Within-subjects tests** such as the Wilcoxon signed-rank test, the Friedman test, and the paired-samples t-test require repeated measurements on the same individuals. The cross-sectional, between-subjects survey design precludes their application, as no individual-level pre/post measurement of AI tool adoption was collected.

**Fisher's exact test** would in principle be preferable to the chi-square test of independence for contingency tables containing cells with expected frequencies below 5. Chi-square was retained for comparability with prior software engineering survey research that conventionally reports the statistic; the resulting p-values are accordingly interpreted as approximate indicators of association rather than precise probability estimates.

A post-hoc power analysis assessed whether N=54 could detect meaningful effects. For the Kruskal-Wallis tests comparing experience-level groups (k=5) and role groups (k=3), power came to approximately 0.38 and 0.48 respectively for medium effect sizes (f=0.25) at $\alpha$=0.05, which leaves the study underpowered for reliable detection of moderate group

differences. Power to detect a Spearman correlation of $\rho$=0.30 was approximately 0.55 at $\alpha$=0.05. Insufficient sample size is therefore the study's primary statistical limitation, and non-significant findings here should be read as inconclusive rather than as evidence of no effect. Reaching 80% power for Kruskal-Wallis tests detecting medium effects across five experience-level groups would need roughly 180 respondents.

### 3.6.2 Qualitative analysis

Open-ended responses were analyzed thematically, following Braun and Clarke's (2006) method for identifying and reporting patterns within qualitative data. The approach was inductive, letting themes emerge from the responses rather than sorting them into a framework decided in advance.

Analysis moved through six phases. All open-ended responses were read several times to build familiarity with their content and depth. Initial codes were then generated by working systematically across the dataset, marking features, concerns, benefits, and patterns; the codes stayed data-driven and descriptive, capturing what respondents actually said. Related codes were collated into candidate themes, which were then reviewed for internal coherence and fidelity to the data. Themes were defined and named with a clear statement of what each covers. Finally they were integrated into the research narrative with representative quotations.

This produced contextual information about developer experience, surfaced concerns and benefits the structured questions had not anticipated, and generated hypotheses for later research. Thematic findings were then read alongside the quantitative results.

## 3.7 JIRA Project Management Metrics Analysis

To set objective behavioral indicators against the self-reported survey data, the study drew productivity metrics from the organization's JIRA project management system. JIRA is the primary work item tracking platform across all development teams, holding issue creation, status transitions, resolution timestamps, story point estimates, and issue type classifications. Team selection ran in two stages. Candidate teams were first filtered on active development volume and data completeness across the study period, and the five teams reported here were among the highest on both counts. Several other teams met the same

criteria, and the final choice among them rested on the researcher's familiarity with each team's domain and work, since interpreting movements in throughput and cycle time requires knowing what kind of work a team does. This introduces a convenience element into the sampling: the five teams are not a random sample of the organization's development teams, and their metrics may not generalize to teams the researcher knew less well. Team identifiers were replaced with pseudonyms, Team Alpha through Team Epsilon, to preserve organizational confidentiality. No individual developer data was extracted or analyzed; every metric is aggregated at team level.

The analysis compares two seven-month periods: a pre-AI baseline from January 1 through July 31, 2025, and a post-AI period from August 1, 2025 through February 28, 2026. The boundary comes from a specific event. At a company-wide meeting in August 2025 it was announced that every engineer was eligible for a GitHub Copilot license, use of AI tools was encouraged, and a dashboard tracking adoption across the organization was introduced.

That dashboard deserves noting as a limitation on everything that follows. It was public, and it showed each engineer's monthly AI usage together with the cost that usage incurred, visible to every other engineer. Per-person visibility of that kind creates pressure in two directions at once: toward using the tools, since visible non-use stands out against colleagues who are using them, and toward describing them favourably, since anyone whose consumption is visibly costing the company money has reason to say it is working. Both bear on the survey responses analysed in Chapter 4 and compound the social desirability bias discussed in Section 3.7.1.

The dashboard is also a reason to read the post-AI period as an organizational push rather than organic adoption alone. Some developers, the researcher included, were using these tools well before the announcement, so the boundary marks the point at which use became official and encouraged rather than the point at which it began. Stories, Tasks, Bugs, Production Bugs, and Sub-tasks were included. Epics, Initiatives, and administrative items were excluded as planning artifacts rather than units of deliverable work.

Four metrics carry the pre-post comparison. *Throughput* is the average number of issues resolved per month per team, measuring volume of work completed. *Cycle time* is the median calendar days from issue creation to resolution, capturing workflow efficiency. *Bug ratio* is the percentage of bug and production bug issues against total issues created

in a period, standing in as a defect proxy. *Story point velocity* is average monthly story points completed, reported only for teams with at least 70% non-null story point coverage in both periods. The 70% threshold was a judgment call rather than a figure drawn from prior work: it was set high enough that a reported velocity would rest on most of a team's issues rather than a minority of them. In the event no team cleared it, and velocity is not reported for any team (Section 4.9.3).

Metrics were aggregated monthly rather than by sprint, because sprint definitions and cadences differed across the five teams and could not be compared directly. The approach is strictly descriptive: percentage change between pre-AI and post-AI period averages, computed per metric per team, alongside cross-team averages. No inferential tests were applied, as the small number of teams (N=5) cannot support hypothesis testing. These metrics triangulate the survey findings; they are not standalone evidence of AI tool impact.

Extraction ran programmatically through the JIRA REST API using read-only credentials. The scripts retrieved all qualifying issues for each team across the study period, anonymized team identifiers and issue keys, classified issues by period on creation date, and exported the anonymized datasets to CSV. Scripts and datasets are stored separately from the thesis source files to prevent accidental disclosure of organizational data. Extraction yielded 5,169 issues across the five teams.

## 3.8 Validity and Reliability

### 3.8.1 Internal validity

Internal validity concerns whether the study accurately captures causal or associational relationships between AI tool usage and productivity outcomes. Several threats were mitigated by design. Inviting every software engineer in the organization, rather than sampling known enthusiasts or critics, addressed selection bias. Response anonymity reduced social desirability bias by letting participants answer honestly without worrying about managerial scrutiny. Questions were framed both positively and negatively to counter acquiescence bias, the tendency to agree with a statement regardless of its content.

Causality nonetheless stays out of reach with cross-sectional survey data. An observed association between AI tool usage and productivity may reflect self-selection, where more

productive developers adopt the tools, rather than any effect of the tools themselves. Establishing causality would take longitudinal collection or an experimental design.

Three further threats deserve naming. Common method bias comes first. Predictor and outcome variables, usage frequency and perceived productivity for instance, are measured through the same self-report instrument, and common method variance can inflate the correlations that result (Podsakoff et al. 2003). Objective behavioral metrics reduce that threat, which is part of why the JIRA data was brought in.

Social desirability is the second, and the organizational setting sharpens it beyond the usual case. The company ran a public dashboard showing each engineer's monthly AI usage and its cost, visible to everyone (Section 3.5). Participants may therefore have felt pressure to report positive experiences, particularly anyone whose consumption was visible and attributed to them by name. Survey anonymity mitigates this but does not remove it, since a pressure of that kind shapes how developers have come to think about the tools, not only what they are willing to write down about them.

The third is the novelty effect. With 46.3% of respondents having used AI tools for less than six months, the productivity perceptions reported here may partly reflect enthusiasm rather than settled assessment. The 53.7% at six months or more counterbalance that to a degree. Only a longitudinal design tracking perceptions over time would separate novelty from durable benefit.

### 3.8.2 External validity

External validity concerns how far findings travel beyond the case. This research covers a single mid-size software development organization, which limits generalizability by construction. The findings may not transfer to small startups, large enterprises, other industries, or organizations with different AI tool policies. That said, the characteristics of this organization are widely shared: varied roles and experience levels, voluntary adoption, enterprise licenses available, modern development practices. Case study research also trades breadth for depth, aiming at analytical generalization to theoretical concepts rather than statistical generalization to populations (Yin 2018). What the findings contribute is theoretical understanding of how AI tools affect developer work, which future research across other contexts can test.

### 3.8.3 Construct validity

Construct validity concerns whether the instruments measure the constructs they claim to: productivity, code quality, cognitive load, and the rest. The survey was built on established constructs from software engineering and human-computer interaction research, and Likert scales for productivity and code quality perceptions have precedent in prior developer survey work. Complex constructs were measured with multiple items, so productivity was assessed through ratings of overall productivity change, meeting deadlines, and focus on complex tasks together, which captures different facets of one concept. Pilot testing indicated that questions were read as intended and that response options fit. Self-reported perception may still diverge from objective performance, which is why JIRA metrics were integrated: throughput, cycle time, and bug ratio data from five development teams across pre-AI and post-AI periods triangulate the perceptions against team-level behavioral indicators.

### 3.8.4 Reliability

Reliability concerns whether measurements are consistent and repeatable. Pilot feedback was used to word questions clearly enough that respondents would read them the same way. Standardized Likert scales with defined anchor points reduce measurement error. Because the survey was anonymous and online, every participant met identical questions in identical order, which removes interviewer effects and inconsistent administration. Internal consistency for multi-item constructs can be assessed with Cronbach's alpha, which measures whether items intended to capture one construct produce similar scores; here, though, the instrument was designed to cover diverse facets of AI tool impact rather than a single unified construct, so high internal consistency across all items is neither expected nor desirable. The open-ended questions offered a way to cross-validate the structured responses, and the convergence between quantitative ratings and qualitative description strengthens confidence in the findings.

## 3.9 Ethical Considerations

Research with human participants in a workplace raises questions of informed consent, privacy, confidentiality, voluntary participation, and potential harm. This study followed ethical research principles through design and implementation.

Participation was entirely voluntary. Employees were invited through internal communication channels, with the research purpose, expected time commitment, data usage, and anonymity guarantees stated up front. Declining carried no penalty or consequence, and no incentives were offered that might have pressured anyone into taking part.

Anonymity and confidentiality were maintained strictly. No personally identifiable information was collected: no names, employee IDs, team affiliations, or any combination of demographic variables that could identify an individual respondent. Responses were stored securely with access limited to the researcher and thesis advisor. No individual response is reported anywhere; findings appear only in aggregate. The organization is anonymized in all publications to protect institutional privacy.

Informed consent came through the survey invitation, which set out the research purpose, the voluntary nature of participation, data handling procedures, and anonymity guarantees. Participants gave implicit consent by completing and submitting the survey, and could withdraw at any point by closing it without submitting.

Risk to participants was minimal. The questions covered professional work practices and perceptions rather than sensitive personal information, and no deception was involved. The obvious concern in workplace research is that employee responses might reach management or feed into performance evaluation; that risk was mitigated explicitly through the anonymity guarantees and through clear communication that managers would not have access.

Data security rested on Google Forms with access restricted to the researcher's university account, storage on password-protected devices, and adherence to data protection practice. On completion of the thesis, data will be retained according to university policy for research data management and verification, then securely deleted.

The research was conducted in accordance with Ankara University's research ethics policies and with organizational approval from relevant stakeholders. Formal Institutional Review Board approval was not required for low-risk workplace survey research of this kind, but the principles governing human subjects research, respect for persons, beneficence, and justice, were upheld throughout.

# 4. RESULTS AND DISCUSSION

## 4.1 Survey Response Overview

The survey went to approximately 100 software engineers across multiple product teams in November 2025. Fifty-four returned complete responses, a rate near 54%. Every respondent reported active use of AI-powered development tools. The final analytical sample is therefore 54 AI tool users (N=54).

Roles and experience levels vary across the sample in the way they would across any mid-size engineering organization. Figure 4.1 and Figure 4.2 present the distribution of respondents by role and experience level, respectively.

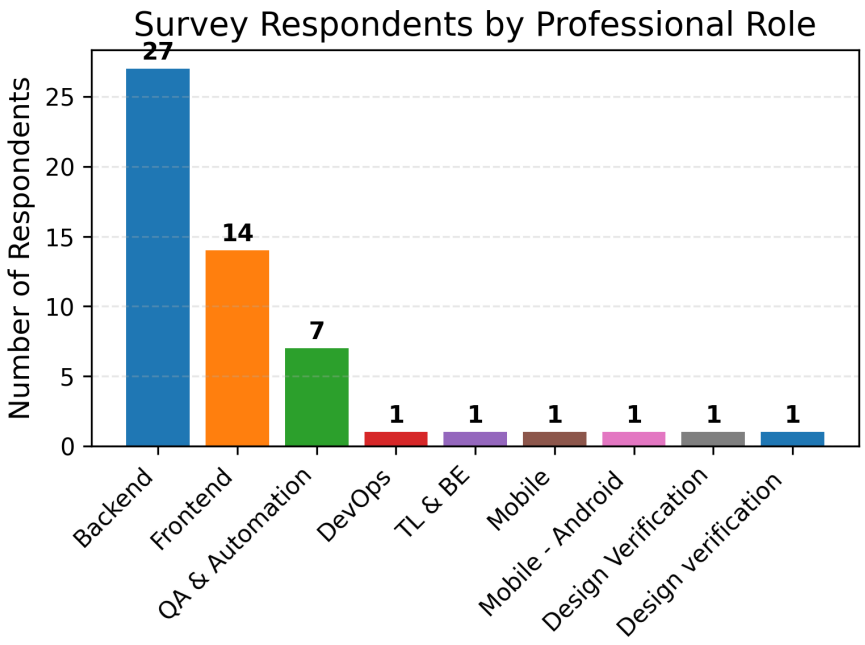


Figure 4.1 Distribution of respondents by professional role (N=54).

Three role groups carry the sample: Backend (n=33, 61.1%), Frontend (n=14, 25.9%), and QA & Automation (n=7, 13.0%). Smaller categories (DevOps, Mobile, Design Verification, Team Lead/Backend) were merged into Backend on shared technical focus, since analysis across nine sparse groups would have been meaningless. The shape is what product-focused software companies usually look like: backend absorbs the headcount needed for API design, business logic, and data management, while frontend concentrates on interface and experience.

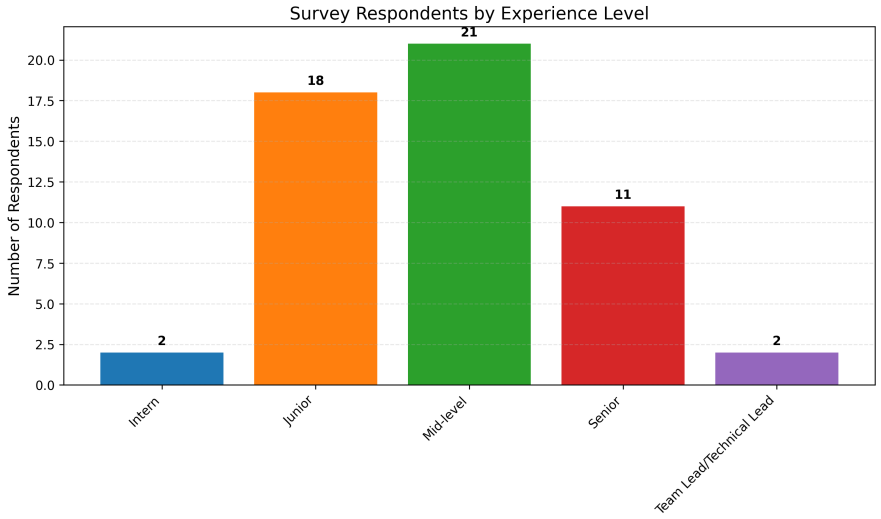


Figure 4.2 Distribution of respondents by experience level (N=54).

Experience level breaks down to 38.9% mid-level (n=21), 33.3% junior (n=18), and 20.4% senior (n=11), with interns and team leads at 3.7% each (n=2). This is the familiar pyramid, wide through the junior and mid ranks and narrowing sharply toward seniority and leadership.

Data cleaning covered response completeness, missing values, and categorical field consistency. Missing values were minimal on structured Likert questions and more common on open-ended items. Nothing in the responses suggested random or malicious entry.

## 4.2 AI Tool Adoption and Usage Patterns

Respondents reported using a wide range of AI-powered development tools, and the overlap between them is substantial: most developers keep several in rotation and pick according to the task at hand. Figure 4.3 presents the frequency of tool adoption across the sample.

ChatGPT has the highest adoption rate at 90.7% (n=49), followed closely by GitHub Copilot at 77.8% (n=42). Claude shows 46.3% adoption (n=25), OpenCode 44.4% (n=24), and Gemini 27.8% (n=15). Adoption drops off after that: DeepSeek 18.5%, Cursor 16.7%, Grok 7.4%, Codex 3.7%, with single mentions of Kiro, JetBrains AI Assistant, and Google Colab Assistant at 1.9% each.

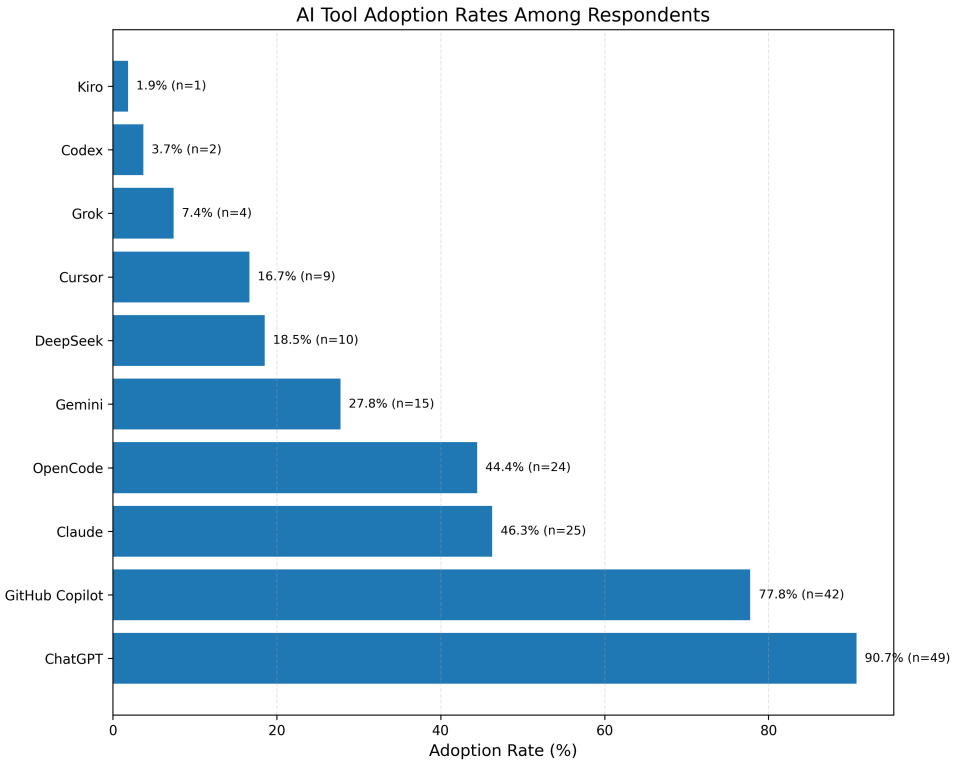


Figure 4.3 AI tool adoption rates across respondents (N=54, multi-select question).

That ChatGPT and GitHub Copilot lead is a consequence of their being used for different things. ChatGPT, reached through a web interface or an API, works as a general-purpose problem-solving assistant: debugging, explanation, learning an unfamiliar concept, generating a snippet outside the editor. Copilot lives inside the editor and completes code as it is written. Since 74.1% of respondents use both, developers evidently want tactical in-editor help and conversational problem-solving as separate things rather than one substituting for the other.

This reading rests on observation rather than on a survey item. Developers at this organization were seen using the chat tools for questions that were not about writing code at all, and for conceptual ones such as the difference between two libraries, while the in-editor tool handled code as it was typed.

Claude at 46.3% and OpenCode at 44.4%, both recent arrivals, show how quickly an alternative can take hold. Claude offers longer context windows and more controllable output, which suits developers working across large codebases. OpenCode is built around development workflows specifically. Its adoption here also had organizational help: management promoted it actively, without mandating it, at a point when the tool was drawing

attention generally. Nearly half the sample adopting a tool most organizations have never deployed reflects that push as much as the tool itself.

These tools are embedded deep in daily work. Figure 4.4 shows 63.0% of developers (n=34) using them daily, 29.6% (n=16) several times a week, and only 7.4% (n=4) once a week or less. Cumulatively that puts 92.6% of AI tool users in contact with them at least several times a week. AI assistance is routine rather than supplementary.

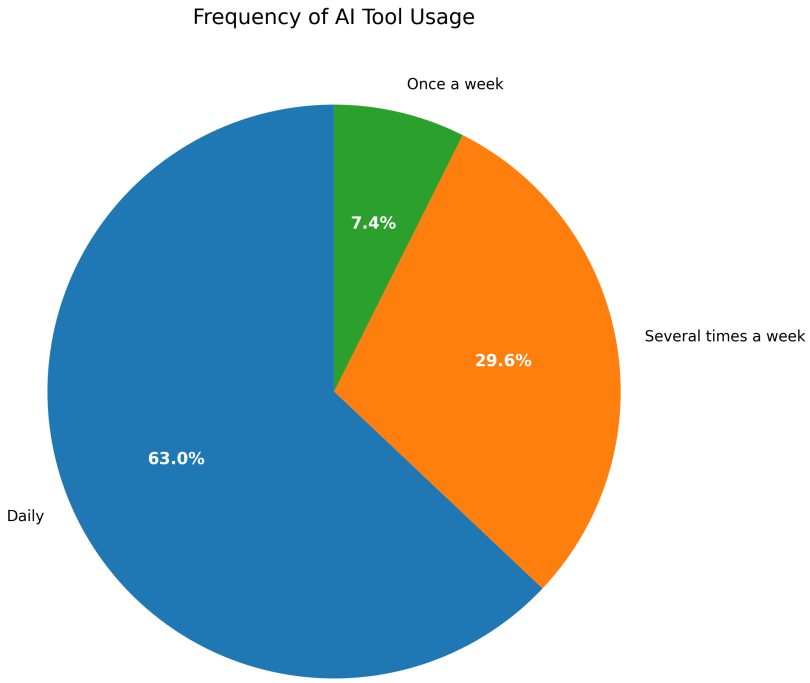


Figure 4.4 Frequency of AI tool usage among respondents (N=54).

License type, in Figure 4.5, comes out at 47.6% business/enterprise, 28.0% free tier, and 24.4% personal paid subscription, with percentages summing above 100% because the question allowed multiple selections. The share on enterprise licenses reflects what the organization pays for, subsidizing or fully covering access to premium tools. Free tiers at 28.0% tell a different story: some developers supplement what the company provides with personal accounts elsewhere, most likely to reach a specific feature or to get around rate limits on the company account.

Duration of use, in Figure 4.6, spreads fairly evenly across the adoption timeline: 25.9% (n=14) at 1–3 months, 20.4% (n=11) at 4–6 months, 22.2% (n=12) at 7–12 months, 25.9%

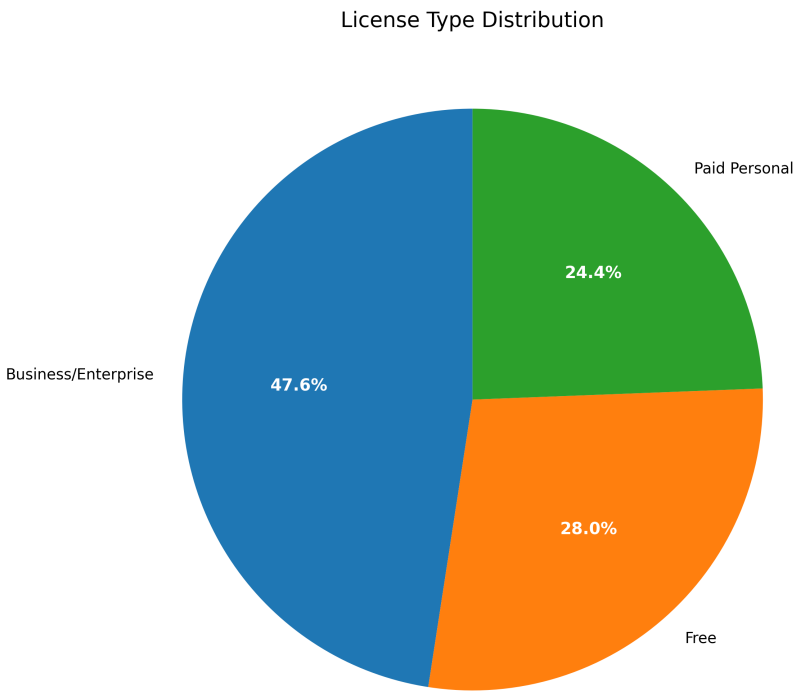


Figure 4.5 License type distribution among AI tool users (N=54, multi-select question).

(n=14) at 1–2 years, and 5.6% (n=3) beyond 2 years. Grouped, that is 46.3% under six months against 53.7% at six months or more. Newer and more established adopters are both well represented, which widens the range of perspectives and makes comparison across adoption stages possible.

Chi-square tests of independence asked whether usage patterns track role or experience level. Neither association reached significance: role against usage frequency gave $\chi^2$=11.06, df=16, p=0.8058, and experience level against usage frequency gave $\chi^2$=4.32, df=8, p=0.8270. License type against role behaved the same way ($\chi^2$=8.73, df=16, p=0.9242). One caveat carries real weight here. Degrees of freedom this high against a sample this small produce sparse contingency tables, several cells fall below an expected frequency of 5, and that violates the test's assumptions, so these p-values indicate association approximately rather than estimating probability precisely. Read with that caveat, the picture is of adoption spread evenly through the organization: backend, frontend, QA, junior, mid-level, and senior developers all use these tools at similar rates.

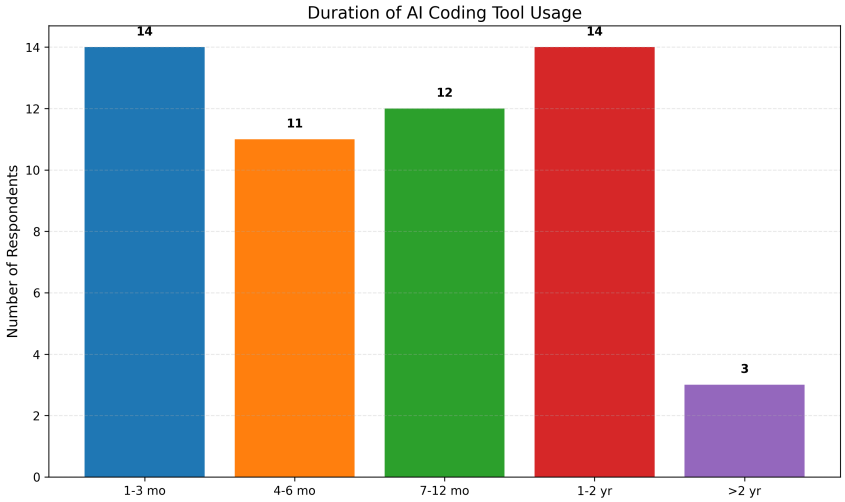


Figure 4.6 Duration of AI tool usage among respondents (N=54).

## 4.3 Perceived Productivity Impact

Respondents rated the impact of AI tools on several productivity dimensions using a 5-point Likert scale (1=Very Negative, 2=Negative, 3=Neutral, 4=Positive, 5=Very Positive). Every dimension came back positive, as Table 4.1 and Figure 4.7 show.

Table 4.1 Descriptive statistics for productivity impact metrics (N=54).

| **Productivity Metric** | **Mean** | **Median** | **Mode** | **SD** |
|---|---|---|---|---|
| Overall Productivity | 3.63 | 4.0 | 4 | 0.76 |
| Code Quality | 3.50 | 3.5 | 3 | 0.77 |
| Meeting Deadlines | 3.24 | 3.0 | 3 | 0.73 |
| Focus on Complex Tasks | 3.57 | 4.0 | 4 | 0.86 |
| Overall Work Experience | 3.50 | 4.0 | 4 | 0.86 |

Overall productivity carries the strongest perceived positive impact, with a mean rating of 3.63, median of 4.0, and mode of 4 (Positive). The typical developer, then, perceives a clear productivity benefit. Focus on complex tasks scores almost as well (mean=3.57, median=4.0, mode=4). That fits the idea that AI tools take over routine work and leave more attention for hard architectural and algorithmic problems.

Code quality and overall work experience both come in at 3.50 with medians of 3.5-4.0:

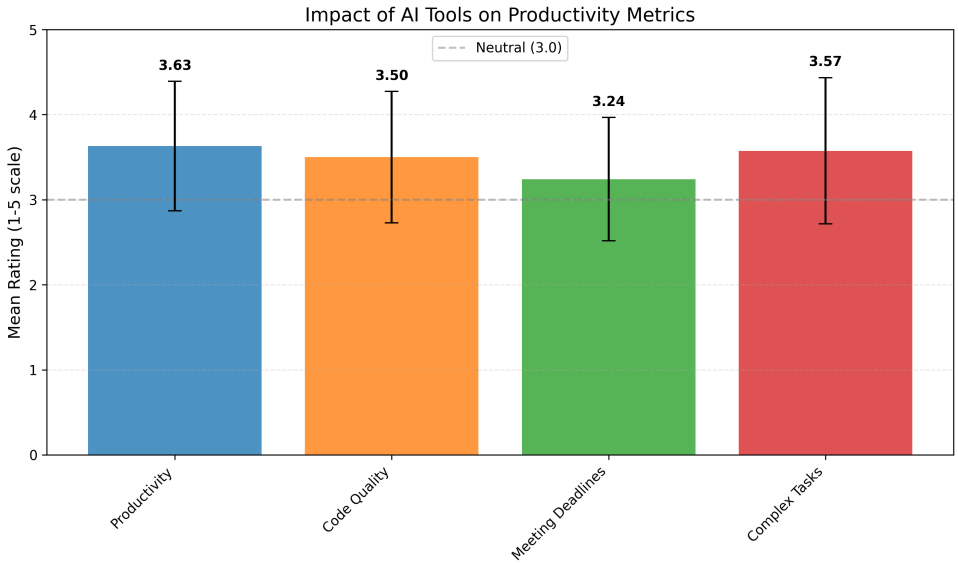


Figure 4.7 Distribution of productivity impact ratings across five dimensions (N=54).

moderately positive. Code quality has a neutral mode of 3, and that is where developers split. Some developers see quality improvements, perhaps from fewer syntax errors and exposure to better patterns; others see no change, perhaps because AI-generated code demands so much review and correction.

Meeting deadlines rates lowest at 3.24, though it stays above neutral. Several things could account for the weaker showing. Deadlines in an agile environment answer to much besides individual coding speed, including requirements clarification, review cycles, and dependencies on other teams. Developers may also not yet trust AI-accelerated code enough to compress a delivery timeline around it. And time saved writing the first draft can be partly given back in review, which blunts the effect on end-to-end schedules.

Standard deviations run from 0.73 to 0.86. The central tendency is positive while individual experience is not uniform: some developers report strong benefit at 4-5, others minimal or neutral effects at 2-3. That spread is what the cross-group analyses below set out to explain, by asking whether the differences follow role or experience level.

### 4.3.1 Productivity by experience level

Overall productivity ratings vary descriptively by experience level. Figure 4.8 illustrates these patterns.

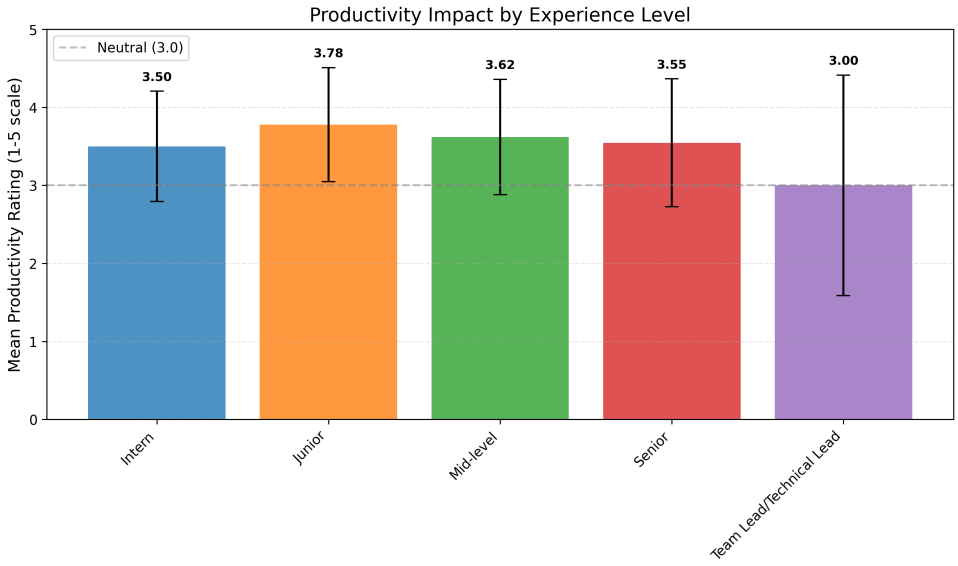


Figure 4.8 Overall productivity ratings by experience level (N=54).

Junior developers report the highest mean productivity rating at 3.78 (SD=0.73). One plausible reason is that AI tools cover knowledge gaps in APIs, syntax, and common patterns that juniors have not yet closed. Mid-level developers report a mean of 3.62 (SD=0.74), and senior developers 3.55 (SD=0.82), both in the moderately positive range. Interns show a mean of 3.50 (SD=0.71), though the small sample size (n=2) limits interpretability. Team Leads report the lowest mean at 3.00 (SD=1.41), with high standard deviation indicating divergent experiences within this small group (n=2).

A Kruskal-Wallis H-test tested whether those descriptive differences amount to statistically significant variation, being the non-parametric alternative to ANOVA appropriate for ordinal Likert data. It returned H=3.13 (df=4, p=0.5355) and failed to reject equal group medians. The descriptive trend toward higher junior ratings therefore does not clear the conventional $\alpha$=0.05 threshold. Power is the obvious complication: with N=54 overall and subgroups as small as n=2 for interns and team leads, a moderate effect could exist and go undetected.

AI tools may genuinely deliver similar benefit whatever a developer's experience level. Power may be insufficient, given N=54 and those n=2 subgroups. Or within-group variability may simply swamp whatever differences exist between groups. Cui et al. (2025) found larger gains for junior developers, which cuts against the first reading, though they measured objective delivery under randomized assignment rather than asking people what they perceived.

### 4.3.2 Productivity by professional role

Role produces the same picture: descriptive variation, no statistical significance. Figure 4.9 presents these patterns.

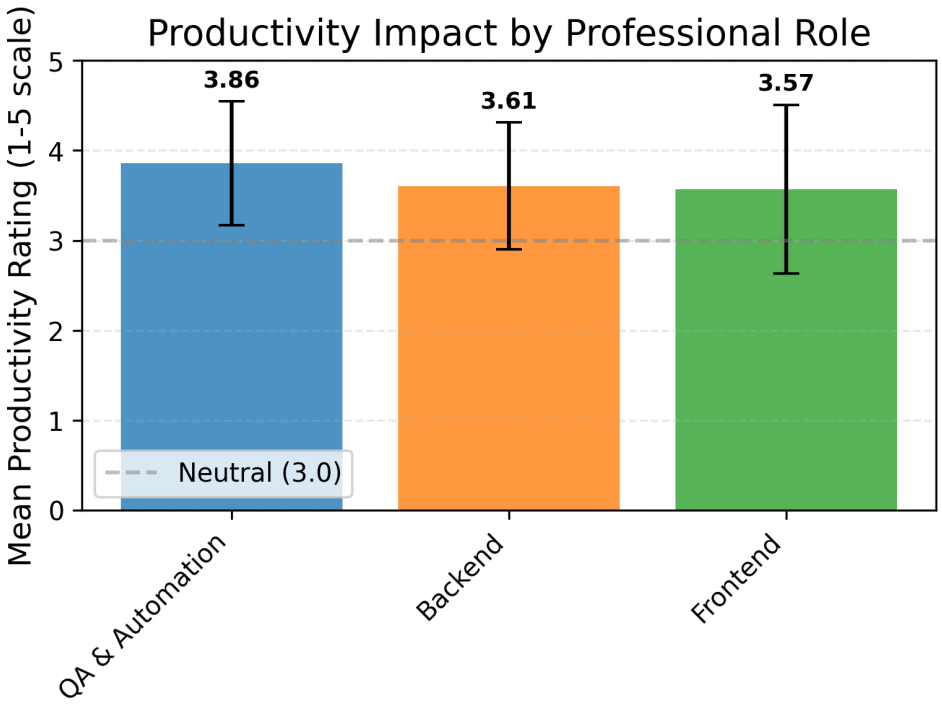


Figure 4.9 Overall productivity ratings by professional role (N=54 AI users). Error bars represent standard deviation.

QA & Automation engineers rate productivity highest at 3.86 (n=7, SD=0.69). What QA work consists of probably explains it, since repetitive test case writing, data setup, and validation scripting are precisely where a generated template saves the most time. Backend developers average 3.61 (n=33, SD=0.70) and Frontend 3.57 (n=14, SD=0.94), close enough to suggest that these assistants deliver comparable value across the two domains.

Kruskal-Wallis across the three consolidated categories returned H=0.43 (df=2, p=0.8060), so role produces no statistically significant difference either. That p-value sits well above the pre-consolidation result (H=7.92, df=8, p=0.4413 across nine original categories). The drop in degrees of freedom and the averaging of small groups into Backend account for the difference. As with experience level, the descriptive edge for QA is not statistically confirmed. It may be that AI tool value really is uniform across roles, benefiting developers whatever domain they work in.

### 4.3.3 Usage frequency and productivity correlation

Spearman rank correlation examined usage frequency against perceived productivity impact and found a statistically significant positive association ($\rho$=0.2823, p=0.0386), the only significant result among the productivity-related tests. Developers who reach for these tools more often report larger gains.

A coefficient of 0.28 is a weak-to-moderate association. It is not large, but p=0.0386 clears the threshold and counts as evidence against there being no relationship at all. Direction is another matter, and the cross-sectional design cannot supply it. Frequent use may build the prompting strategies that produce the gains; alternatively, developers who perceive more benefit may simply reach for the tools more. Both are plausible, and they are not mutually exclusive. If both operate, this is a reinforcing loop rather than a one-way effect.

For an organization trying to get a return on what it spends here, the practical reading is to encourage regular sustained use over sporadic trial. Developers who fold these assistants into a daily workflow report meaningfully more benefit than those who reach for them occasionally.

By conventional standards the effect is small. Cohen's (1988) guidelines put coefficients below 0.30 in the small range, 0.30-0.50 medium, and above 0.50 large. At $\rho$=0.28 this correlation falls just short of medium. Statistically significant or not, usage frequency accounts for roughly 8% of the variance in perceived productivity ($\rho^2$=0.08). The other 92% comes from somewhere else: individual differences in prompting skill, how tasks are allocated, prior experience, organizational context, none of which this correlation touches.

There is a second caution. This result came out of a battery of tests across multiple relationships, and running many comparisons raises the chance that at least one comes back positive by accident. No formal correction such as Bonferroni was applied, on the grounds that these tests address distinct research questions rather than repeatedly testing one hypothesis. Even so, a single significant finding at p=0.0386, that close to $\alpha$=0.05, deserves to be held loosely. Replication in a larger sample would settle it.

Experience level, treated as an ordinal variable (Intern=1, Junior=2, Mid=3, Senior=4, Team Lead=5), correlated weakly and negatively with productivity ratings, and not sig-

nificantly (ρ=-0.18, p=0.1812). Usage frequency against code quality ratings came out at ρ=0.26, p=0.0605, just the wrong side of the conventional threshold. A positive relationship may well be there; this sample cannot establish it.

## 4.4 Task-Specific Effectiveness

Overall productivity is an average across work that varies enormously, so the survey also asked about effectiveness on six specific activities: design and architecture, documentation, team collaboration, repetitive tasks, unit test generation, and bug diagnosis. Respondents rated each on a 5-point Likert scale (1=Not Effective to 5=Very Effective). Table 4.2 and Figure 4.10 present the descriptive statistics.

Table 4.2 Effectiveness ratings for specific development tasks (N=54).

| Development Activity | Mean | Median | Mode | SD |
|---|---|---|---|---|
| Repetitive Tasks | 3.80 | 4.0 | 4 | 0.96 |
| Unit Test Generation | 3.72 | 4.0 | 4 | 0.98 |
| Documentation | 3.56 | 4.0 | 4 | 0.98 |
| Design & Architecture | 3.07 | 3.0 | 3 | 1.01 |
| Bug Diagnosis | 3.06 | 3.0 | 3 | 1.05 |
| Team Collaboration | 2.70 | 3.0 | 3 | 1.02 |

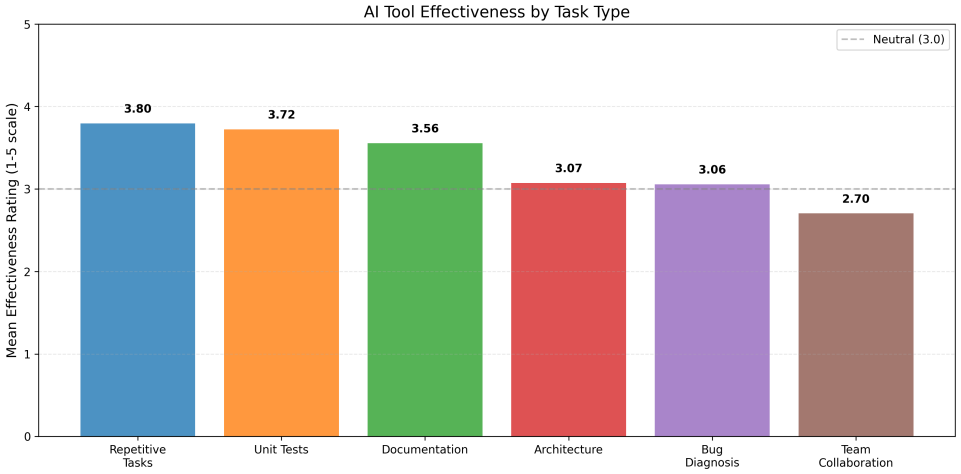


Figure 4.10 Effectiveness ratings across six development task types (N=54).

Repetitive tasks receive the highest effectiveness rating (mean=3.80, median=4.0,

mode=4), consistent with the view that AI tools work well on routine, pattern-based coding such as CRUD operations, boilerplate generation, API endpoint scaffolding, and data transformation logic. This matches what language models are actually good at: recognizing patterns and applying templates. Developers report that AI tools speed up these necessary but low-interest tasks, leaving more attention for the rest of the work.

Unit test generation follows closely at mean=3.72 (median=4.0, mode=4). Test writing is widely regarded as tedious, and a rating this high implies AI tools lower a real barrier to decent coverage. Qualitative responses added a caveat: AI-generated tests need careful review for meaningful assertions and edge cases, not just syntactic correctness.

Documentation sits at mean=3.56 (median=4.0, mode=4), moderately positive. These tools handle docstrings, README content, API documentation, and code comments competently. The moderate rather than high rating likely comes from a trade-off between speed and quality: AI produces initial documentation quickly, but it often misses the domain-specific detail and user perspective an experienced developer would supply.

Design and architecture land at a neutral-to-positive 3.07 (median=3.0, mode=3), with a standard deviation of 1.01 that marks how widely opinion splits. Architectural decisions run on organizational context, knowledge of legacy systems, and concern for maintainability years out, none of which these tools have. Some developers use AI architectural suggestions as brainstorming material; others find them superficial or simply wrong for the situation.

A senior backend developer gave a more specific account of why this fails at scale: “They don’t have one mentality. When we program we think about code from one perspective, our perspective. But an AI tool will combine perspective of all patterns it learnt from training data. Hence, for large tasks code produced is less coherent or event not at all.” Architectural work depends on holding one consistent set of commitments across a system, and a model averaging over every pattern in its training data has no such position to hold.

Bug diagnosis sits at much the same level (mean=3.06, median=3.0, mode=3, SD=1.05). AI can help by proposing candidate causes or writing debugging scaffolding, but complex debugging demands domain knowledge and system-specific understanding that limit how far the help goes. A syntax error or a common API misuse resolves quickly. A subtle logic bug, a race condition, or an architectural fault stays a human problem.

The obstacle is assembling the context in the first place. Diagnosing a real defect means tracing across the client, the feature configuration, the logs, and the data, and every one of those has to be located and handed to the model before it can say anything useful. Gathering that material is most of the work of debugging, and it is exactly the part these tools cannot do. Figure 4.11 shows the distribution of these ratings.

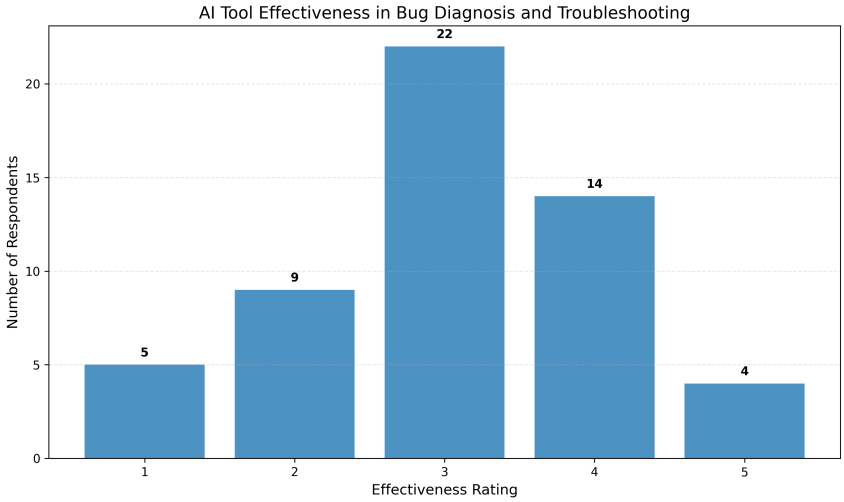


Figure 4.11 Distribution of bug diagnosis effectiveness ratings (N=54).

Team collaboration rates lowest (mean=2.70, median=3.0, mode=3) and is the only metric to fall below 3.0 once standard error is taken into account. Collaboration is interpersonal work, and these tools are not built for it. An AI assistant can help one developer write code or think through a problem; it does nothing for communication, shared understanding, or a decision several people have to reach together. Where AI use has displaced pair programming or thinned out discussion during review, some developers may read the effect as mildly negative.

### 4.4.1 Repetitive task effectiveness by role

Repetitive tasks rate highest overall, which raises the question of whether that benefit falls evenly across professional functions. Figure 4.12 segments the ratings by role.

Frontend developers rate it highest at 4.00. Their work is pattern-heavy: UI components, styling, and layout all take scaffolding well. QA & Automation follow at 3.71, matching workflows built around repetitive test case creation, data setup scripts, and validation logic. Backend comes in at 3.73. Ratings stay high across all three, so repetitive task

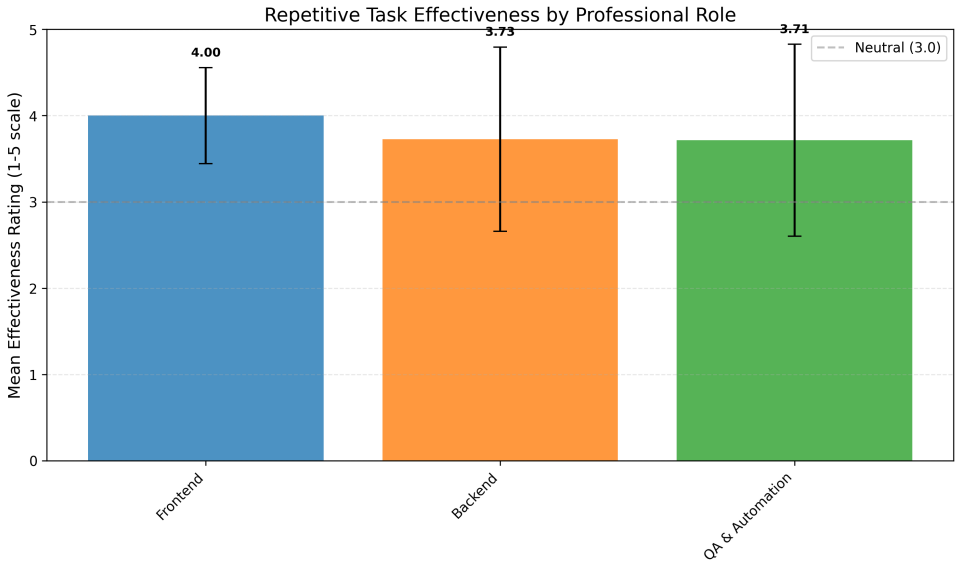


Figure 4.12 Repetitive task effectiveness ratings by professional role (N=54).

automation is a benefit every role gets, with the size of it depending on what repetitive work looks like in that particular job.

## 4.5 Code Quality and Review Practices

### 4.5.1 Code review habits

Respondents reported how often they review AI-generated code before integrating it. Figure 4.13 presents the distribution.

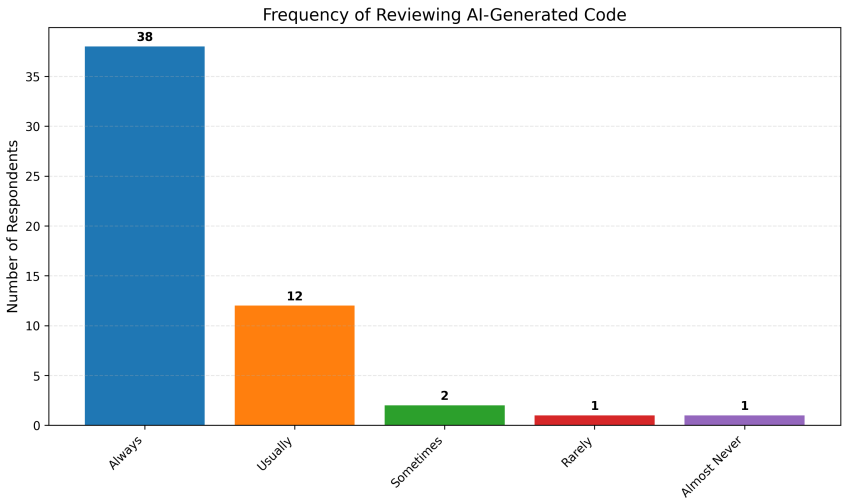


Figure 4.13 Frequency of code review for AI-generated output (N=54).

Verification is near-universal: 70.4% of developers (n=38) report that they always review AI-generated code before accepting it, 22.2% (n=12) usually review it, 3.7% (n=2) sometimes review it, 1.9% (n=1) rarely review it, and 1.9% (n=1) almost never review it.

Review here is policy as well as individual discipline. Every merge request and pull request in this organization requires at least one reviewer and one approval before it can merge. The 70.4% who always review AI-generated code are therefore working inside a process that already mandates review of all code, whatever produced it, which is worth holding in mind when comparing this figure against organizations with lighter requirements.

Reviewing AI code almost without exception is both professional responsibility and hard-won awareness of what these tools get wrong. Generated code is frequently syntactically clean and functionally plausible while still carrying a subtle error, a security hole, an inefficiency, or a pattern that clashes with project conventions. Always reviewing treats the tool as an assistant rather than an authority.

Reviewing at that rate has a cost, though. Time saved generating the first draft comes back partly as review overhead, and if the generated code needs heavy modification or careful reading before anyone can trust it, net productivity gains fall short of what raw generation speed would suggest. This trade-off between generation speed and review burden surfaced independently in the qualitative responses (Section 4.7).

Spearman correlation tested whether review frequency tracks perceived code quality impact and returned a weak positive association that missed significance ($\rho$=0.09, p=0.5177). Review frequency does not predict quality perceptions here, though the likely reason is mechanical. Nearly everyone reviews thoroughly, so there is too little variance in the independent variable for any relationship to show up.

### 4.5.2 First-try acceptance rates

Developers reported how often AI-generated code is acceptable on the first attempt without modification. Figure 4.14 illustrates the distribution.

The modal response is “sometimes” (53.7%, n=29), so most of the time AI-generated code

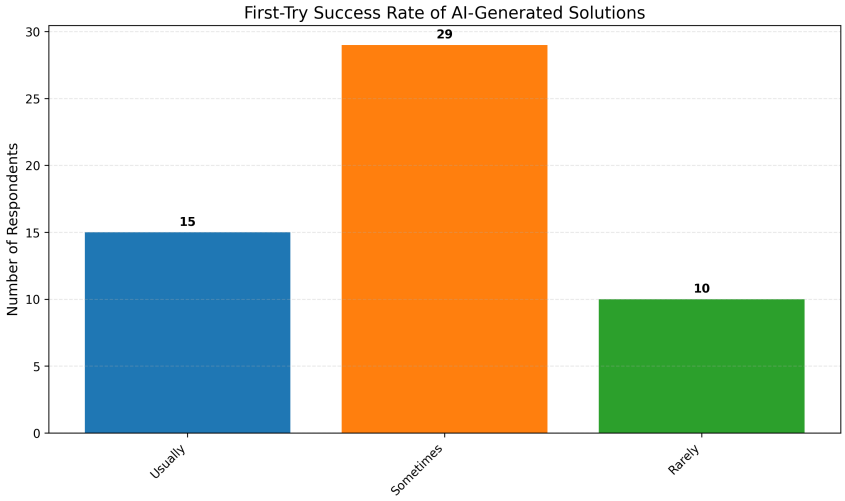


Figure 4.14 First-try acceptance rates of AI-generated code without modification (N=54).

needs modification before it can be integrated. “Usually” was selected by 27.8% (n=15): over a quarter of developers find AI output generally acceptable on the first try. For 18.5% (n=10) it is “rarely” acceptable; nearly a fifth of developers, then, almost always adjust what they are given.

These tools accelerate the first draft without removing the need to refine it. The reasons a modification becomes necessary are varied: wrong logic or algorithm, an inefficient implementation, an unhandled edge case, a misread requirement, a clash with project coding standards, a deprecated API. A modal response of “sometimes” places AI output squarely among useful starting points rather than production-ready code.

### 4.5.3 Retry frequency

When AI-generated code is unsatisfactory, developers may refine prompts and request regeneration. Figure 4.15 shows the distribution of retry attempts.

Most developers (59.3%, n=32) typically attempt 2-3 refinements and 24.1% (n=13) attempt 4-5, while 9.3% (n=5) go to 6-10, 3.7% (n=2) persist beyond 10, and 3.7% (n=2) abandon the attempt after one try. That puts 83.4% cumulatively inside the 2-5 range: developers will rework a prompt a few times to get usable code, and few keep going past that.

The retry pattern is the learning curve of prompt engineering made visible. Getting good

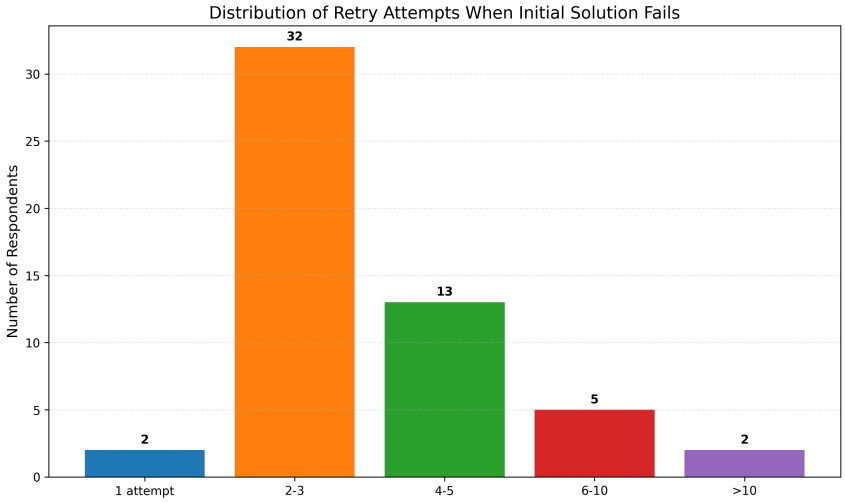


Figure 4.15 Number of prompt refinement attempts before obtaining acceptable code (N=54).

output means supplying enough context, stating requirements precisely, and steering the model toward the pattern you want. Practised users get there in fewer attempts; newer ones iterate more. Prompt engineering turned out to be the most frequently cited limitation in the qualitative responses (Section 4.7).

The survey asked for a single retry count, but the number varies with what is being asked. On a small, well-defined change one attempt is often enough. On a real feature the iteration divides in two: several rounds working the approach out with the model before any code is written, then a further two or three on the code itself. An aggregate retry figure averages those phases together and hides the distinction, which is a limitation of the instrument rather than of the tools.

The cumulative distribution also implies a working decision rule. If acceptable code has not appeared within 3-5 attempts, developers stop prompting and write it themselves. That threshold balances what AI generation might still save against the diminishing returns of another round of prompt iteration.

## 4.6 Cross-Group Analysis

### 4.6.1 Statistical tests summary

Three Kruskal-Wallis H-tests were conducted to examine group differences in productivity-related metrics. All tests failed to reject the null hypothesis of equal group medians at the conventional $\alpha$=0.05 significance level:

1. **Productivity × Experience Level:** H=3.13, df=4, p=0.5355. No significant difference in overall productivity ratings across the five experience levels (Intern, Junior, Mid-level, Senior, Team Lead).
2. **Productivity × Professional Role:** H=0.43, df=2, p=0.8060. No significant difference in overall productivity ratings across the three consolidated professional role categories (Backend, Frontend, QA & Automation).
3. **Code Quality × Experience Level:** H=3.97, df=4, p=0.4101. No significant difference in code quality impact ratings across experience levels.

These non-significant findings indicate that, despite descriptive variations in mean ratings (e.g., junior developers reporting higher productivity at 3.78 vs. senior at 3.55, or QA reporting 3.86 vs. Backend 3.61), the differences are not statistically reliable given sample size and within-group variability.

### 4.6.2 Interpretation of non-significant findings

The value may really be uniform, accruing much the same to junior and senior developers, to backend and frontend engineers, and to QA specialists. On that reading the proposition is universal: offloading routine work and speeding up common tasks helps everyone, whatever their context.

The second is that power limitations hide differences that exist. N=54 with subgroups from n=7 to n=33 leaves little ability to detect small-to-moderate effects. Kruskal-Wallis suits ordinal data but needs substantial samples to reach adequate power, especially against

within-group variability as high as this (standard deviations from 0.69 to 0.94 across role subgroups). A larger sample might well confirm the descriptive trends.

The third is that self-reported perception cannot see objective performance differences. Cui et al. (2025), using randomized field experiments and objective metrics, did detect experience-level differences, with less experienced developers completing relatively more tasks. Likert-scale perception carries measurement error and social desirability bias, either of which could flatten real variation. Developers at every level may feel they are benefiting even where the objective gains diverge sharply.

The fourth is that contextual factors moderate this relationship in ways a simple group comparison cannot resolve. Prompt engineering skill, willingness to iterate, trust or skepticism toward AI output, and how tasks get allocated within a team all vary between individuals, and that within-group heterogeneity can swamp anything between groups. Multilevel modeling or qualitative case work would separate these mechanisms better than group comparison does.

### 4.6.3 Practical implications

Non-significant results still leave usable descriptive patterns. Junior developers rating productivity above seniors (3.78 against 3.55) matches what theory predicts, since juniors gain knowledge augmentation and pattern exposure while seniors already hold the knowledge and get acceleration rather than transformation. An organization onboarding juniors could reasonably treat AI tool training as both a productivity measure and a learning aid.

The QA figure works the same way. High effectiveness on repetitive tasks (3.71) suggests these tools are worth more to roles with a heavier share of automatable pattern-based work, and QA workflows fit that description closely: test case generation, data setup, validation scripting. An organization looking for a quick return would sensibly start its rollout and training with QA and automation teams.

## 4.7 Professional Concerns and Challenges

### 4.7.1 Thematic analysis of benefits

An open-ended question asked: "Are there any other benefits from using AI tools that weren't mentioned in the survey?" A total of 18 developers provided responses (33.3% response rate). Inductive coding identified five primary themes, presented in Table 4.3.

Table 4.3 Thematic analysis of open-ended responses regarding AI tool benefits (n=18, 33.3% response rate).

| **Theme** | **n** | **%** | **Representative Quote** |
|---|---|---|---|
| Time Savings / Speed | 3 | 16.7% | "It helps you think faster, unblock yourself, and spark ideas you wouldn't have reached..." |
| Learning / Knowledge | 3 | 16.7% | "I guess one non-technical benefit is that, to get the most of the AI tool from my experience..." |
| Problem Solving | 3 | 16.7% | "It helps you think faster, unblock yourself, and spark ideas you wouldn't have reached..." |
| Boilerplate / Repetitive | 1 | 5.6% | "Benefits such as explanation of bad documentation. sometimes some websites explain..." |
| Unit Testing | 1 | 5.6% | "Writing unit tests, they help a lot in not neglecting unit tests, because they automate that process..." |

Three themes came back at equal frequency (16.7% each): *time savings and speed*, *learning and knowledge acquisition*, and *problem-solving assistance*. Time savings matches the productivity metrics and effectiveness ratings, with the value concentrated in routine

work. Learning is the more interesting one, since productivity-focused studies tend not to look for it: these tools work as on-demand tutors, explaining unfamiliar code, unpicking bad documentation, walking someone through an API or framework they have never touched. That role should matter most to junior developers, who face the steepest curve.

Problem-solving assistance describes AI tools working as thinking partners, helping developers past a mental block, offering an alternative approach, opening up a solution space. One developer put it as help to "spark ideas you wouldn't have reached," which is creative augmentation rather than automation.

The fullest account of this came from a frontend developer who described the tools as "a talking rubber duck." The argument was that feedback helps whether or not it is correct: good feedback confirms an idea, and bad feedback still forces the developer to articulate why it is wrong, and "that exercise of going through your own idea makes you understand it better, which in turn makes it stronger." The same respondent emphasised availability over quality: "ideas don't come to your head on schedule from 9 to 5," so having somewhere to put a thought at the moment it arrives is itself the benefit. On that account the value lies in the interaction rather than in the accuracy of what comes back, which is not something the effectiveness ratings in Section 4.4 are able to capture. That kind of cognitive support may be what lies behind the positive rating for focus on complex tasks (mean=3.57): clearing routine problems quickly leaves mental energy for the hard ones.

One benefit fell outside the coding frame entirely. A senior backend developer observed that getting good output requires explaining a task from the reader's perspective, and that the habit carries over: "when writing a task or giving it to someone else. You learn to put yourself in their place and explain it from their perspective. Hence, giving information they need." A frontend developer described the same skill from the difficulty side, arguing that "AI coding is all about being good with words, having good communication skills." Prompting practice appears to transfer into how developers specify work for other people, which is a form of professional development the productivity literature does not look for.

Boilerplate reduction (5.6%) and unit testing support (5.6%) appeared less often, both consistent with the structured responses. At a 33.3% response rate, the structured questions appear to have already captured most of what developers had to say about benefits, leaving the open-ended answers to add at the margin.

### 4.7.2 Thematic analysis of challenges

An open-ended question asked: "What are the main challenges or disadvantages you've experienced with AI coding tools?" A total of 27 developers provided responses (50.0% response rate), substantially higher than the benefits question. Inductive coding identified eight primary themes, presented in Table 4.4.

*Prompt engineering* dominates at 33.3% (n=9): learning to work with these tools, writing prompts that get what you asked for, and iterating when the first attempt does not. It connects straight to the retry distribution in Section 4.5.3, where 1-3 retries are typical. What developers are describing is a genuinely new skill, absent from traditional software engineering education and inconsistent across tools and contexts. As one respondent put it, "learning how to interact with AI tools effectively" is itself an investment of time, and it offsets some of the productivity gain, most sharply during early adoption.

*Limited context awareness* follows at 25.9% (n=7). These tools have no grasp of the codebase, the architectural decisions behind it, project conventions, business requirements, or the organizational constraints a human developer navigates without thinking. The gap produces suggestions that parse correctly and mean the wrong thing: a deprecated API, a violated coding standard, an existing utility function ignored, domain logic misread. One developer described the tool as something that "misses context" and returns solutions that look reasonable on their own and are wrong for this project. The limitation bites hardest on complex tasks, and architecture at 3.07 is exactly where context matters most.

*Hallucinations and incorrect code* come third at 18.5% (n=5). LLMs produce confident-sounding code that is simply wrong: functions that do not exist, incorrect API signatures, logic errors that compile cleanly and fail at runtime. In one developer's words, "the AI confidently produces incorrect, misleading, or completely fabricated code," and every instance of that adds verification work. This is why review is universal (Section 4.5.1). Confidence in the presentation carries no information about correctness, so nothing can be trusted unverified.

*Code review burden* at 11.1% (n=3) names the paradox directly: time saved generating comes back as time spent reviewing. "Even with perfect prompts, the code generated needs extensive review which could take more time" than writing it by hand, particularly when the reviewer has to understand generated logic thoroughly before certifying it. This

Table 4.4 Thematic analysis of open-ended responses regarding AI tool challenges (n=27, 50.0% response rate).

| **Theme** | **n** | **%** | **Representative Quote** |
|---|---|---|---|
| Prompt Engineering | 9 | 33.3% | "One of the main challenges is learning how to interact with AI tools effectively..." |
| Limited Context | 7 | 25.9% | "AI sometimes gets too confident and gives wrong or over-engineered solutions. It also misses context..." |
| Hallucinations / Incorrect Code | 5 | 18.5% | "The biggest issue is hallucinations, where the AI confidently produces incorrect, misleading, or completely fabricated code..." |
| Code Review Burden | 3 | 11.1% | "Even with perfect prompts, the code generated needs extensive review which could take more time..." |
| Dependency / Loss of Skills | 3 | 11.1% | "Too dependents on it..." |
| Over-engineering | 2 | 7.4% | "AI sometimes gets too confident and gives wrong or over-engineered solutions..." |
| Debugging Difficulty | 2 | 7.4% | "My challenges would be to actually take time to review what was written or output by the AI..." |
| Outdated Patterns | 1 | 3.7% | "They sometimes produce subtly flawed code. They reinforce bad patterns or outdated methods..." |

may well be what sits behind the modest deadline rating of 3.24, since end-to-end delivery cannot fall in proportion to generation speed if the review phase expands to meet it.

*Dependency and skill atrophy concerns*, also 11.1% (n=3), look further out. Developers worry about becoming "too dependent," losing the ability to solve problems unaided, letting fundamentals decay. The worry sharpens around junior developers, who can use AI to skip precisely the experiences that build competence: struggling with a problem, reading the documentation, working out what an error message means.

Three challenges appeared less often. *Over-engineering* (7.4%) covers solutions more elaborate than the problem required. *Debugging difficulty* (7.4%) covers the extra effort of fixing code written in patterns you did not choose. That is harder than debugging your own. *Outdated patterns* (3.7%) covers deprecated methods and superseded practice arriving via training data full of legacy code.

Challenges drew a 50% response rate against 33% for benefits. Feedback surveys show this asymmetry routinely: people write more readily about what frustrates them. Length may also play a part: benefits compress into a word or two, speed or productivity, while a challenge takes a paragraph to explain.

## 4.8 ROI and Organizational Impact

Organizations evaluating these tools have to answer a question individual developers do not: whether the spending returns anything. What follows draws the survey findings relevant to that decision together.

### 4.8.1 Enterprise license adoption

As noted in Section 4.2, 47.6% of developers use business/enterprise licenses. The organization is paying for that access. Assuming typical enterprise pricing (e.g., GitHub Copilot Business at approximately $19-$39 per user per month, ChatGPT Team at $25-$30 per user per month), annual costs for a 54-developer team range from $12,000 to $25,000+ depending on tool selection and negotiated rates.

The organization did not constrain that spending during the adoption period. Developers

were given unlimited usage and unlimited budget for AI tools, so cost never entered into how much anyone used them. This bears on how far the findings travel: the usage patterns reported here come from an environment where nobody had to ration requests or justify a subscription, and organizations that meter access should not expect the same frequency of use.

Positive productivity ratings (overall mean=3.63, repetitive tasks mean=3.80) alongside 63% daily usage indicate that developers value what the organization is paying for. Turning that into a financial return is a further step, requiring objective metrics such as throughput, cycle time, and feature delivery rates, and requiring the offsetting costs to be counted too: review burden, the prompt engineering learning curve, and the overhead of integrating another tool.

### 4.8.2 Perceived value and continued use intention

The survey asked nothing directly about satisfaction or intention to continue, but 92.6% using these tools at least several times a week, combined with positive productivity ratings, is a strong revealed preference. Developers vote with their behaviour, and sustained frequent use means the perceived value clears the friction of adoption. The correlation between usage frequency and productivity ($\rho$=0.28, p=0.0386) points the same way: the developers who find these tools valuable are the ones who reach for them.

### 4.8.3 Areas for improvement

The challenge themes point to priorities on both sides. Organizations should train developers to work with these tools rather than assume the skill arrives with the license, and workshops, a documented set of effective prompt patterns, and peer mentorship are all plausible vehicles for that. Vendors have the harder problem: closing the context gap that produces inappropriate suggestions. That means feeding the tool project-specific material: coding standards, architectural patterns, existing utility libraries, domain knowledge.

Tools should also signal their own uncertainty. A suggestion resting on a strong pattern match and one that is largely speculative currently arrive looking identical. Calibrated trust is impossible under those conditions. On the workflow side, development environments could carry AI-specific review checklists or automated testing so that verification gets

faster without getting weaker. And a "learning mode" aimed at junior developers, one that explains generated code instead of simply offering it, would address the skill atrophy concern at its source.

## 4.9 Discussion

The preceding sections reported the empirical findings with minimal interpretation. What follows reads them against the existing literature, answers the four research questions, and works out what they imply in theory and in practice.

### 4.9.1 Addressing research questions

**RQ1: How do AI-powered development agents impact software developer productivity?**

Developers perceive a positive but moderate impact. Overall productivity averaged 3.63 on a 5-point scale with a median of 4.0 (Positive). The typical developer reports a meaningful benefit. The significant correlation between usage frequency and productivity ($\rho$=0.28, p=0.0386) fits the reading that deeper engagement goes with higher perceived productivity.

The magnitude sits well below what vendors claim and what the early studies found. Peng et al. (2023) reported 55% faster task completion under controlled conditions, an acceleration nothing here approaches; 3.63/5 falls roughly midway between neutral and very positive. Peng et al.'s task was a standard HTTP server. A typical feature in this organization touches around three microservices, plus the frontend and mobile repositories, so roughly five repositories in all. A developer picking up that feature has to understand the domain and the business logic behind it before writing anything. AI tools do not supply that understanding, and they hallucinate, so what they do supply has to be checked. The volume of generated code then creates a second problem: when there is a lot of code to review, developers lag, review less carefully, and business logic errors or missed requirements get through, in code that was claimed to be generated correctly. A respondent described the mechanism directly, reporting that "the generated text can contain a lot of fluff so I often skip reading everything." Review does not fail by being skipped; it fails by being skimmed, and volume is what causes the skimming.

The effectiveness ratings qualify that further: AI tools do well on repetitive tasks (3.80) and unit test generation (3.72), show limited value for team collaboration (2.70), and land in the middle for architecture (3.07) and debugging (3.06). Productivity impact is therefore task-dependent. These tools speed up automatable work and do much less for creative, contextual, or interpersonal activities.

Meeting deadlines at 3.24 suggests those gains do not carry all the way through to delivery speed, held up somewhere by review overhead, integration problems, or organizational constraints such as fixed sprint durations and dependencies on other teams. Individual productivity and organizational throughput are separate quantities. Generating code faster is not the same as delivering features faster, and the second answers to a great deal beyond how quickly any one developer writes.

**RQ2: How does the impact vary across developer roles and experience levels?**

Neither experience level ($H=3.13$, $df=4$, $p=0.5355$) nor professional role ($H=0.43$, $df=2$, $p=0.8060$) produced a significant difference in productivity ratings. That runs against prior work. Cui et al. (2025) found larger relative gains for less experienced developers under randomized field experiment conditions, and the reasoning is intuitive: juniors should gain from knowledge augmentation while seniors gain less, since AI accelerates expertise they already hold.

The most likely explanation is measurement. This study asked about *perceived* productivity on self-report Likert scales; Cui et al. counted *objective* task completions under randomized assignment. Those are different quantities, and there is no reason they must agree. A junior developer with low baseline expectations may underestimate how much they gained, while a senior developer sensitive to any improvement in an already efficient workflow may overestimate. Social desirability compresses the ratings further, pushing every group toward moderately positive regardless of what varied underneath.

Task allocation may also mask a real difference. Juniors tend to get the repetitive work where AI performs best and seniors the architectural work where it performs worst, and that could balance the benefit out by accident. And $N=54$ cannot reliably detect a modest effect, so the gap between juniors at 3.78 and seniors at 3.55 may be real and buried in noise.

Statistical non-significance does not empty the descriptive patterns of practical use. QA engineers rate repetitive task effectiveness highest at 3.71. Their workflow is built on test case generation, data setup, and validation scripting, so these tools should be worth most to QA and automation roles. An organization chasing a fast return would target those roles for deployment and training first.

**RQ3: What are developers' perceptions of AI tool effectiveness across different task types?**

The effectiveness ratings separate task categories cleanly into high, moderate, and low value. Repetitive tasks (3.80) and unit test generation (3.72) rate strongest, confirming that pattern-based, automatable activities match what these models do well. Documentation (3.56) sits in the middle, trading speed against quality. Architecture (3.07), bug diagnosis (3.06), and above all team collaboration (2.70) rate lowest. These tools handle contextual reasoning, deep debugging, and interpersonal interaction poorly.

The pattern supports a complementarity model of human-AI collaboration. Brynjolfsson and McAfee (2017) observe that machine learning systems hardly ever replace an entire job and most often complement human activity in ways that raise its value. The ratings here describe exactly that: productivity peaks where humans and AI each work to their strengths, the machine on automatable pattern-based work and the human on what is creative, contextual, or collaborative. Delegating routine work frees cognitive resources for the rest. That is presumably why focus on complex tasks rates positively at 3.57.

Recent research on task suitability points the same way. Ziegler et al. (2024) found Copilot users spending less time searching for information and holding onto flow more easily, consistent with fewer interruptions for routine lookups. Bakal et al. (2025), across more than 400 developers at ZoomInfo, report a 33% suggestion acceptance rate against 72% developer satisfaction, so developers take a substantial minority of what the tool offers and still rate the experience well. The effectiveness ratings here extend those observations into a mid-size company.

Team collaboration at 2.70 deserves more attention than its unsurprising face value. Nobody built these tools for interpersonal work, so the low rating is expected; the consequences are not. If AI shifts development toward individual coding and away from pair programming, collaborative design, and substantive review discussion, then code quality

and knowledge sharing could erode over time even as individual productivity climbs. That is worth monitoring deliberately, because it would not show up in any of the measures an organization normally watches.

One mechanism appeared in this organization specifically. When a team hands a feature to another team, the handover documentation is now often AI-generated, and that documentation is frequently outdated or describes requirements and features that do not exist. The receiving team works from a document that reads authoritatively and is wrong.

**RQ4: What challenges, concerns, and limitations do developers face when using AI tools?**

Thematic analysis put prompt engineering (33.3%), limited context awareness (25.9%), and hallucinations (18.5%) at the top. The first of these belongs to a broader AI literacy problem, in that using these tools well takes a skill software engineering education has never taught. The learning curve shows up in the retry distribution, where 1-3 iterations is typical, and every one of those iterations is friction that gives back part of the productivity gain, most of all for someone who has just started.

The tooling built to manage prompting becomes its own maintenance problem. Developers write skills, workflows, and harnesses to stop the model producing over-complex output, and those artifacts then have to be maintained and remembered. A large workflow generates a large volume of code, and it becomes hard to follow what is actually happening. The prompting burden does not disappear, it moves.

Context awareness at 25.9% is a basic gap between what these tools can currently do and what developers need. A model trained on public repositories knows nothing about this project's conventions, the architectural decisions behind them, the utilities already written, or the domain the software serves. That gap forces heavy review and modification. It is part of why only 27.8% of developers get first-try acceptance "usually" while 53.7% get it only "sometimes." Closing it takes either real advances in context handling, retrieval-augmented generation reaching project documentation and codebase indices, or a workflow change in which developers spend effort building rich context into their prompts.

Hallucinations at 18.5% are a limitation that has not gone away: the model produces incor-

rect code confidently, and it looks plausible until it fails functionally or semantically. This is what makes review universal at 70.4% and what caps trust in the output. A traditional tool given valid input returns correct output predictably; an LLM is probabilistic and fails without warning. The only workable posture is vigilance, treating every suggestion as a draft to verify. That sits against Qian and Wexler's (2024) finding, where participants leaned on the assistant more heavily as a task went on even as their reported trust fell, which they read as automation complacency. This sample looks different, having settled into steady skepticism paired with verification.

Dependency and skill atrophy concerns at 11.1% look past the current tools to the workforce. Will juniors who lean hard on AI fail to build deep problem-solving ability, opening a generation gap in fundamental competence? Will experienced developers lose skills they no longer exercise? Both are speculative, and both would only be answered by longitudinal work comparing skill development in AI-assisted and unassisted cohorts.

### 4.9.2 Comparison with existing research

These findings converge with the existing literature in some places and part from it in others. Direction of effect is where they agree: moderate positive productivity perceptions here are consistent with the aggregate gains Cui et al. (2025) measured in randomized field experiments, even though the two studies measure through entirely different instruments. The task-effectiveness gradient, high on repetitive work at 3.80 and lower on complex problem-solving, has no direct counterpart in that literature, which reports aggregate effects without breaking them down by task. The challenge themes echo earlier work: Vaithilingam et al. (2022) on how hard generated code is to understand and debug, Pandey et al. (2024) on Copilot's trouble with large functions, multi-file changes, and proprietary contexts.

Divergence shows up in magnitude and in experience-level variation. Peng et al.'s (2023) 55% speed improvement is a different order of thing from a perceived 3.63/5, and the controlled task against the real workplace is the likeliest explanation. The absence of experience-level differences parts from Cui et al. (2025) and their larger junior developer gains. The divergence could be measurement, objective performance against subjective perception, or something about mid-size companies that does not hold at large technology firms.

Adoption rates here (91% ChatGPT, 78% Copilot) and usage frequency (63% daily) line up with industry-wide figures, since Stack Overflow's 2025 Developer Survey found 84% adoption and 51% daily use among professional developers, though the integration in this sample runs deeper than several earlier studies observed. The move from experimental adoption to routine workflow component has happened fast.

Three recent studies put all of this in sharper context. The METR randomized controlled trial (Becker et al. 2025) found AI tools slowing experienced open-source developers by 19% on real tasks. That is a direct counterexample to the literature's generally positive direction. Stray et al. (2026) found flat objective metrics sitting alongside positive perceptions across a longitudinal study at a government IT agency, a pattern this study's moderate perceived gains resemble. Tomaz et al. (2026) applied the SPACE framework to telemetry and survey data from three agile teams over roughly 13 months and found Performance and perceived Efficiency rising sharply while developer Activity stayed flat. On their reading AI raises the value density of development work rather than its volume. Between them these studies establish that effectiveness depends heavily on context, and that a perceived benefit does not reliably become a measurable one.

### 4.9.3 Theoretical implications

Work on cognitive bias in AI-assisted development connects here directly. Zhou et al. (2026) identify automation bias and anchoring as systematic risks when developers work with generated code. That may be why review stays universal at 70.4% despite trust perceptions being positive. Trusting but verifying looks less like caution than like a rational response to a well-documented tendency to produce confidently wrong output.

The findings also bear on how work should divide between developers and their tools. The effectiveness ratings support the hypothesis that AI performs on pattern-based automatable tasks while humans perform on contextual, creative, and collaborative ones. That is a division of labour rather than a replacement. Productivity comes from each side taking what it handles better.

### 4.9.4 Practical implications

Organizations considering AI tool adoption should calibrate expectations to moderate gains, an extrapolated 10–30% time savings on routine tasks based on respondent self-reports, not to transformation. The first deployments belong in roles with a high proportion of repetitive work, QA and backend implementation, where the value is clearest. Prompt engineering training shortens the learning curve, and code review standards should hold rather than relax. One thing worth watching that organizations rarely think to watch: whether AI adoption quietly erodes collaborative practice.

For AI tool vendors, the challenge themes set an agenda. Context awareness needs integration with project documentation and codebase analysis. Uncertainty calibration needs the tool to communicate how confident it actually is. Prompt assistance features would flatten the learning curve that developers named their single biggest obstacle. And the tools are designed around individual coding. Collaborative workflows go unserved.

For developers, the findings support a "trust but verify" posture: use these tools heavily on routine tasks, save the freed attention for hard problems, and review what comes back. Prompting rewards deliberate practice. The vigilance about skill maintenance matters most for junior developers, who can lean on AI hard enough to skip the struggle that builds problem-solving ability in the first place.

## 4.10 Supplementary JIRA Metrics Analysis

Team-level productivity metrics were extracted from the organization's JIRA system for five anonymized teams (Team Alpha through Team Epsilon), to set behavioral data against what the survey reported. The analysis compares throughput, cycle time, and bug ratio across a seven-month pre-AI baseline (January through July 2025) and a seven-month post-AI period (August 2025 through February 2026). Causality is not the aim, and too many confounders sit in the way for any causal claim to survive. The question is narrower: whether objective team-level patterns align with, complicate, or contextualize the moderate perceived gains survey respondents reported (overall mean=3.63/5).

Extraction produced 5,169 issues across the five teams, 2,215 in the pre-AI period and 2,954 in the post-AI period. Counts varied considerably by team: Team Alpha 1,243

issues (485 pre, 758 post), Team Gamma 1,180 (504 pre, 676 post), Team Beta 966 (496 pre, 470 post), Team Delta 943 (475 pre, 468 post), Team Epsilon 837 (255 pre, 582 post). Most teams recorded more issues after adoption. That may reflect organic project growth, seasonal workload, or expanded capacity as much as anything AI did.

### 4.10.1 Throughput

Average monthly throughput, the number of issues resolved per month, increased during the post-AI period for four of the five teams. Table 4.5 and Figure 4.16 present per-team throughput and cycle time comparisons.

Table 4.5 Average monthly throughput and median cycle time by team across pre-AI and post-AI periods.

| **Team** | **Avg. Monthly Throughput** | | | **Median Cycle Time (Days)** | | |
|---|---|---|---|---|---|---|
| | Pre-AI | Post-AI | Change | Pre-AI | Post-AI | Change |
| Alpha | 56.1 | 71.3 | +27.1% | 34.9 | 26.1 | -25.2% |
| Beta | 39.0 | 41.7 | +6.9% | 26.9 | 26.6 | -1.1% |
| Gamma | 55.6 | 63.4 | +14.0% | 21.0 | 20.8 | -1.0% |
| Delta | 49.7 | 47.9 | -3.6% | 22.8 | 13.2 | -42.1% |
| Epsilon | 34.4 | 50.7 | +47.4% | 33.8 | 14.0 | -58.6% |

Team Epsilon gained the most (+47.4%, from 34.4 to 50.7 issues per month), then Team Alpha (+27.1%), Team Gamma (+14.0%), and Team Beta (+6.9%). Team Delta alone declined slightly (-3.6%). Cross-team average throughput rose 17.1%, from 47.0 to 55.0 issues per month. These increases coincide with the post-AI period without belonging to it: team composition changed, projects matured, business priorities shifted, and seasonal workload moved, any of which could account for part of the difference.

### 4.10.2 Cycle time

Median cycle time, calendar days from issue creation to resolution, decreased for all five teams during the post-AI period, as shown in Table 4.5 and Figure 4.17.

Team Epsilon again moved furthest (-58.6%, from 33.8 to 14.0 days), then Team Delta (-

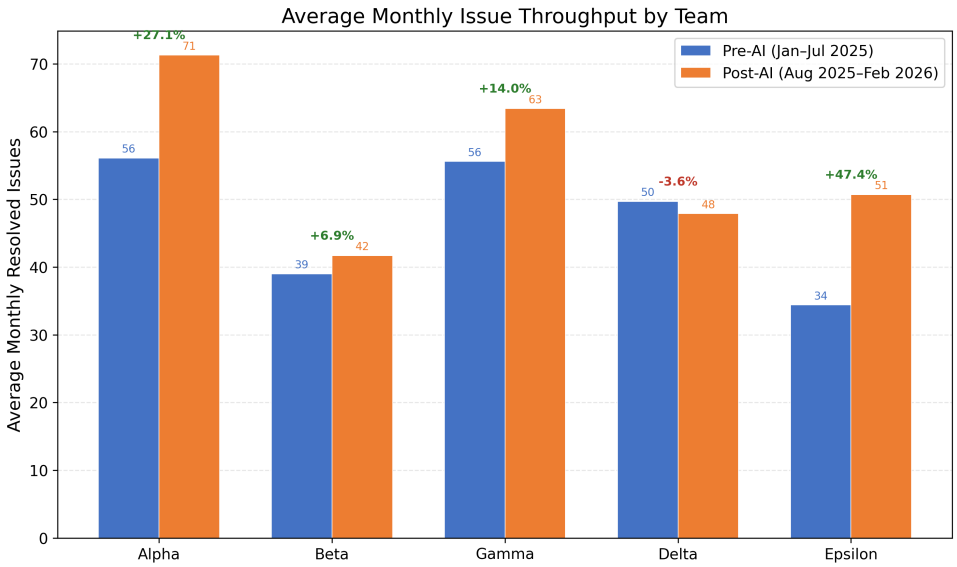


Figure 4.16 Average monthly throughput (issues resolved per month) by team, comparing pre-AI (January–July 2025) and post-AI (August 2025–February 2026) periods.

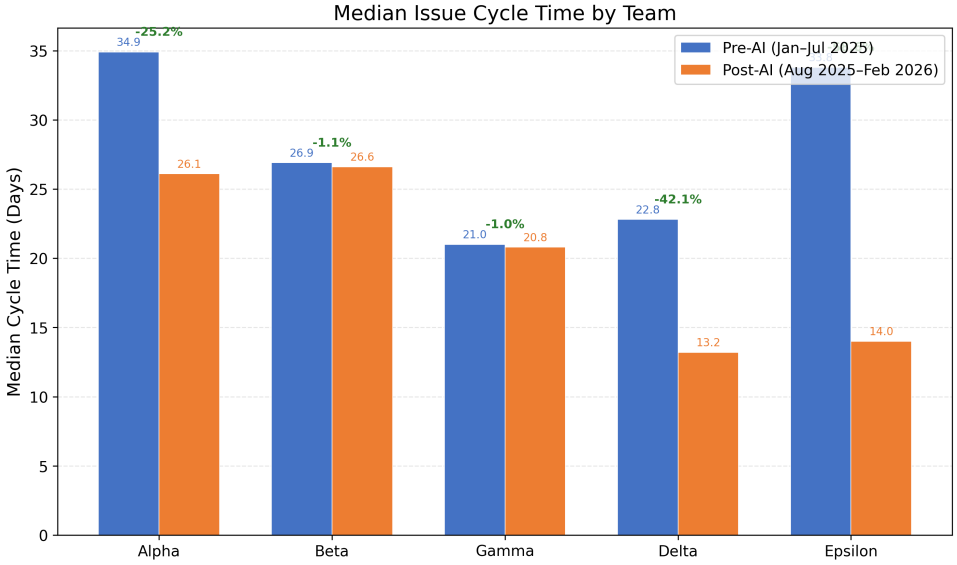


Figure 4.17 Median cycle time (days from issue creation to resolution) by team, comparing pre-AI and post-AI periods.

42.1%, from 22.8 to 13.2 days) and Team Alpha (-25.2%, from 34.9 to 26.1 days). Teams Beta and Gamma barely moved at -1.1% and -1.0%. Cross-team average median cycle time fell 27.8%, from 27.9 to 20.1 days. Every team moved the same way, but the magnitudes are not comparable. The teams starting from long cycle times, Epsilon, Delta, and Alpha, reduced them proportionally the most. Workflow improvement, bottlenecks clearing, or anything else that happened during the adoption window could account for that. Gamma and Beta were already efficient and had little left to gain.

### 4.10.3 Bug ratio and story point velocity

Bug ratio, the proportion of bug and production bug issues relative to total issues, showed mixed patterns across teams. Table 4.6 presents bug ratio and story point velocity data.

Table 4.6 Bug ratio and story point velocity by team.

| **Team** | **Bug Ratio (%)** | | | **SP Velocity (Monthly)*** | | |
|---|---|---|---|---|---|---|
| | Pre-AI | Post-AI | Change | Pre-AI | Post-AI | Change |
| Alpha | 16.9 | 17.8 | +0.9pp | — | — | — |
| Beta | 30.2 | 28.1 | -2.1pp | — | — | — |
| Gamma | 31.9 | 21.3 | -10.6pp | — | — | — |
| Delta | 13.5 | 8.1 | -5.4pp | — | — | — |
| Epsilon | 14.1 | 14.1 | +0.0pp | — | — | — |

* Reported only for teams with $\geq$70% story point coverage in both periods.

Bug ratios fell for three teams. Team Gamma dropped furthest at -10.6 percentage points, from 31.9% to 21.3%, with Team Delta down 5.4 points and Team Beta down 2.1. Team Alpha edged up 0.9 points and Team Epsilon held steady. The cross-team average fell 3.4 percentage points, from 21.3% to 17.9%. Defect proportions did not rise after adoption. That is the relevant answer to the worry that AI-assisted development would introduce more bugs.

Story point velocity was planned as a fourth metric and could not be reported. No team reached the 70% story point coverage threshold in both periods, because estimation practice varied too much between teams and across time. That leaves the supplementary analysis less complete than intended.

### 4.10.4 Synthesis with survey findings

The JIRA metrics give a second angle on the survey findings. Throughput up 17.1% across teams and cycle time down 27.8% point the same direction as the moderate positive perceptions respondents reported (mean=3.63/5). Reading objective metrics against subjective ones still calls for caution. Stray et al. (2026) watched flat objective metrics sit next to positive perceptions across a longitudinal study at a government IT agency. The two do not track each other reliably. Tomaz et al. (2026), following three agile teams for roughly 13 months with the SPACE framework, found Performance and perceived Efficiency rising sharply while developer Activity stayed flat, so gains can appear in outcome measures without any rise in raw activity at all.

These findings land between those two precedents. Objective metrics moved positively and broadly in line with survey perceptions, while the size and consistency of that movement varied a great deal by team. Team Delta is the instructive case: throughput fell 3.6% while cycle time improved 42.1%. Productivity dimensions can move in opposite directions inside one team at one time. That may be the complexity behind meeting deadlines rating lowest at 3.24/5. How fast an individual task closes, which cycle time captures, is not how fast a feature ships, which throughput captures, and the second answers to organizational factors the first never touches.

The small number of teams (N=5) rules out inferential statistics, leaving descriptive patterns rather than validated effects. Confounders are everywhere: team composition changes, project maturity, shifting business priorities, seasonal workload, process improvements with nothing to do with AI, market conditions. The pre-post design cannot establish causality, and AI adoption coinciding with metric change is not AI adoption causing it. And aggregating at team level conceals within-team variation that might carry a finer pattern. What the analysis does supply, despite all that, is objective behavioral data, which offsets part of the study's dependence on self-report.

### 4.10.5 Limitations

Several limitations bound what these findings support. The single-company design limits external validity, so nothing here should be assumed to hold at organizations with different cultures, technical stacks, project types, or developer demographics. N=54, drawn from approximately 100 invited engineers at roughly a 54% response rate, leaves limited power

for detecting small-to-moderate effects or running fine-grained subgroup analysis. Non-response bias remains possible, since the 46% who did not answer may hold systematically different attitudes or usage patterns.

Capturing perceptions at one point in time rules out conclusions about causality or how any of this moves. The correlation between usage frequency and productivity could run either way, or could come from a third variable such as individual motivation or task allocation shaping both. Longitudinal data following individuals as they adopt and integrate these tools would show the mechanism and the trajectory in a way a single measurement cannot.

Self-report brings its own biases: social desirability, imperfect recall of past behaviour, and the subjectivity of a rating scale where one developer's "4" is another's "3." The JIRA analysis in the preceding section offsets part of that with team-level behavioral data on throughput, cycle time, and bug ratio, none of which depends on what anyone remembers or chooses to report. The offset is partial, though, limited by five teams (N=5) and a descriptive-only approach. Larger team samples over longer observation would strengthen the mixed-methods design considerably.

The 33% response rate on open-ended benefit questions thins the qualitative material there, though the 50% rate on challenges gave thematic analysis enough to work with.

Finally, these tools are moving fast enough that the findings describe 2023-2025. Longer context windows, better reasoning, or fewer hallucinations could resolve the limitations reported here and introduce others nobody has met yet.

# 5. CONCLUSION

## 5.1 Summary of Findings

This research examined how AI-powered development agents affect software developer productivity, through a mixed-methods case study at a mid-size software company. Four research questions were addressed using survey data from 54 AI tool users working in backend, frontend, and quality assurance roles, supplemented by team-level JIRA project management metrics from five anonymized development teams.

### 5.1.1 RQ1: How do AI-powered development agents impact software developer productivity?

Perceived productivity gains are positive and moderate: developers rated improvement above the neutral threshold on every dimension measured. Usage frequency and perceived productivity correlate positively (Spearman $\rho$=0.2823, p=0.0386). Daily sustained use appears to return more than occasional use does. Effectiveness depends heavily on the task. Pattern-based, automatable work such as repetitive coding and unit test generation shows the strongest gains; architectural guidance and team collaboration show almost none. These field perceptions come in well below the effects reported from controlled laboratory studies (Peng et al. 2023, Ziegler et al. 2024). Production work is a great deal messier than a lab task, so the gap is unsurprising. Every respondent used at least one AI tool and 63% used one daily, so moderate gains are evidently enough to keep people coming back.

### 5.1.2 RQ2: How does the impact vary across developer roles and experience levels?

Neither role nor experience level produced a statistically significant difference in productivity impact. Junior developers and QA engineers rate their gains slightly higher, but with N=54 spread over uneven subgroups that difference is not reliable. The null result sits against prior research using objective metrics under randomized assignment (Cui et al. 2025), which did find experience-level differences. The divergence may come from measuring perception rather than performance, from something particular to mid-size organizations, or simply from too little power to detect a moderate effect. Code quality ratings behaved identically, showing no significant variation by experience level. Code

review practices may level quality across seniority regardless of who or what wrote the code.

### 5.1.3 RQ3: What are developers' perceptions of AI tool effectiveness across different task types?

The effectiveness ratings fall into a clear order. Repetitive tasks (3.80/5) and unit test generation (3.72/5) lead, matching what these models genuinely do well: recognize patterns and apply templates. Documentation (3.56/5) sits in the middle. Bug diagnosis (3.06/5) and architecture guidance (3.07/5) land near neutral, held back by weak contextual reasoning, and team collaboration comes last at 2.70/5, since these tools were never built for interpersonal work. The pattern supports a division-of-labor reading of human-AI collaboration: automatable pattern-based work to the machine, judgment and creativity and working with other people to the human.

### 5.1.4 RQ4: What challenges, concerns, and limitations do developers face when using AI tools?

Prompt engineering difficulty (33.3%), limited context awareness (25.9%), and hallucinations (18.5%) dominated, with code review burden and dependency concerns following at 11.1% each. That prompt engineering tops the list points to a skill gap, since it appears in no standard software engineering curriculum. Limited context awareness, meaning tools that do not know a project's conventions, internal libraries, or domain rules, forces heavy review and rewriting: only 27.8% of developers find AI code usually acceptable on the first try, and 53.7% find it acceptable only sometimes. Hallucinations, where the model produces wrong code with full confidence, explain why 70.4% always review AI output before merging it. What these tools produce are drafts. Taken together these eat into the productivity gains and point at clear priorities for vendors and for internal training alike.

## 5.2 Contributions

Theoretically, the research supplies evidence from a mid-size, non-elite organization, a setting largely absent from a literature dominated by Microsoft, GitHub, and Google. Moderate task-dependent gains and no role or experience differences complicate any assumption that FAANG-scale findings carry over to ordinary software companies. The task-

effectiveness pattern also supports a division-of-labor model of human-AI collaboration, with AI strong on automatable pattern-based work and weak on architecture, debugging, and collaboration. That is a more specific claim than "AI helps productivity."

Methodologically, the study brings together Likert survey data, JIRA-based team metrics, and thematic analysis of open-ended responses, using non-parametric tests (Kruskal-Wallis, Spearman, chi-square) suited to ordinal data and small samples rather than defaulting to parametric tests the data cannot support. The 49-item instrument itself covers adoption, productivity, task effectiveness, and code quality in a single pass, and can be reused at other organizations.

Practically, the recommendations in Section 5.4 turn these findings into concrete guidance for organizations, teams, and individual developers deciding how to adopt these tools.

## 5.3 Limitations

### 5.3.1 Single-company case study design

The single-company design is the most serious limitation. All 54 participants work at one mid-size software organization and share its culture, management practices, technical stack, project types, and team dynamics. Company-specific factors may shape the findings in ways that will not travel. The stack here is primarily .NET, Java, and JavaScript, and AI tool effectiveness may differ at organizations built on Python or Go. The code review culture, where 70.4% always review, may be heavier or lighter elsewhere, changing how AI-generated code moves into a codebase.

Case study methodology buys contextual depth and supports analytical generalization, meaning generalization to theory rather than to populations, but it cannot support statistical generalization. What these findings show is how AI tools affect developers in this specific context, which transfers to similar organizations as insight rather than as evidence. Readers have to judge applicability against how closely their own context resembles this one.

### 5.3.2 Sample size and statistical power

Fifty-four AI tool users, drawn from approximately 100 invited engineers, limits statistical power for detecting small-to-moderate effects and for fine-grained subgroup analysis. The 54% response rate is normal for organizational survey research but still leaves room for non-response bias, since engineers who declined may differ systematically in their attitudes or usage. Several comparisons that matter came back non-significant (productivity by role, productivity by experience, code quality by experience), and insufficient power is at least as plausible an explanation as a genuine absence of difference.

After consolidating smaller role categories into three primary groups, the smallest (QA & Automation, n=7) still supports only modest power for detecting moderate effects, while Backend (n=33) and Frontend (n=14) do somewhat better. Larger and more balanced samples across roles and organizations would be needed before any conclusion about role or experience moderation could be held with confidence.

### 5.3.3 Cross-sectional design and causality

Capturing perceptions and behaviors at one point in time rules out causal conclusions. The correlation between usage frequency and productivity (Spearman $\rho$=0.28, p=0.0386) admits several readings: heavier use may drive productivity, more productive developers may reach for the tools more often, or something else entirely, individual motivation or task allocation or technical aptitude, may drive both.

The design also cannot see change over time: how the developer-AI relationship evolves across months, whether gains grow as prompting improves, whether they decay as novelty fades, or whether they settle somewhere stable. Longitudinal work following individuals across months or years would show the mechanism in a way a snapshot never can.

### 5.3.4 Self-reported perceptions and response biases

Self-reported perception carries several biases. Social desirability may push participants toward presenting themselves favorably, inflating productivity ratings or muting complaints. Recall bias may distort retrospective reports of usage frequency, task effectiveness, or past behavior. Common method bias, where variance comes from the measure-

ment method rather than the construct, may inflate correlations between self-reported variables.

How individuals read a rating scale adds further variance. One developer's "4" is another's "3" for an identical experience. That is a difference in internal reference points, not in what actually happened, and the cross-sectional design offers no repeated measures to control for individual baselines.

Supplementary analysis of JIRA data was conducted to offset the reliance on perception. Team-level throughput, cycle time, and bug ratio metrics across five anonymized teams over seven-month pre-AI and post-AI periods supply behavioral data immune to reporting bias. The direction of change is consistent with the survey: throughput up 17.1% on a cross-team average, cycle time down 27.8%, bug ratio down 3.4 percentage points. That partially supports the moderate positive perceptions developers reported. The strength of this triangulation is nonetheless limited by the small number of teams (N=5), a descriptive-only approach, and confounds including team composition changes, project maturity, and seasonal variation. Larger team samples over longer observation periods would strengthen the design considerably.

### 5.3.5 Open-ended response rates

Response rates on the open-ended questions were uneven: 33% wrote about benefits (n=18), 50% about challenges (n=27). Developers are more motivated to articulate a problem than a benefit, a familiar pattern in feedback surveys. The lower rate on benefits suggests the structured questions already captured most of the positive impact, while the higher rate on challenges gave thematic analysis considerably more to work with.

### 5.3.6 Rapidly evolving technology

AI coding assistant capability moves in months, not years, with new model releases and features arriving continuously. This study captures the tools as they stood in 2023-2025, a snapshot of something that keeps changing. Longer context windows, better reasoning, fewer hallucinations, or tighter IDE integration could each shift effectiveness substantially.

Findings about current limitations, particularly limited context awareness (25.9%) and

hallucinations (18.5%), may lose relevance as models improve, and new difficulties may appear as capability expands and usage patterns shift with it. What is reported here belongs to a transitional period in AI coding assistant maturity and should be read as describing current-generation tools rather than AI capability in general.

### 5.3.7 Scope boundaries

Several things that probably matter fall outside this study. It does not compare specific tools against one another, so nothing here supports a claim that Copilot beats ChatGPT or Claude. It does not investigate individual developer characteristics such as cognitive style, educational background, or prior attitudes toward AI. Organizational factors that shape adoption success, management support, team culture, training approaches, integration strategy, go unexamined, as do project characteristics that may moderate suitability, greenfield against legacy, domain complexity, regulatory constraints.

Studying those moderators across multiple organizations would map the boundary conditions of AI tool effectiveness and allow recommendations tuned to a specific context rather than averaged across all of them.

## 5.4 Implications for Practice

### 5.4.1 For software organizations

Expectations should start modest. The average productivity rating of 3.63/5 points to incremental gains, and budgets or success metrics built around dramatic transformation will read as failure even when a tool is working about as well as it realistically can.

Deployment is worth targeting rather than spreading evenly. QA and Automation (3.86) and Backend teams (3.61) doing repetitive implementation, API work, and test generation see the clearest benefit, while architecture, complex debugging, and collaboration see little, so rollout and training should follow that pattern instead of treating every task the same.

Buying licenses is the easy part. A third of developers (33.3%) name prompt engineering as their top challenge, and only 27.8% find AI output usually acceptable on the first try.

Structured training in prompting, iterative refinement, and how to give the model useful context addresses that directly, and internal documentation or mentoring spreads the skill faster than developers working it out alone.

Code review standards should hold. With 18.5% of developers reporting hallucinated code and 70.4% saying they always review AI output, the sensible response is to keep treating AI code as a draft rather than loosen review because a tool wrote it.

Regular use is worth encouraging, since usage frequency correlates with perceived productivity ($\rho$=0.28, p=0.0386) and occasional trial use is unlikely to show the same benefit as daily integration. Enterprise licensing that removes cost friction helps, as does leadership example that normalizes the habit.

The one thing worth watching is collaboration. It is the weakest-rated use case at 2.70/5, and heavier individual AI use could quietly displace pairing and design discussions. If that starts to show, protecting time for collaborative work beats assuming it will take care of itself.

### 5.4.2 For development teams

Teams should agree on their own norms: when to use AI assistance, how to flag AI-assisted code in review, and what level of scrutiny it needs. Clear norms beat everyone improvising their own rules.

Prompting has a learning curve, and teams that share effective prompts and discuss what didn't work will climb it faster than developers figuring it out alone.

Pairing needs deliberate protection. AI helps with individual coding but not with collaboration (2.70/5), so time for pair programming and group design sessions matters more, not less, as individual work gets faster.

### 5.4.3 For individual developers

Individual developers can lean on these tools for repetitive tasks, tests, and documentation, provided they review what comes back. With an 18.5% hallucination rate and only 27.8% first-try acceptance, treating AI output as a draft rather than a finished answer isn't optional.

Prompting rewards practice. Clear context, specific output requirements, and iterating on results all improve what comes back, and the skill is learned the way any other technique is: by doing it and paying attention to what worked.

Some 11.1% of respondents raised dependency or skill-atrophy concerns. Solving problems without AI occasionally, and actually understanding the code it hands you instead of accepting it, keeps those fundamentals intact.

The time saved on repetitive work (3.80) and tests (3.72) is best spent on architecture and hard debugging, the areas where human judgment still clearly wins.

## 5.5 Future Research Directions

What this study could not settle points in several directions.

### 5.5.1 Longitudinal tracking of skill and productivity

This survey captured a single point in time. Following the same developers for a year or more would show whether productivity gains grow as people get better at prompting, or fade once the novelty wears off. It would also allow a real comparison of skill development between heavy AI users and developers who mostly write code by hand. That is the kind of evidence needed to test the skill-atrophy concern several respondents raised, rather than guess at it.

### 5.5.2 Comparison across multiple organizations

One company, however varied its teams, cannot show what changes by industry, company size, or tech stack. Running the same survey and the same JIRA-style metrics across several organizations (a startup, a large enterprise, a team on a legacy codebase versus one on a modern one) would show which findings here are general and which were specific to this workplace.

### 5.5.3 Objective data from AI tool telemetry

Self-reported productivity has real limits, and the JIRA metrics used here are only a rough proxy. Several coding tools already log this directly: GitHub Copilot and Claude Code, for example, track suggestion acceptance rates, how much of the suggested code survives into the final commit, and how long it takes a developer to accept or reject a suggestion. Pairing that telemetry with a survey like this one would give a much sharper picture of how much AI-generated code developers actually keep versus rewrite, without relying on what people remember or choose to report.

### 5.5.4 Head-to-head tool comparisons

This study treated "AI coding assistant" as one category, but an in-editor autocomplete tool like Copilot and a chat tool like Claude work in very different ways. A controlled comparison, same developers, same tasks, different tools, would show which interaction style actually helps with which kind of work, instead of averaging them together.

### 5.5.5 Tracking model improvements over time

The tools measured here reflect model generations from 2023 to 2025. Coding models keep climbing on public benchmarks, and it is not obvious whether developers' day-to-day productivity keeps pace or whether gains plateau once the easy tasks are automated. Repeating this survey as new model generations ship would test whether benchmark progress actually shows up in working developers' experience, or whether the two have quietly come apart.

### 5.5.6 How prompting skill is learned

Prompt engineering was the single most common complaint in this survey (33.3%), but almost nothing is known about how developers get better at it. Tracking developers as they learn to prompt, what trips them up early, what kind of examples or feedback actually help, would turn a widely cited pain point into something teams can train for.

### 5.5.7 Better context for AI tools

Limited context awareness was the second most common complaint (25.9%). Techniques like retrieval-augmented generation and codebase indexing are already being built to address it, but there is little field evidence on whether they change what developers actually experience day to day. Testing these approaches directly against the complaints raised here would be a natural next step.

## 5.6 Closing Reflection

AI-powered development agents have not transformed software development the way vendors promised. What they have become is a routine part of the job: useful for repetitive and pattern-based work, weak on architecture and collaboration, and still requiring the same verification discipline as any other draft. Every developer surveyed used at least one AI tool, and most used one daily; in two or three years, AI assistance has gone from novelty to something developers simply expect, the way they expect autocomplete.

What this study can't answer is how that changes over time, at other companies, or as the models themselves improve. Section 5.5 lays out where the next studies should look. For now, the practical takeaway holds regardless of how the technology moves: these tools speed up the parts of programming that are already mechanical, and the parts that require judgment, context, and working with other people remain, for now, a human job.

**REFERENCES**


Amazon Web Services. 2024. Code Generation Using Code Llama 70B and Mixtral 8x7B on Amazon SageMaker. AWS Machine Learning Blog. [Industry Blog]. https://aws.amazon.com/blogs/machine-learning/code-generation-using-code-llama-70b-and-mixtral-8x7b-on-amazon-sagemaker/

Anthropic. 2026. How AI Assistance Impacts the Formation of Coding Skills. Anthropic Research Report. [Company Research Report]. https://www.anthropic.com/research/AI-assistance-coding-skills

Bakal, G., Dasdan, A., Katz, Y., Kaufman, M., and Levin, G. 2025. Experience with GitHub Copilot for Developer Productivity at Zoominfo. arXiv preprint arXiv:2501.13282. https://arxiv.org/abs/2501.13282

Barke, S., James, M.B., and Polikarpova, N. 2023. Grounded Copilot: How Programmers Interact with Code-Generating Models. Proceedings of the ACM on Programming Languages, 7(OOPSLA1), 85-111. https://doi.org/10.1145/3586030

Becker, J., Rush, N., Barnes, E., and Rein, D. 2025. Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. METR Technical Report. arXiv preprint arXiv:2507.09089. https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/

Brandebusemeyer, C., Schimmer, T., and Arnrich, B. 2026. Developers' Experience with Generative AI—First Insights from an Empirical Mixed-Methods Field Study. In Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2026) (pp. 259-268). ACM. https://doi.org/10.1145/3786583.3786870

Brannon, D. 2026. DORA vs. SPACE: A Framework for Measuring What Matters. Method Insights. [Practitioner Blog]. https://www.method.com/insights/dora-vs-space/

Braun, V., and Clarke, V. 2006. Using thematic analysis in psychology. Qualitative Research in Psychology, 3(2), 77-101.

Brooks, F.P. 1975. The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, Reading, MA.

Brynjolfsson, E., and McAfee, A. 2017. The business of artificial intelligence. Harvard Business Review, 7, 3-11.

Cavalcante, S., Ribeiro, E., and Oran, A.C. 2025. The Impact of AI Tools on Software Development: A Case Study with GitHub Copilot and Other AI Assistants. In Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025) - Volume 2 (pp. 245-252). SciTePress. https://doi.org/10.5220/0013294700003929

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.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., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. 2021. Evaluating Large Language Models Trained on Code. arXiv preprint arXiv:2107.03374. https://arxiv.org/abs/2107.03374

Chen, V., He, J., Williams, B., Valentino, J., and Talwalkar, A. 2026. Beyond the Commit: Developer Perspectives on Productivity with AI Coding Assistants. In Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2026) (pp. 1-9). ACM. arXiv preprint arXiv:2602.03593.

Chukkala, R. 2025. The Future of Human-AI Collaboration in Software Development: Automating Code, Deployment and Testing. World Journal of Advanced Engineering Technology and Sciences, 15(2), 312-320. https://doi.org/10.30574/wjaets.2025.15.2.0501

CodePulse Team. 2026. DORA vs SPACE Framework: Which Metrics Framework Should You Use? CodePulse Guides. [Practitioner Blog]. https://codepulsehq.com/guides/dora-vs-space-framework-comparison

Codex, A.C. 2026. DORA Metrics vs SPACE Framework: Which Developer Productivity Framework to Use in 2026. Reintech. [Practitioner Blog]. https://reintech.io/blog/dora-metrics-vs-space-framework-developer-productivity-2026

Cohen, J. 1988. Statistical Power Analysis for the Behavioral Sciences (2nd ed.). Lawrence Erlbaum Associates, Hillsdale, NJ.

Creswell, J.W., and Creswell, J.D. 2018. Research design: Qualitative, quantitative, and mixed methods approaches (5th ed.). Sage Publications, Thousand Oaks, CA.

Crupi, G., Tufano, R., Velasco, A., Mastropaolo, A., Poshyvanyk, D., and Bavota, G. 2025. On the Effectiveness of LLM-as-a-Judge for Code Generation and Summarization. IEEE Transactions on Software Engineering, 51(8), 2329-2345. https://doi.org/10.1109/TSE.2025.3586082

Cui, K.Z., Demirer, M., Jaffe, S., Musolff, L., Peng, S., and Salz, T. 2025. The Productivity Effects of Generative AI: Evidence from a Field Experiment with GitHub Copilot. Working Paper. Published as: The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers. Management Science (2026). https://doi.org/10.1287/mnsc.2025.00535

Davila, N., Wiese, I., Steinmacher, I., Lucio da Silva, L., Kawamoto, A., Favaro, G.J.P., and Nunes, I. 2024. An industry case study on adoption of AI-based programming assistants. In Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice (pp. 92-102).

De La Cruz, E., Le, H., Meduri, K., Nadella, G.S., and Gonaygunta, H. 2025. Redefining the Programmer: Human-AI Collaboration, LLMs, and Security in Modern Software Engineering. Computers, Materials and Continua, 85(2), 3569-3582. https://doi.org/10.32604/cmc.2025.068137

DeMarco, T., and Lister, T. 1987. Peopleware: Productive Projects and Teams. Dorset House Publishing, New York.

Dillman, D.A., Smyth, J.D., and Christian, L.M. 2014. Internet, phone, mail, and mixed-mode surveys: The tailored design method (4th ed.). Wiley, Hoboken, NJ.

Dong, T. 2026. Code Review Queue Health Score: A Team Ops Metric That Keeps PRs Moving. Propel Engineering Blog. [Practitioner Blog]. https://www.propelcode.ai/blog/code-review-queue-health-score

Erhabor, D., Udayashankar, S., Nagappan, M., and Al-Kiswany, S. 2025. Measuring the Runtime Performance of C++ Code Written by Humans Using GitHub Copilot. In Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025) (pp. 2062-2074). IEEE. https://doi.org/10.1109/ICSE55347.2025.00059

Forsgren, N., Humble, J., and Kim, G. 2018. Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution Press, Portland, OR.

Forsgren, N., Storey, M.A., Maddila, C., Zimmermann, T., Houck, B., and Butler, J. 2021. The SPACE of Developer Productivity: There's More to It Than You Think. ACM Queue, 19(1), 20-48. https://doi.org/10.1145/3454122.3454124

Huang, L., Yan, M., Yin, T., Sun, W., Liu, Z., Zhang, H., and Lo, D. 2026. Steer Your Model: Secure Code Generation With Contrastive Decoding. IEEE Transactions on Software Engineering, 52(3), 809-834. https://doi.org/10.1109/TSE.2025.3650127

Huynh, N., and Lin, B. 2025. Large Language Models for Code Generation: A Comprehensive Survey of Challenges, Techniques, Evaluation, and Applications. arXiv preprint arXiv:2503.01245v2. https://arxiv.org/abs/2503.01245

Khojah, R., Gomes de Oliveira Neto, F., Mohamad, M., and Leitner, P. 2025. The Impact of Prompt Programming on Function-Level Code Generation. IEEE Transactions on Software Engineering, 51(8), 2381-2395. https://doi.org/10.1109/TSE.2025.3587794

Kitchenham, B., and Charters, S. 2008. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE-2007-01, Keele University.

Kruskal, W.H., and Wallis, W.A. 1952. Use of ranks in one-criterion variance analysis. Journal of the American Statistical Association, 47(260), 583-621.

Lazar, J., Feng, J.H., and Hochheiser, H. 2017. Research methods in human-computer interaction (2nd ed.). Morgan Kaufmann, Burlington, MA.

Likert, R. 1932. A Technique for the Measurement of Attitudes. Archives of Psychology, 140, 1-55.

Microsoft. 2025. Microsoft Fiscal Year 2025 Fourth Quarter Earnings Conference Call. Microsoft Investor Relations. [Corporate Earnings Call Transcript]. https://www.microsoft.com/en-us/investor/events/fy-2025/earnings-fy-2025-q4

Microsoft. 2026. Microsoft Fiscal Year 2026 Second Quarter Earnings Conference Call. Microsoft Investor Relations. [Corporate Earnings Call Transcript]. https://www.microsoft.com/en-us/investor/events/fy-2026/earnings-fy-2026-q2

Moradi Dakhel, A., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M.C., and Jiang, Z.M. 2023. GitHub Copilot AI Pair Programmer: Asset or Liability? Journal of Systems and Software, 203, 111734. https://doi.org/10.1016/j.jss.2023.111734

Ni, A., Yin, P., Zhao, Y., Riddell, M., Feng, T., Shen, R., Yin, S., Liu, Y., Yavuz, S., Xiong, C., Joty, S., Zhou, Y., Radev, D., and Cohan, A. 2023. L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models. arXiv preprint arXiv:2309.17446. https://arxiv.org/abs/2309.17446

Noda, A., Storey, M.A., Forsgren, N., and Greiler, M. 2023. DevEx: What Actually Drives Productivity. ACM Queue, 21(2), 35-53. https://doi.org/10.1145/3595878

Pandey, R., Singh, P., Wei, R., and Shankar, S. 2024. Transforming Software Development: Evaluating the Efficiency and Challenges of GitHub Copilot in Real-World Projects. arXiv preprint arXiv:2406.17910. https://arxiv.org/abs/2406.17910

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In 2022 IEEE Symposium on Security and Privacy (SP), 754-768. https://doi.org/10.1109/SP46214.2022.9833571

Pendyala, V.S., and Thakur, N.B. 2025. Performance and Interpretability Analysis of Code Generation Large Language Models. Neurocomputing, 656, 131461. https://doi.org/10.1016/j.neucom.2025.131461

Peng, S., Kalliamvakou, E., Cihon, P., and Demirer, M. 2023. The impact of AI

on developer productivity: Evidence from GitHub Copilot. arXiv preprint arXiv:2302.06590. https://arxiv.org/abs/2302.06590

Podsakoff, P.M., MacKenzie, S.B., Lee, J.Y., and Podsakoff, N.P. 2003. Common method biases in behavioral research: A critical review of the literature and recommended remedies. Journal of Applied Psychology, 88(5), 879-903. https://doi.org/10.1037/0021-9010.88.5.879

Princis, H., Sharma, A., and David, C. 2025. TreeCoder: Systematic Exploration and Optimisation of Decoding and Constraints for LLM Code Generation. arXiv preprint arXiv:2511.22277. https://arxiv.org/abs/2511.22277

Qian, C., and Wexler, J. 2024. Take It, Leave It, or Fix It: Measuring Productivity and Trust in Human-AI Collaboration. In Proceedings of the 29th International Conference on Intelligent User Interfaces (IUI '24) (pp. 370-384). ACM. https://doi.org/10.1145/3640543.3645198

Rogers, E.M. 2003. Diffusion of Innovations (5th ed.). Free Press, New York.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C.C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. 2024. Code Llama: Open Foundation Models for Code. arXiv preprint arXiv:2308.12950. https://arxiv.org/abs/2308.12950

Sabouri, S., Eibl, P., Zhou, X., Ziyadi, M., Medvidovic, N., Lindemann, L., and Chattopadhyay, S. 2025. Trust Dynamics in AI-Assisted Development: Definitions, Factors, and Implications. In Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE 2025) (pp. 1678-1690). IEEE. https://doi.org/10.1109/ICSE55347.2025.00199

Schoonenboom, J., and Johnson, R.B. 2017. How to Construct a Mixed Methods Research Design. Kölner Zeitschrift für Soziologie und Sozialpsychologie, 69(Suppl 2), 107-131. https://doi.org/10.1007/s11577-017-0454-1

Shah, A., Rexin, T., Tomson, E., Porter, L., Griswold, W.G., and Soosai Raj, A.G. 2025. Evolution of Programmers' Trust in Generative AI Programming Assistants. In Proceedings of the 25th Koli Calling International Conference on Computing Education Research (Koli Calling '25) (pp. 1-11). ACM. https://doi.org/10.1145/3769994.3770029

Sonar. 2026. State of Code Developer Survey Report 2026. SonarSource. [Industry Report]. https://www.sonarsource.com/state-of-code-developer-survey-report.pdf

Spearman, C. 1904. The proof and measurement of association between two things. The American Journal of Psychology, 15(1), 72-101.

Stack Overflow. 2025. Developer Survey 2025. Stack Overflow Insights. [Industry Survey]. https://survey.stackoverflow.co/2025/

Storey, M.A., Zimmermann, T., Bird, C., Czerwonka, J., Murphy, B., and Kalliamvakou, E. 2021. Towards a Theory of Software Developer Job Satisfaction and Perceived Productivity. IEEE Transactions on Software Engineering, 47(10), 2125-2142. https://doi.org/10.1109/TSE.2019.2944354

Storey, M.A., Hoda, R., Milani, A.M.P., and Baldassarre, M.T. 2025. Guiding Principles for Mixed Methods Research in Software Engineering. Empirical Software Engineering, 30(5). https://doi.org/10.1007/s10664-025-10629-x

Stray, V., Brandtzæg, E.G., Wivestad, V.T., Barbala, A., and Moe, N.B. 2026. Developer Productivity With and Without GitHub Copilot: A Longitudinal Mixed-Methods Case Study. In Proceedings of the 59th Hawaii International Conference on System Sciences (HICSS-59) (pp. 7413-7422). https://arxiv.org/abs/2509.20353

Tomaz, R., Guenes, P., Araújo, A.A., Baldassarre, M.T., and Kalinowski, M. 2026. Impacts of Generative AI on Agile Teams' Productivity: A Multi-Case Longitudinal Study. In Proceedings of the IEEE/ACM Third International Conference on AI Foundation Models and Software Engineering (Forge@ICSE 2026) (pp. 128-138). ACM. arXiv preprint arXiv:2602.13766. https://doi.org/10.1145/3793655.3793728

Ulloa, M., Butler, J.L., Haniyur, S., Miller, C., Amos, B., Sarkar, A., and Storey, M.A. 2026. Product Manager Practices for Delegating Work to Generative AI. In Proceedings of the IEEE/ACM 48th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2026) (pp. 10-21). ACM.

Vaithilingam, P., Zhang, T., and Glassman, E.L. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In Chi Conference on Human Factors in Computing Systems Extended Abstracts (pp. 1-7).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., and Polosukhin, I. 2017. Attention Is All You Need. In Advances in Neural Information Processing Systems 30 (NeurIPS 2017), 5998-6008.

Wang, R., Cheng, R., Ford, D., and Zimmermann, T. 2024. Investigating and Designing for Trust in AI-powered Code Generation Tools. In Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency (FAccT '24) (pp. 1475-1493). ACM. https://doi.org/10.1145/3630106.3658984

Yin, R.K. 2018. Case study research and applications: Design and methods (6th ed.). Sage Publications, Thousand Oaks, CA.

Zhao, J., Sun, Y., Huang, C., Liu, C., Guan, Y., Zeng, Y., and Liu, Y. 2025. Towards Secure Code Generation With LLMs: A Study on Common Weakness Enumeration. IEEE

Transactions on Software Engineering, 51(12), 3507-3523. https://doi.org/10.1109/TSE.2025.3619281

Zhou, X., Saghi, Z., Sabouri, S., Pandita, R., McGuire, M., and Chattopadhyay, S. 2026. Cognitive Biases in LLM-Assisted Software Development. In Proceedings of the 48th International Conference on Software Engineering (ICSE 2026). arXiv preprint arXiv:2601.08045. https://arxiv.org/abs/2601.08045

Ziegler, A., Kalliamvakou, E., Li, X.A., Rice, A., Rifkin, D., Simister, S., Sittampalam, G., and Aftandilian, E. 2024. Measuring GitHub Copilot's Impact on Productivity. Communications of the ACM, 67(3), 54-63. https://doi.org/10.1145/3633453

## APPENDIX 1: SURVEY INSTRUMENT

This appendix reproduces the survey items presented to AI tool users, the analytical sample described in Section 3.4 (N=54). Response options are listed as they appeared in the original instrument.

### Section 1: General Information

1. **Professional Background** — Backend; Frontend; DevOps; QA & Automation; Other
2. **Experience Level** — Intern; Junior; Mid-level; Senior; Team Lead/Technical Lead; Product Owner/Manager
3. **Primary Programming Language** — C#; JavaScript/TypeScript; Java; Python; PHP; C/C++; Other
4. **How long have you been using AI coding tools** — Less than 1 month; 1–3 months; 4–6 months; 7–12 months; 1–2 years; More than 2 years

### Section 2: AI Tool Usage Patterns

5. **Type of AI Tools Used** (check all that apply) — Web-based AI tools (ChatGPT, Claude, Gemini — not integrated with codebase); Codebase-integrated AI tools (GitHub Copilot — integrated within IDE)
6. **Specific AI Tools Currently Used** (check all that apply) — GitHub Copilot; OpenCode; JetBrains AI Assistant; ChatGPT; Claude; Gemini; DeepSeek; Grok; Windsurf; Cursor; Kiro; Other
7. **How often do you use AI coding assistants** — Daily; Several times a week; Once a week; Rarely
8. **License Type** (check all that apply) — Free; Paid Personal; Business/Enterprise

**Section 3: Tool Effectiveness**

9. **How often do you check generated code by AI tools** — Always; Usually; Sometimes; Rarely; Almost Never
10. **How often do AI tools provide the solution you need on the first try** — Always; Usually; Sometimes; Rarely; Almost never
11. **When an AI tool doesn't provide the right solution initially, how many attempts do you typically make** — 1 (give up after first try); 2–3 attempts; 4–5 attempts; 6–10 attempts; More than 10 attempts
12. **AI tools sometimes suggest more code than I need** — Always; Often; Sometimes; Rarely; Never
13. **When you encounter a coding question or problem, what do you check first** — AI tools; Google search; Stack Overflow; Official documentation; Ask colleagues
14. **Since implementing AI tools in your workflow, how would you rate the change in each of these areas** (1 = Significantly decreased to 5 = Significantly increased):
    - Overall Productivity Change
    - Code Quality Impact
    - Meeting Deadlines
    - Focus on Complex Tasks
    - Overall work experience
15. **How effective AI tools have supported you in each of these activities** (1 = Not effective to 5 = Extremely effective):
    - Design and Architecture Planning
    - Technical Documentation and Knowledge Management
    - Team Collaboration and Communication
    - AI tools help reduce time spent on repetitive coding tasks
    - Creating Comprehensive Unit Tests
    - Bug Diagnosis and Troubleshooting
    - Help discover edge cases or business logic scenarios I might have missed

16. **How AI tools have impacted your dependency on the following resources** (1 = Much less dependent to 5 = Much more dependent):
    - Stack Overflow
    - Official documentation
    - Technical blogs and articles
    - Video tutorials
    - Colleague consultation
17. **How much do you agree with the following statements about AI tools in your work** (1 = Strongly disagree to 5 = Strongly agree):
    - Improve my learning of new technologies
    - Help me understand new codebases
18. **For which types of coding tasks do you find AI tools most effective** (1 = Not effective to 5 = Extremely effective):
    - Writing boilerplate code
    - Implementing algorithms
    - API integration
    - Database queries
    - Unit test creation
    - Code refactoring
    - Documentation generation
    - Debugging assistance
19. **How concerned are you about the following potential impacts of AI tools on your professional development** (1 = Not concerned at all to 5 = Extremely concerned):
    - Becoming too dependent on AI assistance
    - Losing fundamental programming skills
    - Reduced problem-solving abilities
    - Impact on career advancement
    - Job security concerns

**Section 4: Return on Investment and Open-Ended Questions**

20. **How do you quantify the return on investment from AI tools** (check all that apply) — Time saved per day/week; Increased project completion rate; Reduced debugging time; Improved code quality metrics; I don't measure ROI
21. **What are the main challenges or disadvantages you've experienced with AI coding tools?** (open-ended)
22. **Are there any other benefits from using AI tools that weren't mentioned in the survey?** (open-ended)
23. **What improvements would you like to see in AI coding tools?** (open-ended)