UPPSALA
UNIVERSITET

UPTEC STS 26024
Examensarbete 30 hp
May 2026

Measuring the Productivity Impact of Agentic Coding Systems

A Metadata-Based Framework for SaaS Organisations

Emelie Edberg

Civilingenjörsprogrammet i system i teknik och samhälle

UPPSALA
UNIVERSITET

Measuring the Impact of Agentic Coding Systems

Emelie Edberg

Abstract

This thesis proposes and evaluates a framework for measuring the productivity impact of agentic coding systems in software engineering teams. The framework consists of five metadata-based metrics covering the two key dimensions of software engineering productivity established in the literature: development velocity and software quality. The metrics rely on metadata extracted from GitHub and Jira. The framework was evaluated through a longitudinal case study of three software engineering teams (team A, team B, and team C) within a Nordic B2B SaaS company over a six-month period. Weekly quantitative data was combined with semi-structured interviews with software engineers from each team and complemented by expert interviews providing context on the broader adoption journey of agentic coding systems.

The three teams exhibited distinctly different adoption patterns. In team A, adoption varied significantly between team members, ranging from non-adopters to engineers who use coding agents for full task execution in most of their work. Team B showed a more uniformly distributed adoption pattern, where all team members use coding agents selectively for tasks they believe agents perform well on. Team C displayed a more foundationally invested pattern, where team members have made an effort to provide agents with the infrastructure needed to complete work in line with the team's standards. Team C showed the clearest upward trend in both development velocity and software quality indicators, followed by team B, while team A's individually driven adoption pattern resulted in fluctuating outcomes that could not be reliably attributed to agent usage. Overall, the study shows that adoption is driven by both organisational and individual factors. Software engineers' perceptions of the advantages that come with using coding agents play a particularly significant role in adoption.

The study concludes that agent usage does not automatically translate into measurable productivity gains. Rather, the quality of adoption, characterised by foundational investment, shared practices, and integration into workflows, appears to support sustained improvement. The proposed metadata-based framework offers a feasible and scalable approach to evaluating the productivity impact of agentic coding systems, but its reliability depends on continuous qualitative input that can contextualise the quantitative trends.

Teknisk-naturvetenskapliga fakulteten

Uppsala universitet, Utgivningsort Uppsala/Visby

Handledare: Nils Hulth Ämnesgranskare: Göran Lindström

Examinator: Elísabet Andrésdóttir

Populärvetenskaplig sammanfattning

Artificiell intelligens (AI) har under de senaste åren blivit ett allt mer omtalat ämne inom mjukvaruindustrin. Tidigare har mjukvaruutvecklare främst använt sig av generativ AI, alltså tekniken som gör det möjligt att generera kod utifrån att en utvecklare beskriver problemet de vill lösa och hur koden ska fungera på naturligt språk. Denna typ av interaktion med AI går att jämföra med ett bollplank. Utvecklaren kan ställa konceptuella frågor och få hjälp med begränsade kodavsnitt, men i slutändan är det upp till den mänskliga utvecklaren att granska, anpassa och integrera koden i den större helheten.

Nu sker dock ett nytt skifte som många menar kommer att förändra hur företag utvecklar mjukvara med framväxten av så kallade autonoma kodningsagenter. Dessa agenter har kapaciteten att ta en mer aktiv roll i utvecklingsarbetet. En utvecklare kan delegera en uppgift till en agent som sedan självständigt läser igenom kodbasen, planerar vilka ändringar som behövs, skriver och testar koden och utvärderar sitt eget resultat innan den överlämnar arbetet till utvecklaren. Vissa mjukvaruutvecklare jämför samarbetet med kodningsagenter med att ha en junior utvecklare i teamet där man kan få mycket avlastning i sitt dagliga arbete och på så vis bli mer produktiv. Samtidigt beror värdet på hur bra man är på att förklara problemet, hur kodbasen är strukturerad och hur väl man introducerar den nya ”kollegan” till företagets sätt att arbeta.

Tidigare har mjukvaruföretag varit begränsade av antalet utvecklare de kunnat anställa medan numera handlar det i högre grad om hur väl organisationer lyckas integrera agenter i sina arbetsflöden. Monterro, Nordens största investerare inom Business-to-Business (B2B) mjukvara, arbetar just nu med denna utmaning genom att stötta sina 30 portföljbolag i att bli mer agentdrivna i sin mjukvaruutveckling. Det är dock svårt att veta om deras arbete leder till några faktiska produktivitetsvinster. Den här studien föreslår och utvärderar ett ramverk som kan användas för att mäta produktivitetseffekterna av det nya arbetssättet som kommer med agenter.

Ramverket består av fem mått som tillsammans täcker de två viktigaste dimensionerna av mjukvaruutvecklingsproduktivitet: hur snabbt utvecklingsarbetet blir gjort och vilken kvalitet det håller. Måtten bygger på så kallad metadata, det vill säga information och loggar kopplade till utvecklingsarbetet. För att utvärdera ramverket samlades metadata in från sex månaders utvecklingsarbete hos tre olika team inom ett av Monterros portföljbolag. Datan hämtades med hjälp av analysplattformen Solidafy, ett svenskt startupföretag som har bidragit till framtagningen av ramverket. Att basera ramverket på metadata är ett medvetet val som gör det enklare för flera portföljbolag att använda det utan att behöva öppna upp sin kodbas. I slutet av studien genomfördes intervjuer med utvecklare från de tre teamen, vilket gjorde det möjligt att kombinera den kvantitativa datan med utvecklarnas egna upplevelser av hur deras arbetssätt har förändrats med användningen av autonoma kodningsagenter.

De tre teamen visade sig ha mycket olika sätt att använda kodningsagenter. I det första teamet varierade användningen kraftigt mellan medlemmarna, vissa förlitade sig nästan helt på agenter medan andra inte använde dem alls. Det andra teamet hade ett mer enhetligt mönster där alla använde agenterna selektivt för uppgifter de ansåg att agenterna fungerade bra för. Det tredje teamet skiljde sig från de andra genom att aktivt investera i den infrastruktur som krävs för att agenterna ska kunna leverera kod som är anpassad till företagets kontext och teamets förväntningar. Denna spridning, både mellan teamen och inom dem, visar på att adoptionsmönster formas av såväl organisatoriska som individuella faktorer. Studien visar även att användningen av agenter ökade i samband med att nya versioner av de bakomliggande AI modellerna släpptes, vilket visar på utvecklarnas syn på tekniken och dess förmåga att bidra till utvecklingsarbetet är avgörande för adoption.

De uppmätta produktivitetsmönstren följde i stort sett respektive teams adoptionsmönster. Teamet som hade investerat mest i grundläggande infrastruktur visade tydligast positiv utveckling, både i hastighet och kvalitet. Teamet med enhetlig men selektiv användning visade också förbättringar, om än mer blygsamma. Teamet med spridd användning av kodningsagenter visade däremot fluktuerande resultat som var svåra att med säkerhet koppla till själva agentanvändningen. Den övergripande slutsatsen är att användning av autonoma kodningsagenter inte automatiskt leder till bättre resultat. Det avgörande är hur tekniken införs och att team tar sig tid att bygga upp den infrastruktur som gör att agenter kan bli mer träffsäkra i sitt bidrag till företagets utvecklingsarbete. Detta arbete kräver inte nödvändigtvis att alla individer i ett team är delaktiga i uppbyggnaden av struktur, men det krävs att de mer engagerade användarna bidrar till en gemensam infrastruktur som resten av teamet kan dra nytta av.

Acknowledgments

I would like to express my sincere gratitude to my supervisor at Uppsala University, Göran Lindström, for guidance and valuable input throughout the thesis. Additionally, I would like to thank my supervisor at Monterro, Nils Hulth for the effort to help me succeed in this project. He has provided me with great ideas for tackling the most challenging parts of the project, as well as introducing me to industry experts and stakeholders at companies within the Monterro portfolio. I also want to thank the industry experts that provided their perspective on adoption of agentic coding systems as well as all the respondents at the portfolio company including the Chief Technology Officer and the Team Lead for Platform Engineering. Lastly, I want to thank the team at Solidafy for letting me use their platform as part of this research initiative. The founder of Solidafy was very generous with explaining the technology behind the platform and collaborated closely to ensure the data and calculations were accurate and correctly understood in the context of my study.

Emelie Edberg,

Stockholm, May 2026

Glossary and Abbreviations

Agentic coding system - A system where one or more autonomous coding agents collaborate to solve coding tasks within the developer's environment.

Autonomous coding agent - An AI system that independently carries out coding tasks by reading codebases, planning modifications, writing code, running tests, and iterating based on feedback.

Deep learning - A branch of machine learning based on artificial neural networks structured in multiple layers, capable of detecting complex patterns in data.

Generative AI - Artificial intelligence capable of generating new content such as text, images, or code that resembles its training data.

Large language model (LLM) - Foundation model capable of language comprehension and generation, forming the basis of many generative AI systems.

Machine learning - A subset of AI in which models learn patterns from data without being explicitly programmed.

Markdown file - A text file using lightweight formatting syntax, often used to provide coding agents with instructions and context.

Measure - A high-level dimension that describes what is being assessed, such as development velocity or software quality.

Metadata - Data about data. In software development, information about coding activity such as pull requests and commits, rather than the source code itself.

Metric - A specific, quantifiable measurement used to capture a measure.

Pull request (PR) - A proposal to merge code changes from one branch into another, typically reviewed by colleagues before integration.

Repository - A storage location for all code, documentation, and history associated with a software project.

Skill - A reusable, file-based resource that provides a coding agent with domain-specific expertise, loaded on demand to turn a general-purpose agent into a specialist.

Source code analysis - A method where a tool inspects the source code itself to evaluate properties such as complexity and technical debt.

Ticket - An entry in a project management system describing a specific task to be carried out, such as a feature, bug fix, or investigation.

Table of Contents

1. Introduction

The future of software engineering is already unfolding with the rise of Artificial Intelligence (AI) teammates: autonomous, goal-driven systems that collaborate with human developers in real-world software work (Hassan, Li and Zhang, 2025). Central to this development are autonomous coding agents, agentic coding systems that operate directly within the developer's environment, capable of reading entire codebases, planning modifications, writing code, running tests, and iterating on their output based on feedback. These agents act as an extension of the developer, making decisions and interacting throughout the software engineering lifecycle in ways previously associated with human talent (Elkhalili and Otoum, 2026). In the literature, this paradigm shift is referred to as agentic software engineering or software engineering 3.0 (Elkhalili and Otoum, 2026; Hassan, Li and Zhang, 2025).

Autonomous coding agents promise significant productivity gains, making their implementation and scaling across software engineering workflows a top priority for many companies. Software as a Service (SaaS) companies are showing growing interest in agentic practices, driven by the highly dynamic nature of the software market. In this environment, the speed and quality of software delivery can determine whether a company succeeds or fails (Elkhalili and Otoum, 2026). The adoption journey is largely shaped by experimentation as software engineers usually begin by applying agents to isolated tasks before gradually extending them to larger parts of the workflow as the adoption matures (Hassan, Li and Zhang, 2025). As the technology is new and evolving rapidly, there is no established playbook for how to scale coding agents effectively or how to address the challenges that emerge along the way. One such challenge is organisational, as companies are trying to motivate software engineers to engage with the technology and adopt it into their daily work (Ahlawat et al., 2026). Another challenge involves providing agents with sufficient context to execute coding tasks with high quality. A general AI model can generate text, analyse images, and write code, but it cannot produce code that adheres to company-specific standards and policies. In fact, less than 1% of enterprise data is used to train public AI models today, which means that deep industry-specific knowledge, including proprietary data, embedded workflows, and organisational rules, must be explicitly provided to the coding agents before they can execute complex coding tasks effectively (IBM, 2025). As organisations navigate these challenges largely through trial and error, it remains unclear how their adoption patterns affect software development productivity.

This question is especially important for organisations overseeing multiple software companies who are currently integrating autonomous coding agents into software development workflows. One such company is Monterro, a Nordic private equity firm established in 2012 as a dedicated Business-to-Business (B2B) software investor with a portfolio consisting of 30 B2B SaaS companies. Monterro takes an active investment approach, which includes the establishment of an in-house AI team tasked with supporting portfolio companies in their AI adoption. One of the AI team's focus areas is

helping portfolio companies automate software engineering workflows using autonomous coding agents. As part of this initiative, the AI team organises coding workshops, facilitates knowledge sharing across development teams, and distributes a weekly newsletter with best practices and use cases for coding agents (Monterro, 2026). However, the AI team lacks an efficient framework for evaluating the true effects of their initiatives. Specifically, for measuring whether the portfolio companies' adoption of agentic coding systems in software development leads to productivity gains in their software engineering workflows.

While measuring software engineering productivity is a well-studied area in the literature, few studies have evaluated the impact of agentic coding systems on software productivity due to its recent emergence (Akour and Alenezi, 2025; Deissenboeck and Wagner, 2019). The limited research available in this area tends to rely on datasets drawn from open-source projects. Such studies may not capture the broader organisational context of SaaS companies, where coding practices, workflows, and developer roles are more dynamic and interdependent (Akour & Alenezi, 2025). Among the early contributions, Hassan, Li, and Zhang (2025) introduced AIDev, a large-scale dataset of pull requests (PRs) authored by autonomous coding agents. A PR is a proposal to merge code changes into a shared codebase (GitHub, 2026). In their study, productivity was primarily measured through PR acceptance rate and resolution time. Agarwal et al. (2026) proposed an alternative approach by studying how development velocity and software quality change in repositories over time, using a source code analysis tool to collect metrics that correlate with development velocity and code quality. Driven by the need to bridge the gap between adoption of agentic coding systems and measurable productivity gains, several startups have developed intelligence platforms for this purpose, some based on source code analysis and others on repository metadata such as PR activity and commit logs (Deventura, 2025; Swarmia 2026; Jellyfish 2026; Solidafy, 2026). Source code analysis refers to the process of examining a source code to identify security vulnerabilities, quality defects, coding standards, and architectural risks (Ruef, 2014). Metadata is in simple explanation described as "data about data", as it provides details about data that is separate from the content of the data itself (IBM, n.d).

1.1 Purpose and Research Questions

This study proposes and evaluates a framework for measuring the productivity impact of agentic coding systems on software development that can be realistically adopted by individual organisations. While this framework is applied within the Monterro portfolio, it can be adopted by any software company to track the development outcomes from adopting agentic coding systems.

Purpose:

The purpose of the study is to propose and evaluate a framework for measuring productivity outcomes related to the progress of adopting agentic coding systems in software engineering teams.

- RQ1: How well do software engineering productivity measures identified in the literature translate into a framework for evaluating the productivity impact of agentic coding systems?
- RQ2: What patterns of adoption regarding agentic coding systems emerge across different software engineering teams within a single organisational context?
- RQ3: How has software engineering productivity evolved in these teams during the adoption period, as measured through the proposed framework?
- RQ4: How do the measured outcomes align with developers' own descriptions of their agentic coding systems usage?

2. Autonomous Coding Agents: From Classical Software Engineering to Agentic Software Engineering, a Technical and Historical Review

The following chapter provides the necessary background for understanding the context in which autonomous coding agents operate. It begins with a technical and historical review of how AI has progressively transformed software engineering, from classical development practices to the emergence of agentic software engineering. This is followed by a description of the software engineering workflow, which serves as a reference point throughout the study as it is the workflow used by the three software engineering teams.

2.1 Classical Software Engineering

Classical software engineering represents the traditional era in which software was hand-coded using explicit rules and deterministic flows (Hassan, Li and Zhang, 2025). These workflows typically consist of sequential activities such as planning, design, development, testing, and deployment, where the output of one phase serves as the input to the next (Cocco et al., 2011; Balaji and Murugaiyan, 2012). During this time, programs were built and deployed in tightly controlled environments, requiring clear specifications and predictable behaviour. Building software was largely restricted to professional programmers with technical expertise and a comprehensive understanding of complex codebases, system architectures, and formal development processes (Hassan, Li and Zhang, 2025). Companies operating under SaaS models spent most of their financial resources on human labor and were limited by the number of engineers they could hire (Noda and Tacho, 2025).

2.2 Introduction to Artificial Intelligence

The early introduction of AI in software development during 2020 marked a major shift in the evolution of software development, gradually transforming the principles of classical software engineering (Hassan, Li and Zhang, 2025). However, the concept of AI has existed since 1956, when John McCarthy, a computer and cognitive scientist, stated that precisely described intelligence and learning could be simulated by machines (Dartmouth, 2026). Today, AI tools can perform a wide range of tasks and produce diverse outputs. A broad definition describes AI as a branch of computer science that creates systems and software capable of learning from experience and adapting to new information. Using data, algorithms, and computational power, AI can interpret complex information and make decisions with minimal human input (ISO, n.d.).

To understand how AI has shaped software development over time, a fundamental understanding of machine learning is required (Hassan, Li and Zhang, 2025). Machine learning, a subset of AI, has driven significant technological progress in the field by fundamentally shifting how intelligent systems are developed (Collins et al., 2021).

Machine learning models are built to mimic human cognitive abilities by processing vast amounts of information and utilising abstract representations to better understand incoming input (Janiesch et al., 2021). A model is trained on a dataset to obtain behaviour that reflects patterns within the data. During the training process, a learning algorithm continuously updates internal model parameters, improving the accuracy of predictions or decisions made by the model (ISO, n.d.). This approach enables computer systems to solve problems without being explicitly programmed to do so (Koza et al., 1996). For instance, instead of explicitly telling a computer system which customer characteristics indicate a certain customer preference or customer behaviour, the system learns these patterns by analysing sufficient training data (Sarker, 2021). Machine learning systems learn autonomously and refine their models over time as new information is encountered (Bishop, 2006).

Deep learning is a branch of machine learning inspired by the principles of information processing in biological systems. It is often compared to the human brain, where synapses act as connections between neurons and transmit signals. Similarly, deep learning relies on artificial neural networks that are hierarchically structured into multiple layers. This structure enables the computation of enormous multidimensional datasets significantly faster than humans can process them (Dutta and Sahoo, 2024). Through mathematical operations performed at each neuron, the layers collectively form a system that allows deep learning models to extract meaningful representations from raw input. As data flows through successive layers, the model detects increasingly complex patterns, enabling it to iteratively adjust and learn (Heinrich, Janiesch and Zschech, 2021). The concept of deep learning has further paved the way for deep generative models, which integrate generative and deep learning techniques (Dutta and Sahoo, 2024).

2.3 Generative AI and AI-Assisted Software Engineering

Generative AI is built on models capable of producing data that is indistinguishable from real data based on the information on which they were trained. While traditional AI systems are primarily designed to recognise patterns and make predictions, generative AI instead creates new content in various forms, such as images, text, audio, and code (Routley, 2023). Many generative AI systems are based on Large Language Models (LLMs). These models rely on a neural network architecture known as the transformer, first introduced by Google in the 2017 research paper “Attention Is All You Need.” The transformer represents natural language words as numerical vectors, allowing mathematical operations to be performed on them. The core functionality of LLMs is grounded in probabilistic modeling of word sequences. Transformer models generate text by predicting the next word or token sequentially while considering the full context of all tokens in the input sequence rather than relying solely on the previous token (Vaswani et al. 2017).

The first application of generative AI in Software Engineering was predictive coding, a form of context-aware assistance such as autocompletes and smart suggestions (Hassan, Li and Zhang, 2025). By using Integrated Development Environment (IDE) tools such as GitHub Copilot and Cursor, which operate synchronously within the editing workflow, developers may receive suggestions for the next line or block of code based on recent edits and surrounding context (Agarwal et al., 2026). At this early stage, these suggestions operated on a micro-scale and were mainly efficient for writing standardised, repetitive parts of the codebase that must be included in many places with little or no modification. As the technology matured, developers gained the ability to interact with generative AI applications using natural language while operating at the file or project level. This development marked a shift from micro-scale code suggestions to more comprehensive code generation. At this stage, AI was capable of generating full functions, tests, and boilerplate code, while human developers remained responsible for reviewing AI-generated code, integrating it, and revising AI suggestions (Hassan, Li and Zhang, 2025).

2.4 AI Agents and Agentic Software Engineering

In contrast to generative AI, agentic AI refers to systems that are designed to actively modify the state of their environment through perception, reasoning, and action. This changes the role of AI systems from conversational partners to active collaborators that can carry out end-to-end tasks (Buyya, G.R. and V., 2026). Although the concrete architecture of LLM-based agents differs, there is growing agreement on the basic mathematical structure and foundational building blocks (Buyya, G.R. and V., 2026; Anthropic, 2024). These design principles have also been articulated by the engineering team at Anthropic, one of the leading developers of advanced AI agent models (Anthropic, 2024). AI agents use a pretrained LLM as a general-purpose cognitive controller, meaning that the LLM acts as a reasoning engine built on external memory and execution modules (Celikyilmaz et al., 2023). Anthropic refers to this concept as “the augmented LLM”, where the LLM is enhanced with augmentations such as retrieval, tools, and memory. The models can use these capabilities to generate their own search queries, select appropriate tools, and determine what information to retain (Anthropic, 2024).

An agent typically initiates its process in response to a user’s command or through an interactive exchange that clarifies the task. Once the task is clear, the agent can plan and act autonomously while retaining the ability to return to the user for additional information or evaluative input when necessary. During task execution, the agent must obtain “ground truth” from the environment, such as tool outputs or code execution results, at each step to evaluate its progress accurately. Agents can then pause for human feedback at checkpoints or when encountering blockers. In summary, agents are LLMs that use tools based on environmental feedback in a loop (Anthropic, 2024). In an agentic coding system, several autonomous coding agents with different areas of expertise collaborate to solve a task (Gutowoska, 2026).

In software engineering, agents are referred to as autonomous coding agents and they have demonstrated remarkable potential, with their capabilities evolving from simple code completion to autonomous problem-solving (Anthropic, 2024). Hassan, Li, and Zhang (2025) describe this shift as a new era called agentic software engineering. Compared to earlier IDE tools, autonomous coding agents operate at the task level, they can read entire codebases, plan modifications, execute tools, refactor code, run tests, and review final changes (Hassan, Li and Zhang, 2024). As agents transform AI systems from conversational partners into active collaborators, the role of software developers also changes. Developers move from manually writing code to guiding agents by defining goals, setting permissions, and reviewing their output (Buyya, G.R. and V., 2026).

2.5 The Software Engineering Workflow

As autonomous coding agents become active collaborators throughout the software engineering workflow, it is important to understand what that workflow looks like. A commonly used software engineering workflow is therefore presented in Figure 1, followed by a detailed description of its key parts. Together, these provide a foundation for understanding the context in which repository metadata is captured and how productivity metrics in the proposed productivity framework are calculated.

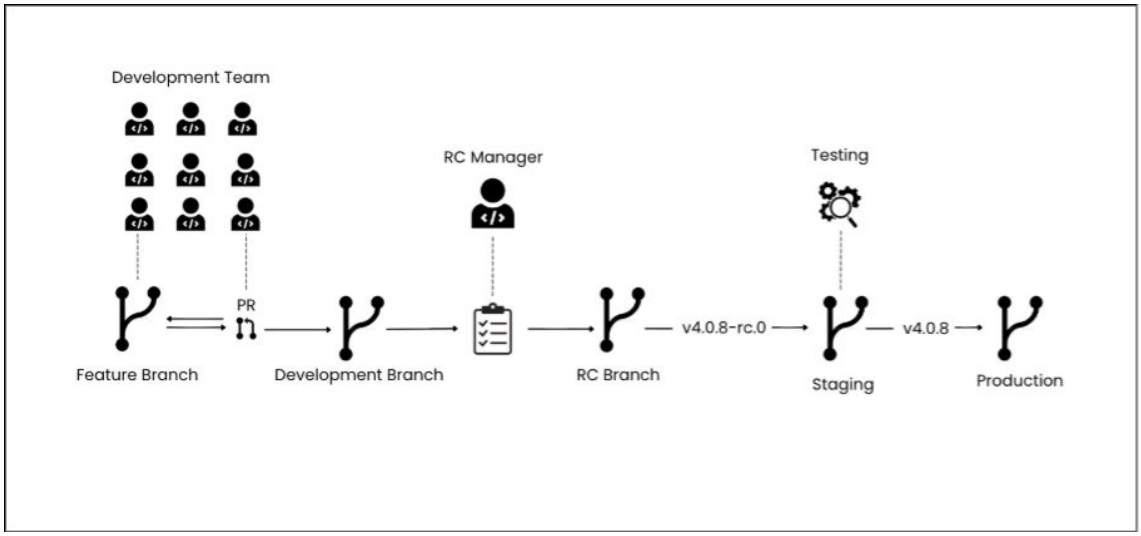

Figure 1. Overview of a Release Candidate (RC) workflow. Tickets are developed on isolated feature branches and submitted as PRs for review. Approved changes are merged into the development branch, bundled into a release candidate by the RC manager, validated in a staging environment, and finally promoted to production.

The software engineering workflow may vary between companies and teams. In Figure 1, a Release Candidate (RC) workflow is shown which is a commonly used approach in organisations with multiple interdependent services built by different teams (GitLab, n.d.). When coding in a company setting, a project management system is commonly used to reflect which tasks need to be completed in line with a product roadmap. These

tasks are described in tickets, containing the information software engineers need to complete a specific task. Many software engineering teams work in sprints, which are fixed time periods during which a selection of tickets is planned to be completed (Rubin, 2012).

When a software engineer is assigned to a specific ticket, they write the corresponding code on a separate feature branch, a temporary copy of the main codebase where a software engineer works on a specific task in isolation without affecting the shared code (Chacon and Straub, 2014). During this phase, the developer may verify their solution through simpler tests. When satisfied with the code, they submit it as a PR to a development branch, a shared branch containing the latest development code. At this point, colleagues review the PR and either request changes or approve it for merging into the development branch (Bacchelli and Bird, 2013).

Once the code is merged into the development branch, the workflow follows an RC deployment pattern. A release manager selects approved changes and incorporates them into an RC branch, from which a release candidate is created (e.g., v4.0.8-rc.0) and deployed to a staging environment. The staging environment mirrors the production infrastructure and allows for testing under conditions closer to production without affecting end users. If the RC passes all staging tests, a final release version is created (e.g., v4.0.8) and deployed to the production environment, making the changes accessible to end users. This separation between staging and production deployments is critical for maintaining software quality, as it ensures that only verified code reaches the production environment (Farley and Humble, 2010).

3. Theoretical Framework: Measuring Productivity in Software Engineering and Adoption Patterns of Emerging IT Technologies

Agentic coding systems may transform the processes through which software is developed, but they do not change the fundamental characteristics of what constitutes good software (SWEPR, 2025). Assessing how these agents affect productivity therefore does not require redefining what to measure. Coding agents have demonstrated remarkable results in increasing development speed but at the cost of quality. Therefore, it is important to have a robust productivity measure that can capture these limitations (Agarwal et al., 2026). The underlying dimensions of productivity remain well established in the literature and form the basis for analysis related to the development of the software engineering productivity framework. The following chapter outlines how productivity in software engineering has been defined and measured. The chapter further identifies the measures and metrics most relevant for evaluating the impact of autonomous coding agents.

As this thesis examines how agentic coding systems affect software engineering productivity, the theoretical framework also includes literature on the adoption of emerging technologies in organisational settings. This perspective helps explain how adoption patterns may vary between individuals working at the same firm, insights that are important when interpreting productivity outcomes from each team.

3.1 Productivity Definition

The focus of software process improvements is often to improve the productivity of software development. In general terms, productivity is defined as outputs divided by inputs. Within the context of software development, outputs can be either qualitative or quantitative, for example, functionality, lines of code, and implemented changes, whereas input can be described as the effort and resources needed for creating the output (Petersen, 2011). Deissenboeck and Wagner (2019) clarify that measuring input involves summarising the costs associated with software production, a task that is considered relatively straightforward. However, measuring the output is a more complex process where factors such as quality, timeliness, autonomy, project success, customer satisfaction, and innovation need to be taken into consideration. The software engineering community has so far been unable to develop a thorough understanding of productivity in software development and the factors that influence it (Deissenboeck and Wagner, 2019). As a result, there are no universally accepted methods for measuring productivity in software engineering, although numerous frameworks have been proposed over time (Forsgren et al., 2025). The following sections introduce these different approaches to measuring software engineering productivity, starting with the earliest and progressing to more recent developments.

3.2 Productivity Based on Technical Factors

Early productivity measurements primarily focused on technical factors such as Function Points (FP), Lines of Code (LOC), and code complexity (Ruhe and Wagner, 2018). The limitations of these methods are that they are often based on a few data points that tend to favour code volume and delivery speed over quality (SWEPR, 2025). Making developers aware of what metrics are used to determine the level of productivity can encourage undesirable behaviours. For example, measuring productivity based on the number of code reviews completed may result in rushed reviews that compromise code quality (GitHub, 2025). Forsgren et al. (2025) argue that an increased number of code reviews completed or similar activity metrics in isolation cannot be taken for productivity; it may reflect bad engineering systems or a tight deadline to meet a predefined product release. In contrast, it can also be a sign of better engineering systems, providing developers with the tools they need to do their jobs effectively, or better collaboration and communication among team members (Forsgren et al., 2025). This dilemma illustrates how pure technical metrics leave room for uncertainty, which motivated researchers and practitioners to explore the impact of non-technical or "soft" factors such as communication, collaboration, and team dynamics (Ruhe and Wagner, 2018).

3.3 Productivity as a Combination of Soft and Technical Factors

There have been several frameworks developed to capture a more holistic view of productivity in software development by incorporating both technical and non-technical factors. One of the most famous frameworks is SPACE, a multidimensional framework developed with the purpose of thinking rationally about productivity by including both soft and technical factors. The technical factors in the SPACE framework are centered around performance, activity, efficiency, and flow, while communication, collaboration, satisfaction, and well-being reflect the soft factors. A productivity analysis using the SPACE framework should include at least three of these factors, including both technical and soft ones (Butler et al., 2021). Forsgren et al. (2025) argue that adopting a more developer-centric perspective focused on developer experience enables deeper insights into the factors that shape developer productivity and creates opportunities for meaningful improvement. Developer experience encompasses how developers feel about, think about, and value their work (Greiler, Noda, and Storey, 2023).

When measuring productivity using technical and soft factors, surveys are often used to collect data, especially related to non-technical factors. To capture technical factors, data from end-to-end systems needs to be collected, often across different tools and teams. This requires software engineering teams to have fully instrumented systems (Forsgren et al., 2025). Such systems can be described as software environments in which every relevant tool and process is equipped with logging mechanisms that automatically generate machine-readable data about how the system is being used

(Google Cloud, n.d). In the absence of fully instrumented systems, organisations can use surveys to capture self-reported information about systems (Forsgren et al., 2025).

3.4 Measuring Productivity Using Machine Learning

The use of machine learning methods to measure developer productivity represents a new paradigm within software engineering productivity research. However, public research of this kind remains limited, but larger research institutions such as Stanford (SWEPR, 2025) are currently investigating these methods for a more accurate understanding of productivity. These initiatives are primarily motivated by the rise of AI in software engineering. SWEPR (2025) developed a machine learning model that assesses code commits based on implementation time, maintainability, and complexity, similar to a panel of 10-15 human experts. The research is based on Git historical data from over 600 companies, a total of 120 000 software engineers' coding work. Measuring productivity using machine learning aims to reduce noise between inputs (the resources used to build software) and outputs (the software produced) by providing objective data on both development speed and development quality. This method is so far the most accurate way of capturing developer productivity, but it is not suitable for industries to adopt, as it is time-consuming and requires a lot of training data (SWEPR, 2025).

3.5 Suggested Metrics

Returning to the definition of productivity as outputs divided by inputs, Agarwal et al. (2026) argue that development velocity is a key dimension of outputs. The literature offers several metrics for estimating development velocity, including lines of code, number of functional units, number of changelists, number of PRs, and number of commits (Canning et al., 2022; Lawrence, 1981; Agarwal et al., 2026). Since these measures tend to be highly correlated with one another, not all need to be included in a single productivity analysis. What matters is that at least one velocity-related metric is present, as it captures the rate at which development work is produced (Butler et al., 2021). However, as discussed in Section 3.2, relying solely on velocity metrics risks repeating the limitations of earlier approaches for measuring software development productivity, where volume and speed were measured at the expense of quality.

Another important dimension to include in the productivity measure is software quality. Software quality can be assessed through several dimensions, including user satisfaction, specification rigour, technical debt, and defect density (Agarwal et al., 2026). Denning (1992) suggests reframing the question "What is software quality?" to "How do we satisfy the customers of our software?", thereby relying on user satisfaction for determining software quality. Although this dimension would allow a comprehensive analysis, it cannot be collected through reliable and scalable methods (Agarwal et al., 2026). Similarly, specification rigour may be difficult to capture. Specification rigour refers to how accurately a software specification, the document that

describes what a system should do, reflects the actual underlying requirements (Farbey, 1990). Farbey (1990) conceptualises software quality more broadly as the gap between what is expected of a system and what is experienced once it is in use. Research suggests that a more rigorous specification increases software quality, as it leaves limited room for misinterpretation and reduces the risk of mismatches between expectation and experience. Assigning a quality value to the gap between what is expected of a system and what is experienced once it is in use is considered a difficult task that relies on subjective judgements (Agarwal et al., 2026; Farbey, 1990).

Technical debt and defect density are more accessible through metadata and source code. Agarwal et al. (2026) assessed software quality by focusing on technical debt, specifically investigating how the number of static analysis warnings, duplicate line density, and code complexity evolved over time. These measures were all derived from a tool that analyses source code at a repository level. Defect density, measuring the number of defects relative to the size of the codebase, has long been recognised as a key indicator of software quality (Grady, 1993; Fenton and Neil, 1999). Computing defect density requires access to data where confirmed defects are logged and linked to specific pieces of code or code releases (Morisio, Muhammad and Torchiano, 2013). Defect density is often associated with review effectiveness, since a more effective review will uncover more defects per line of code compared with a superficial review. Therefore, defect density is also possible to analyse from a predictive perspective to investigate what factors contribute to an effective code review. Research shows that code reviews benefit from smaller PRs under 300 lines of code as the reviewer can get overwhelmed with a large quantity of code as it is not possible to give the same attention to every line (Cohen, 2006).

3.6 Adoption Patterns of Emerging IT Technologies

Gallivan (2001) argues that studying the adoption of an emerging IT technology within an organisational setting requires thinking about adoption decisions both at an organisational level and at an individual level. The usage patterns of individuals within the firm are therefore shaped by a two-stage adoption process: primary adoption and secondary adoption. Primary adoption refers to the decision made by the management team, at an organisational or team level, to adopt a new technology. This decision is crucial for the technology to reach individual users within the organisational workflows, which constitutes secondary adoption (Gallivan, 2001).

The phase between primary and secondary adoption is critical and describes what happens from the point that the management team has decided to adopt an IT innovation until it is used by individuals within the firm. Factors such as managerial intervention, subjective norms, and facilitating conditions have a huge impact on individual adoption patterns (Gallivan, 2001). Managerial interventions are the actions taken and resources made available by managers to expedite secondary adoption (Deschamps and Leonard-Barton, 1988; Agarwal, 2000). Subjective norms are the individuals' perceptions or

beliefs of how others, including managers and colleagues, think about when to adopt an innovation and how much effort to undertake for learning it (Gallivan, 2001). Facilitating conditions are the remaining factors that make implementation more or less likely to occur, like specific attributes of the innovation, the organisational culture, and the workflow into which the technology is being introduced (Orlikowski, 1993).

Secondary adoption, covering adoption at the individual level, has been well discussed in the literature. Rogers (2003) argues that not all members of a social system adopt an innovation at the same time. A social system can be defined as a set of individuals who work together toward a common goal or engage in joint problem-solving. In an organisational setting or in a team, you can therefore expect a widespread when it comes to individual adoption patterns. Furthermore, Rogers (2003) identifies several characteristics of innovations that influence the rate of adoption. Some of these overlap with Gallivan's (2001) findings, such as social influence (subjective norms) and facilitating conditions. However, performance expectancy bring new perspectives to the theoretical framework. Performance expectancy describes the extent to which an innovation is perceived as superior to the alternative it replaces. If an individual believes that the new technology will help them perform their work better or more efficiently, they are more likely to adopt it (Rogers, 2003).

4. Methodology

This chapter presents the overall qualitative research strategy and the design choices behind the longitudinal case study, including the semi-structured interviews conducted with software engineers in the three teams. The methodology also includes a presentation and motivation for the selection of metrics in the proposed software engineering productivity framework. The chapter ends by reflecting on ethical considerations and the trustworthiness of the research.

4.1 Qualitative Research Strategy

To understand the study's research strategy, it is essential to examine the relationship between theory and findings. In social research, the relationship between theory and empirical data can take different forms, shaping how researchers draw on existing theories and engage with empirical observations. It is therefore important to clarify how theory relates to prior research and how it functions as a framework guiding the study (Clark et al., 2021). A deductive approach builds on pre-established knowledge, from which hypotheses are formulated and tested against empirical observations. In contrast, an inductive approach develops theory from patterns identified in the data collected during the research process (Bryman, 2016). Clark et al. (2021) also describe a third approach, abduction, which combines the reasoning of deduction and induction. An abductive approach grounds the study in a relevant theoretical framework while remaining open to revising theoretical assumptions considering empirical findings. Starting from a particular observation, the researcher moves back and forth between data and theory to find the most likely explanation for the observation (Clark et al., 2021).

When conducting research, two research strategies that have gained a lot of attention are the qualitative approach and the quantitative approach (Clark et al., 2021; Wilson, 2013). The overall research strategy chosen for the thesis is a qualitative research approach. According to Bryman (2016), a qualitative approach focuses on understanding phenomena through subjective experiences and meanings, often drawn on contextual data such as expert perspectives and experiences. Typically, data is collected through interviews and structured in a meaningful way to ensure thorough analysis (Yin, 2016). Although the overall research strategy is qualitative, the study also draws on quantitative data captured through five productivity metrics that make up the software engineering productivity framework proposed in the thesis. The productivity outcomes are interpreted qualitatively alongside the insights gathered through interviews with software engineers sharing their perspective on how usage of agentic coding systems has emerged in their respective team.

4.1.1 Exploratory Research Design

This thesis combines two separate fields of research: software engineering productivity research and research on the adoption of agentic coding systems. The two fields are in different stages of maturity, which has shaped the research design of the study. While software engineering productivity is a well-studied area, the adoption of agentic coding systems remains largely unexplored (Akour and Alenezi, 2025; Deissenboeck and Wagner, 2019). This difference makes an exploratory research design suitable, particularly for investigating research questions related to the adoption of agentic coding systems. Exploratory research aims to provide a better understanding of a relatively unknown problem, phenomenon, or behaviour by generating descriptive data (Cargan, 2007).

The difference in maturity between the two research fields has shaped how each part of the thesis has been approached. The productivity framework, consisting of five metrics, was developed through an abductive approach grounded in metrics identified in the literature while remaining open to refinement as limitations were discovered during the research process. The five metrics were calculated and collected using an intelligence platform called Solidafy. The platform is built on data pipeline practices, meaning that historical metadata from code repositories and project management systems can easily be accessed to perform various calculations. This made it possible to adjust the metrics throughout the study period, in line with the abductive approach. The proposed productivity framework was developed in close collaboration with the team at Solidafy, drawing on their knowledge and experience of the platform's functionality and limitations, while the author contributed insights on the selection of metrics based on the literature of software engineering productivity. The part of the study relating to the adoption of agentic coding systems followed an inductive approach, reflecting the limited research within this area. Instead of relying on established theoretical knowledge, an understanding of the adoption patterns was developed from interviews with industry experts and software engineers engaging with coding agents, together with the few early publications available. However, some general adoption theory is presented as part of the theoretical framework to provide the reader with a reference for how the adoption of previous emerging IT technologies has unfolded in organisational settings.

4.1.2 Longitudinal Case Study Design

To evaluate the proposed productivity framework, this thesis examined three software engineering teams at a Nordic B2B SaaS company. For each team, the analysis combined qualitative insights into coding agent adoption patterns with weekly productivity data collected through the productivity framework. A longitudinal study is defined as a research design that follows subjects over a specific period and collects data at regular intervals, allowing researchers to identify associations and observe patterns of change (Aljabi et al., 2023). What characterises this thesis as a longitudinal study is the weekly collection of productivity data over a six-month period. The

qualitative data, in contrast, were collected at the end of this period through a single round of interviews. This approach is particularly well suited for understanding how phenomena evolve over time, making it appropriate for studying how software engineering productivity changes with the use of agentic coding systems (Blankenagel and Hunziker, 2024).

The case study design is motivated by the early formative stage of agentic coding systems. Benbasat, Goldstein, and Mead (1987) argue that emerging research areas characterised by rapid technological change and innovation are particularly well suited to case study research. Such approaches allow researchers to learn directly from practitioners who are actively implementing the technology in real organisational settings (Benbasat, Goldstein, and Mead, 1987). Furthermore, Runeson and Höst (2009) claim that a case study is a suitable methodology for software engineering research since it studies contemporary phenomena in its natural context. Therefore, a case study design provides a strong basis for examining the impact of autonomous coding agents across the software development process. From an investor perspective, developing a framework that can guide and support portfolio companies through different stages of the adoption process is strategically important.

4.1.3 Productivity Metrics

The software engineering productivity framework proposed in the thesis is one of the main contributions of this study. The framework itself is presented in Section 5.3 as part of the study's results. However, since the framework was used to guide data collection throughout the study period, a description of its selection of metrics along with the motivation behind them is presented below to help the reader understand what type of quantitative data was collected.

For each team's repositories at week *t*, three metrics related to development velocity were collected.

- *PR Cycle Time:* How fast the PRs created by the team members go from opened to merged on average, measured in hours. Returning to the RC workflow described in Section 2.5, this metric captures the time from when a developer opens a PR until it is merged into its target branch. In the portfolio company's development organisation, PRs typically follow a journey from the feature branch to the development branch, from development to the RC branch, and from the RC branch to staging. The metric thus reflects how long PRs sit at each stage before being integrated into their respective target branches. Merges to the production branch are excluded as the final deployment step is typically governed by release schedules and organisational processes outside the control of individual software engineers. A shorter PR Cycle Time indicates that the team efficiently reviews, approves, and integrates code changes.

- *Output:* The number of PRs merged to production per active engineer averaged at the team level. A higher number of PRs merged into production indicates that the team is more productive.

- *Throughput:* The number of Jira tickets completed per active engineer, averaged at the team level. A higher throughput indicates that the team is more productive. Jira is the project management system used by the portfolio company and refers to the platform where tickets are created and managed throughout the software development workflow seen in Section 2.5 (Atlassian, 2026).

In line with the theoretical framework, development velocity is considered a key dimension of software engineering productivity. Both PR Cycle Time and Output have been used as productivity proxies for development velocity in earlier research (Google, n.d, Forsgren et al., 2025). Throughput was added to the framework as a complement to Output, as some engineering work does not involve active coding but rather investigation and validation of functionality and logic. Such work is typically logged as a Jira ticket describing the task, for example "Investigate reported performance degradation in payment service", if the payment service works without any degradation there will be no PR related to this work, meaning it would not be captured by the Output metric alone.

For each team's repositories at week *t*, two metrics related to software quality were collected.

- *PR Size:* Whether the team writes small, focused PRs or large ones, based on benchmarks for ideal PR size where smaller, focused PRs are considered beneficial. Specifically, PRs are categorised into five different size categories (XS, S, M, L, and XL) based on lines of code (LOC). The size categories are defined by the following thresholds: XS (≤10 LOC), S (11–50 LOC), M (51–200 LOC), L (201–500 LOC), and XL (>500 LOC). The metric captures the proportion of each size category among the team's PRs produced during a given week. PRs with only a few lines of code or with extremely many lines are excluded, as they can be considered outliers.

- *Review Intensity:* The number of reviews performed per active engineer, averaged at the team level.

As discussed in the theoretical framework, software quality is multi-faceted and can be assessed through several dimensions. In this study, we take the predictive perspective of defect density, investigating the size of PRs and code review contributions. PR Size captures how easy it is for reviewers to give attention to every line of code, with smaller PRs making it easier to identify potential bugs and maintain higher quality in the code

base (Cohen, 2006). The PR Size is especially relevant when assessing quality from AI driven development as coding agents tend to disregard code reuse opportunities, resulting in larger PRs with higher levels of redundancy (Chen et al., 2026). Furthermore, agents tend to combine multiple tasks within a single PR, increasing its size and reviewer cognitive load (Hassan, Li and Zhang, 2025). Review Intensity captures whether team members actively participate in the code review process, which has been shown to significantly reduce the likelihood of defects reaching production (Kemerer and Paulk, 2009). This metric has been criticised in the literature by GitHub (2025) and Forsgren et al. (2025) for encouraging rushed code reviews and for being unreliable as a productivity indicator when used in isolation. To mitigate these concerns, the productivity framework combines Review Intensity with four other metrics. In addition, the metadata covering the number of code reviews is historical. At the time the software engineers performed the reviews, they were not aware that the activity data would be used in a future study.

The five metrics measured on a weekly basis are visualised in this thesis using line graphs and stacked area charts. Three important decisions were made when processing the data that affect the overall interpretation of the graphs. First, some of the metrics are measured per active engineer. The framework counts an engineer as active in a given week if they have contributed to the team's repositories during that week. This means that even if a team member contributed only one out of five working days during a week, they are still included in the team average. Second, all line graphs are plotted using a four-week rolling average to provide a clear picture of the overall trend. Consequently, individual outlier weeks influence the values of the following weeks rather than appearing as isolated peaks or dips. Third, two weeks of data were removed at the end of December 2025 and the beginning of January 2026, as most of the development organisation was on holiday during this period. The low activity during these weeks would otherwise have affected the values of the following weeks due to the rolling average.

4.1.4 Interviews in a Qualitative Setting

Interviews can generally be categorised as structured or qualitative. In structured interviews, the interviewer follows a standardised set of questions based on a formal questionnaire. These questions are typically closed-ended, making structured interviews suitable for surveys aimed at representative samples and statistical analysis (Yin, 2016). In contrast, qualitative interviews are less structured and do not follow a formal questionnaire. Instead, the researcher adapts the questions based on the context, allowing respondents to reflect openly and elaborate on aspects they perceive as relevant to the research (Bryman, 2012). For example, an interviewer may choose to deviate from the prepared questions to explore relevant themes or respond to unexpected insights that emerge during the conversation. Qualitative interviews aim to capture participants' experiences and the deeper meanings they attribute to them by asking open-ended questions, making the format suitable for investigating industry-

driven initiatives where the literature is limited (Yin, 2016). Agentic coding systems in software engineering are primarily driven by industry adoption rather than academic research, further motivating the use of qualitative interviews to capture practitioners' firsthand experiences in this emerging area.

Qualitative interviews can either be unstructured or semi-structured. An unstructured interview can be compared to a normal conversation, with only a few prepared questions and the respondent speaking very freely about the research area. Semi-structured interviews, on the other hand, often include the researcher bringing an interview guide with questions covering the relevant topics in the field of research. This approach allows guided conversations while still providing room for participants to share their unique insights. For example, a respondent may answer a prepared question, but the researcher may end up asking follow-up questions. The lack of established theoretical knowledge when it comes to adoption of agentic coding systems requires a flexible interview format to capture insights that may not align with the picture provided by domain experts and early research. Therefore, the data collection covering the adoption patterns of coding agents was collected using semi-structured interviews.

4.1.5 Selection of Portfolio Company, Software Engineering Teams and Respondents

The thesis is highly dependent on collaboration of three main actors: Monterro, the portfolio company, and Solidafy. Monterro's portfolio currently consists of 30 companies, of which four were initially contacted with the purpose of taking part in the study. These four companies were selected based on their geographic location to facilitate on-site interviews, as well as their prior participation in AI initiatives driven by Monterro's in-house AI team. The selection was further driven by the portfolio company's current tech stack and its use of agentic coding systems. At the time of the study Solidafy supported integrations with GitHub, Jira, Claude Code (an agentic coding system provided by Anthropic) and GitHub Copilot (an agentic coding system provided by Microsoft). Portfolio companies using this combination of hosting platform, project management system, and agentic coding systems were better suited for the study as their data enables the full range of analysis that Solidafy offered. Based on this selection, one portfolio company was chosen for the study.

The initial contact with the portfolio company was driven by gatekeeper-mediated access, where the AI lead at Monterro provided referrals to relevant stakeholders at the firm. A gatekeeper is an essential mediator for accessing participants within social research and may be a person within the organisation who withholds access to people suitable for the research (Andoh-Arthur, 2019). At the portfolio company, the Chief Technology Officer and the Team Lead for Platform Engineering served as initial contacts. These contacts provided an understanding of the development organisation at the portfolio company consisting of several engineering teams out of which three were chosen for the study. In the study, the three teams will be referred to as: team A, team B, and team C. The selection of teams was primarily based on their willingness to

participate in the study. Furthermore, the selection considered demographics and engineering roles, including teams with engineers of varying seniority and nature of work. The initial contacts at the portfolio company served as a springboard by referring the researcher to these respondents. Clark et al. (2021) describe this approach as snowball sampling, highlighting its effectiveness in situations where researchers have limited professional contacts in the field. Table 1 presents interviews conducted with software engineers from the three teams.

Table 1. Compilation of interviews made with software engineers across the three teams sorted by teams and date, respondents have been anonymized according to the team and the date of the interview.

Respondent	Team	Date	Duration
Respondent 1	A	2026-04-10	35 min
Respondent 2	A	2026-04-13	25 min
Respondent 3	B	2026-04-14	40 min
Respondent 4	B	2026-04-16	35 min
Respondent 5	B	2026-04-17	30 min
Respondent 6	C	2026-04-14	50 min
Respondent 7	C	2026-04-16	30 min
Respondent 8	C	2026-04-17	30 min
Respondent 9	C	2026-04-29	40 min

4.1.6 Qualitative Data Using Thematic Analysis

As previously mentioned, few publications address how software companies adopt autonomous coding agents as part of their development workflow. Consequently, there is no established knowledge on the phases and challenges software engineering teams typically face as adoption matures. However, the topic is well discussed within the industry, and domain experts have developed an understanding of how the usage of coding agents evolves over time by studying real-company cases. These insights are shared in industry publications and early academic studies. Although different sources do not always provide a consistent picture of the adoption process, many findings overlap. Taken together, these insights informed the development of the interview guide together with the expert interviews conducted as part of this study. While this shared perception should not be treated as established academic knowledge, it represents the best available understanding of a topic that has not yet been fully examined in academic research. Therefore, these insights are presented as an introduction to the empirical findings in Section 5 rather than in the theoretical framework.

The interviews were firsthand performed in person, as it facilitates a more comfortable interview climate, allowing respondents to better express themselves openly (Bell, Bryman and Harley, 2022). All interviews were recorded with the permission of the respondent, and from these recordings, the interviews were later transcribed to ensure a structured and meaningful analysis using a thematic approach. A thematic analysis is a structured way of allowing the researcher to understand patterns in the data and interpret them as themes (Braun and Clarke, 2006). Additional relevant themes emerged through the open and exploratory nature of the interview guide, as respondents were encouraged to speak freely about how agentic coding systems are used in their daily work and the impact of these tools.

4.2 Ethical Considerations and Awareness

Taking ethical aspects into account is important to ensure responsible research. In line with the research design, this discussion must consider both the evaluation of the software engineering productivity framework and the research process investigating the adoption patterns of agentic coding systems. Collins, Hillman and Winthers (2009) argue that when one organisation depends on critical resources controlled by another, the resource provider may gain power and influence over the dependent organisation's behaviour and strategic decisions. Given Monterro's role as an investor, this potential power asymmetry was carefully considered in the research design. To mitigate any perceived pressure or obligation to participate, the four portfolio companies initially contacted were provided with a transparent explanation of the study's purpose and were explicitly informed that participation was entirely voluntary. Furthermore, measuring productivity in software engineering should be assessed at the team level, with developer data anonymised to protect individual privacy (Forsgren et al., 2025). Consistent with these guidelines, productivity metrics derived from Solidafy were analysed exclusively at a team level, ensuring that no individual-level data were examined. Furthermore, the decision to keep the portfolio company and its three teams anonymised was made in agreement with the portfolio company.

The qualitative data collection was conducted in accordance with the ethical principles outlined by Bryman (2016), including minimising potential harm, securing informed consent, safeguarding privacy, and avoiding deception. Before each interview, participants were informed that their participation was entirely voluntary and were provided with a clear explanation of the study's purpose, the interview topics, and the recording procedures. These measures were implemented to ensure informed consent and reduce any potential risk of harm. In the final report, all responses were anonymised and identifying details were removed. Throughout the research process, qualitative data were accessible only to the authorised researcher, thereby maintaining confidentiality. After the thesis was published, all audio recordings were permanently deleted.

4.3 Methodology Discussion

Discussing the credibility and quality of research is equally important across all kinds of studies. However, doing so is often more straightforward in quantitative research, while the quality of qualitative work tends to be harder to determine. Clark et al. (2021) suggest that alternative criteria such as trustworthiness and authenticity can serve as quality measures in qualitative research, and this methodology discussion is structured around trustworthiness.

The four key criteria that define trustworthiness are: credibility, transferability, dependability, and confirmability. Credibility was strengthened in this study through a systematic thematic analysis, which made sure that the patterns presented in the results originate from the data itself rather than from the researcher's assumptions. The selection of respondents further supported credibility, as software engineers across different teams, roles, and demographics were included to bring a broad range of perspectives into the analysis. Transferability concerns the extent to which the findings can be meaningfully applied to other contexts (Bryman, 2016). Since this thesis builds on a single case study within one development organisation, transferability is addressed by describing the case and its surrounding context in detail throughout the thesis, giving the reader the basis needed to judge whether the findings are applicable in other settings. Dependability concerns the consistency of the research process (Bryman, 2016) and was addressed by documenting all methodological choices, interview questions, and analytical steps in the final thesis. Confirmability was addressed by the author personally listening to and transcribing each interview recording, ensuring that the interpretations stayed close to the words of the respondents.

5. Results

This chapter presents the empirical findings of the study in three parts. Section 5.1 establishes the broader context of how software companies typically adopt agentic coding systems, drawing on industry expert interviews and early academic publications. Section 5.2 describes how the three studied teams have introduced and used coding agents in practice, based on the semi-structured interviews. Section 5.3 introduces the proposed productivity framework and presents the weekly outcomes for each of its five metrics across the six-month study period.

5.1 Introduction to the Empirical Results

Software companies that adopt agentic coding systems as part of their development workflow encounter various phases and challenges as the adoption matures. Gaining an understanding of this adoption process is necessary when investigating how engineering teams' usage of coding agents evolves over time. The following section provides such context by presenting a commonly understood picture based on two expert interviews conducted as part of this research, and the few early academic publications available. The expert interviews were conducted with Per Hassle, AI Specialist at Monterro, and Thomas Wahlström, Head of Software at a B2B software company outside of the Monterro portfolio that has made significant progress in adopting agentic AI for software development. It is important to note that this commonly held view should be treated as an indication grounded in subjective interpretations rather than as established knowledge.

5.1.1 Introduction of Agentic Coding Systems

Before the release of agentic coding systems, some software engineers were already familiar with prompting smaller solutions using generative AI. Drosos and Sarkar (2025) argue that this experience, formulating coding solutions in a natural language, is helpful when starting to experiment with autonomous coding agents. Wahlström describes how the typical first use cases of coding agents involve relatively isolated and well-defined tasks, where the scope and expected outcome are clear to the developer, such as writing unit tests or fixing simpler bugs. At the early stages of introducing coding agents, it is common to encounter skepticism among software engineers (Elkhalili and Otoum, 2026). According to Hassle, some engineers hold unrealistic expectations, applying coding agents to various complex tasks without providing sufficient context, which results in poor output. Instead, he suggests focusing on a single project, one that is sufficiently complex to effectively demonstrate the capacity of coding agents. The purpose of this strategy is to inspire software engineers to begin experimenting with the technology, while also illustrating the importance of establishing the infrastructure and organisational foundations required to effectively leverage agents in an organisational setting.

5.1.2 Implementing Autonomous Coding Agents as Part of the Workflows

To implement autonomous coding agents as part of software development workflows requires significant effort. Hassle compares a coding agent to a newly hired developer who excels at the most widely adopted programming languages and frameworks. The implication is that agents perform best when working with mainstream technologies and conventions, for instance using GitHub rather than GitLab or TypeScript rather than a less common programming language. Updating the codebase to align with such conventions is described as a legacy lift, a necessary step to realise the full potential of agentic coding systems according to Hassle.

Elkhalili and Otoum (2026) further describe the need to provide agents with relevant domain knowledge to complete coding tasks in line with company-specific requirements. This can be achieved through markdown files as well as through documentation embedded in the code itself (Elkhalili and Otoum, 2026). These observations are consistent with the experience of Wahlström, who describes how software engineers spend considerable time developing skills as part of the code repository to provide agents with the domain-specific context they require. Skills are reusable, filesystem-based resources that provide the agent with domain-specific expertise: workflows, context, and best practices that transform general-purpose agents into specialists. They load on demand and eliminate the need to repeatedly provide the same guidance across multiple conversations (Claude API docs, 2026). Developing a comprehensive set of skills is time-consuming, as Wahlström explains, because each area of the codebase requires its own tailored setup. For instance, frontend development, backend development, and architecture work each demand different skills to accommodate their distinct requirements and workflows. Allocating the time to build this structure may pose a challenge for software companies as the product deliveries are often prioritised, according to Wahlström.

Once a sufficient foundation of skills has been established and the codebase has been adapted to support the use of agentic coding systems, the benefits of the earlier investment begin to materialise. Both Hassle and Wahlström, independent of each other, share the experience of coding projects that managed to meet an earlier deadline thanks to the use of agentic coding systems. Agents do not necessarily change how software is built, but they accelerate the process, making it possible to complete more work in a shorter amount of time. This creates a reinforcing dynamic as agents reduce the time required for routine development work, engineers can reinvest the time into further refining skills, improving documentation, and adapting the codebase, which makes the agent usage more effective, unlocking further productivity gains (Rachitsky, 2026).

5.1.3 Scaling Adoption

While software development teams may experience increased productivity, extending these gains more broadly presents additional challenges. Wahlström observed that autonomous coding agents perform remarkably better when contributing to projects

within the existing product portfolio, while performing significantly worse on tasks related to new products, reflecting the limited contextual foundation in these areas. The process of building skills, documentation, and conventions must first take place before agents can perform at a comparable level on new products. This need for foundational investment extends beyond the codebase itself. Hassan, Li, and Zhang (2025) argue that realising the full potential of agentic coding systems requires a broader organisational change. Teams collaborating with software engineers need to adapt their workflows to integrate more seamlessly with the infrastructure built around the coding agents (Hassan, Li and Zhang, 2025). For example, Hassle describes how UX teams need to adopt naming conventions consistent with those defined in the skills, ensuring that their design decisions can be effectively interpreted and implemented by the agents. Scaling the adoption of agentic coding systems is therefore not only a technical challenge but an organisational one.

5.2 Agent Usage within the Three Teams

The following section describes how the usage of agentic coding systems has evolved across the three teams: team A, team B, and team C. The data presented draws on insights from qualitative interviews and cost data related to GitHub Copilot and Claude Code, the agentic coding systems used by the portfolio company. All three teams operate within the portfolio company's development organisation, meaning that their AI usage is shaped by the same environment and shared access to tools and initiatives. The development organisation consists of approximately 60 employees spread across multiple teams working on a B2B SaaS platform.

One of the teams, Technology Strategy, operates across the software development organisation and is responsible for driving AI initiatives within the development organisation, including organising AI workshops and ensuring that software engineers have access to agentic coding systems. The first AI coding tool introduced at the portfolio company was GitHub Copilot in 2024. At that time, usage was voluntary and framed as an experiment. Multiple respondents describe the initial message as "use it if you want" rather than a formal expectation. Several respondents note that as more capable AI models were released, the organisation began to communicate stronger expectations for adoption. At this time, the majority of the software engineers within the development organisation had access to GitHub Copilot. During 2025, Claude Code was gradually rolled out to engineers who expressed an interest in using it and by early 2026, the organisation had taken several steps to formalise the use of agentic coding systems. Usage had become an expectation across the development teams with automated agent-based code reviews enabled on all repositories. In addition, Technology Strategy began organising recurring AI workshops and created channels for sharing AI best practices among the software development organisation.

5.2.1 Agent Usage in Team A

Team A consists of three software engineers with distinctly different levels of usage of agentic coding systems. Respondent 1 uses coding agents for the majority of his daily work, to the extent that he claims writing almost no code himself. He describes a structured approach in which he first plans the solution, formulates it into a detailed specification with a description of the solution that he wants to code including necessary context, and then delegates the implementation to the agents. He estimates that most of his agent usage is dedicated to code generation, reflecting on a minimal setup that he prefers:

"I personally believe that MCPs and skills don't help that much. There are many reports suggesting that agents actually perform worse when you use skills and MCPs, even people I know working in AI companies don't use them. What can help is one or two sentences in a markdown file with instructions, like 'always use web search to check modern documentation', but in general, I prefer a minimal setup."
- Respondent 1

While Respondent 1 has been experimenting with AI since its introduction at the portfolio company two years ago, Respondent 2 describes a more gradual adoption journey. Six months ago, his AI usage was limited to simple questions as an alternative to searching for answers in developer forums. Today, Respondent 2 uses coding agents continuously throughout the workday, though his approach is more iterative, writing a simple prompt, evaluating the output, and refining through additional prompts. The third team member has so far not adopted coding agents.

"Six months ago, I was barely using it, it was more just asking simple questions instead of Googling. Now it's all the time for various coding tasks." - Respondent 2

Both respondents describe a clear shift around December 2025, with the release of more capable AI models. Before that point, AI usage was largely limited to autocomplete functionality and conceptual research. Since December 2025, the team's AI usage has intensified, and today Respondent 1 and Respondent 2 both describe how they use coding agents for full task execution. Both respondents also highlight the automated agent-based code reviews as a meaningful workflow change by complementing the manual reviews:

"We have automatic pull requests or reviews of all our pull requests. And I believe there are usually high standards on the reviews performed by AI... Sometimes I also use agents to ask for input or feedback before I submit a PR for review, so I would say we have a two-layer review process now with AI." - Respondent 1

Both respondents have been using GitHub Copilot throughout the entire study period. Since GitHub Copilot follows a seat-based pricing model where cost is fixed per user regardless of usage intensity, cost data alone cannot capture how actively the agentic coding system was used during the study period. The team's GitHub Copilot usage

patterns are therefore inferred from the descriptions provided during interviews rather than from cost data. One of the respondents gained access to Claude Code in January 2026. Claude Code is billed based on actual usage, meaning its cost data directly reflects usage intensity. The weekly Claude Code spending is presented in Figure 2. It should be noted that the respondent retained access to GitHub Copilot throughout this period, meaning the figure potentially only captures a portion of the respondent's total agent usage.

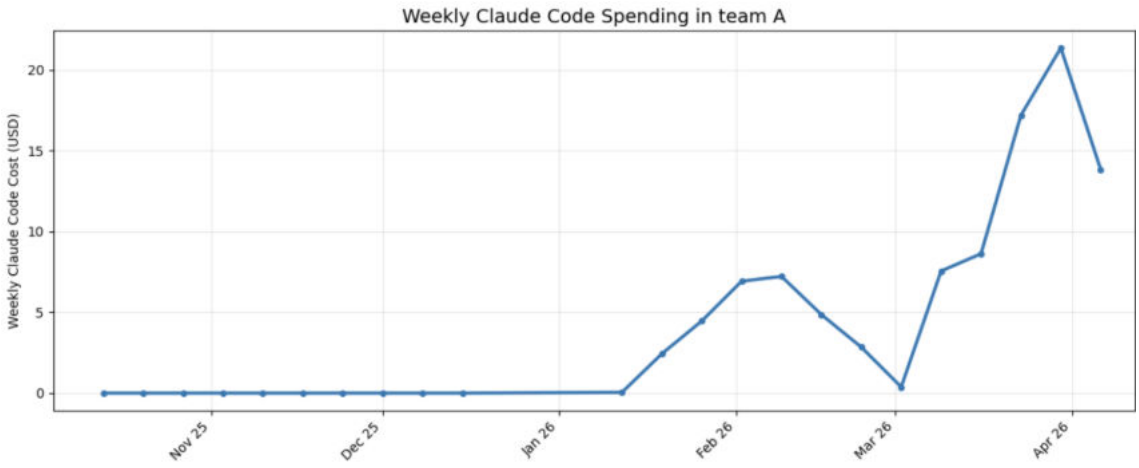

Figure 2. The weekly Claude Code spending in USD in team A from October 2025 - April 2026. Out of three team members, one used Claude Code during the studied period, meaning that the data reflect the usage of one individual.

5.2.2 Agent Usage in Team B

Team B consists of five software engineers with a more evenly distributed use of agentic coding systems across its team members compared to team A. The team lead, Respondent 3, focuses primarily on planning and ticket creation rather than active coding. He describes being able to fully automate parts of his workflow, particularly writing unit tests and drafting user stories based on source code or RFC (Request for Comments) documents. Three team members code full-time, while one person within the team divides his time between cross-team technical support and coding work.

The team's willingness to use the technology has been strong throughout the study period. However, respondents in team B describe a more incremental adoption pattern. They perceive their usage intensity to have remained relatively stable over the past six months. What has changed is not how much they use agentic coding systems, but how precisely they understand when to rely on them and when to write code manually. Accordingly, respondents describe agents as a complementary tool or pair programmer when actively coding, rather than a means of full automation. Developing an understanding of when to rely on coding agents has required experimentation. For example, Respondent 5 describes attempting to use agents for more complex work at the beginning of the study period, spending significant time explaining the desired

output and constantly receiving code versions that did not fulfil his expectations of what good code is. Respondent 3 reflects on a similar challenge:

"Sometimes you can get stuck in a different way, you get to prompt AI again and again and again and waste 30 minutes. By now we have learned which tasks AI actually speeds up and which ones it doesn't, that's made a big difference in how we use it today." - Respondent 3

As agents increasingly contribute to code generation, Respondent 4 anticipates that the team's work is more review-focused, requiring careful inspection of AI-generated code.

"Before AI agents, if a colleague had put up a PR, you could trust larger parts of it because it's a colleague who wrote it and you know they can write code. Then it was more about edge cases you thought of yourself. Now with a PR, you don't know if it's AI-generated or not, and I have to go through it line by line." - Respondent 4

Despite expressing this need for more careful inspection when colleagues write code using agentic coding systems, the team consistently uses the automated agent-based code reviews, which several respondents describe as an appreciated function. However, agent reviews are viewed as supplementary, human review remains mandatory for all the team's PRs. More recently, team B has connected their Jira board to GitHub, enabling agents to generate PRs directly from ticket descriptions, though Respondent 3 emphasises that this workflow remains experimental and that it is too early to determine how well it functions. Several team members recognise that context is necessary to improve agent precision. Team B has done some ad hoc work writing or generating markdown files for limited parts of the code repositories but has not made any further investment in providing agents with sufficient context. Respondent 4 acknowledges that when he writes documentation today, he does so with human developers in mind rather than coding agents.

The team's access to agentic coding systems has been limited to GitHub Copilot, with all team members having access throughout the entire study period. As previously mentioned, GitHub Copilot's seat-based pricing model means that cost data does not reflect usage intensity. The analysis of the team's agent usage therefore relies on qualitative data rather than on cost data.

5.2.3 Agent Usage in Team C

Team C consists of 12 people spread across various roles, including backend developers, frontend developers, a designer, a product owner, and testers. Together they collaborate on developing a new part of the platform. Focusing on the six months of the study, Respondent 7 describes a shift in the team's AI usage around the start of the study period.

"If you ask me, things changed dramatically six months ago. With the introduction of new tools, agents became smarter and faster." - Respondent 7

However, it is worth mentioning that Respondent 7 considers himself a heavy agent user compared to some of his team members, and not everyone in team C shares the same experience. Respondent 6 and Respondent 8 recalled that some team members dealt with a sense of uncertainty about the functionality of code written by agents and were not confident in taking ownership of such code. Respondent 9 reflects how the team has collectively introduced security mechanisms to minimise the risk of releasing code that introduces unwanted bugs, helping people feel more confident in relying on agentic coding systems.

"With the skills and approaches we're introducing, you don't just develop something with an AI agent and ship it straight to production. We have fallbacks in place, tests that have been written, pull requests that are reviewed by an AI agent, and [we have] the ability to review the code ourselves. For example, there's a skill that validates that no endpoints have been changed, which could otherwise create problems for existing clients calling them, and it flags any larger architectural changes before things move on to the next step. The more of these safeguards we introduce, the more trust we build in the system and in our ability to catch issues." - Respondent 9

All respondents from team C share the perception that agent usage within the team has intensified during the past six months. Some highlight the release of more capable models as a reason, while others point to a broader influence from the rest of the organisation. Respondent 6 believes that about half of the team currently use agents when coding, while Respondent 9 estimates a higher number of around 10 people.

Even though team C has made progress in delegating more coding work to AI agents, Respondent 9 believes the team has the potential to automate larger parts of its workflow. For all their repositories, they maintain a markdown file that provides relevant context to the coding agents and use the automated agent-based code reviews. Similar to team B, they have started assigning Jira tickets to agents to generate complete PRs. Respondent 9 recognises that this workflow could benefit from more precisely described tickets that outline the functional requirements and the context the agent needs to produce a useful PR. He believes this requires collaboration between product managers and software engineers, as software engineers often have the insight into where the technical solution needs to be implemented to meet the requirements while the product manager often plans future work.

In team C, the majority of the members had access to GitHub Copilot throughout the entire study period. Looking at the team's Claude Code usage, only one person had access to it at the start of the study. This number increased week by week, reaching four people towards the end of the six months. Similarly, the team's average spending related to Claude Code costs increased over time. This is shown in Figure 3, representing the weekly Claude Code cost per user over time. In team C, three out of the

four Claude Code users also had access to GitHub Copilot, meaning that they had the possibility to use both agentic coding systems simultaneously.

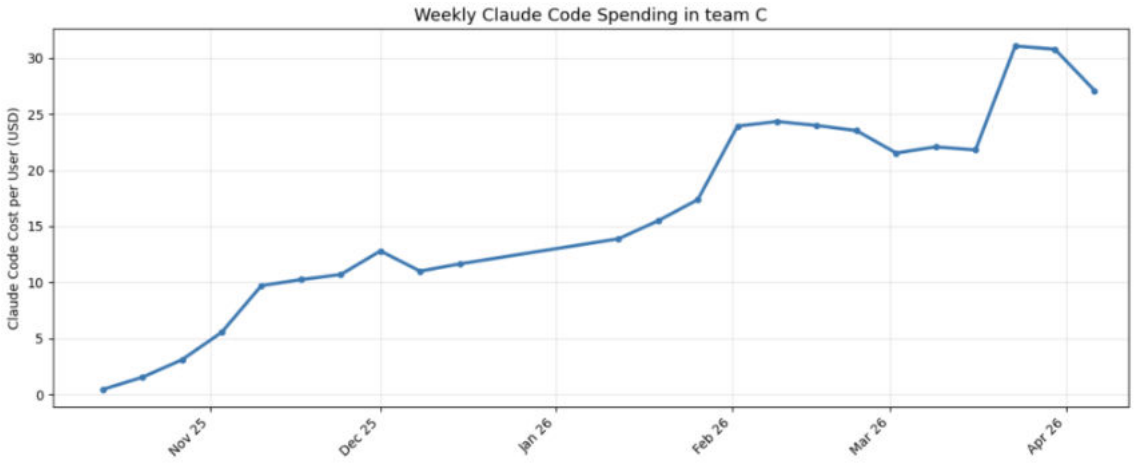

Figure 3. The weekly Claude Code spending in USD per Claude Code user in team C from October 2025 - April 2026.

5.3 Productivity Outcomes in the Three Teams

The following section presents the software engineering productivity framework proposed in this thesis, visualised in Figure 4. The results related to its five productivity metrics are then presented as absolute values, capturing individual team progress over the study period.

Software Engineering Productivity Framework

Development Velocity	Software Quality
PR Cycle Time = Average time from PR opened to merged, measured in hours	**PR Size** = Proportion of PRs in each size category (XS-XL), based on lines of code
Output = Number of PRs merged to production per active engineer, per week	**Review Intensity** = Number of code reviews performed per active engineer, per week
Throughput = Number of Jira tickets completed per active engineer, per week	

Figure 4. The software Engineering productivity framework proposed as part of the thesis. The framework is metadata based and covers the two key dimensions of software engineering productivity, development velocity and software quality.

5.3.1 Findings Related to Development Velocity

Returning to the key dimensions of productivity in software engineering identified in the literature, development velocity and software quality, three metrics were used to capture development velocity: PR Cycle Time, Output (number of PRs merged to its target branch per active engineer) and Throughput (number of Jira tickets completed per active engineer). Figure 5 presents how the average PR Cycle Time for each team has progressed week by week during the six months.

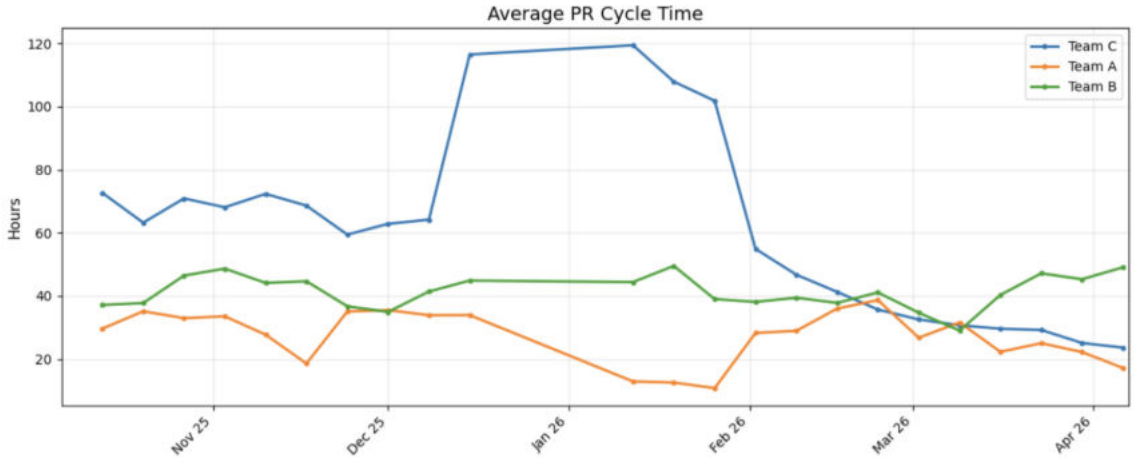

Figure 5. The progression of the average PR cycle time for the three teams covering the time period October 2025 - April 2026. Each data point represents the average time it took for PRs created by a specific team during a specific week to progress from opened to merged, where a lower PR Cycle Time is beneficial.

Team A consistently maintained a lower PR Cycle Time throughout the entire study period, with a notable dip in January 2026 before stabilising at around 20 hours towards the end of the study. Team B remained relatively stable throughout, hovering between 35 and 50 hours with no dramatic changes. Team C showed the most variation, starting at around 70 hours before increasing dramatically in January 2026, reaching a peak of approximately 120 hours, followed by a sharp decline to levels comparable to the other teams by March 2026. When examining the raw data, this observation can be explained by an outlier value in early January 2026, which inflated the following four data points due to the four-week rolling average used in the plotting method. This will be further discussed in Section 6.3, which covers the productivity outcomes across the three teams and their alignment with adoption patterns.

Figure 6 shows how the Output (number of PRs merged to its target branch per active engineer) has evolved week by week during the six months in the three teams.

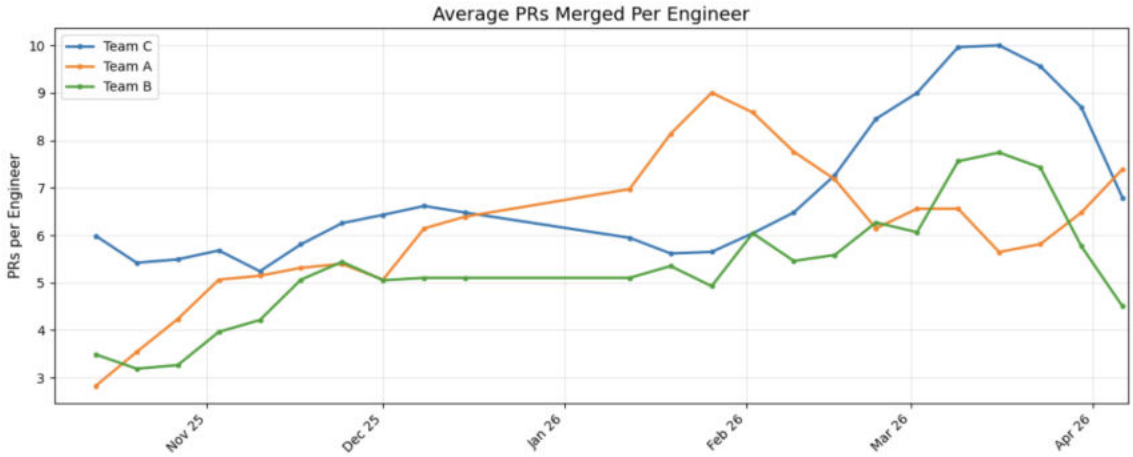

Figure 6. The progression of the average number of PRs merged into its targeted branch per active engineers for the three teams covering the time period October 2025 - April 2026. Each data point represents the mean of PRs merged per engineer in a specific team in a given week with higher values reflecting increased PR merge rates.

As shown in Figure 6, Output increased gradually from October 2025 for team A, reaching its peak in February 2026. The pattern does not indicate a consistent increase in PRs merged into a targeted branch during the study period. Further investigation into how the team's AI usage has evolved is therefore needed, particularly to explain the decrease observed in late February 2026 which will be addressed in Section 6.3. Team B shows a gradual trend from October 2025 to late December 2025 but maintained a more consistent number of PRs merged during January 2026, before experiencing an increase towards the end of the studied period. Team C merged a roughly similar number of PRs until late February, when their development speed increased dramatically.

Figure 7 shows how the Throughput (average number of Jira tickets completed per active engineer) has evolved week by week during the studied period in the three teams.

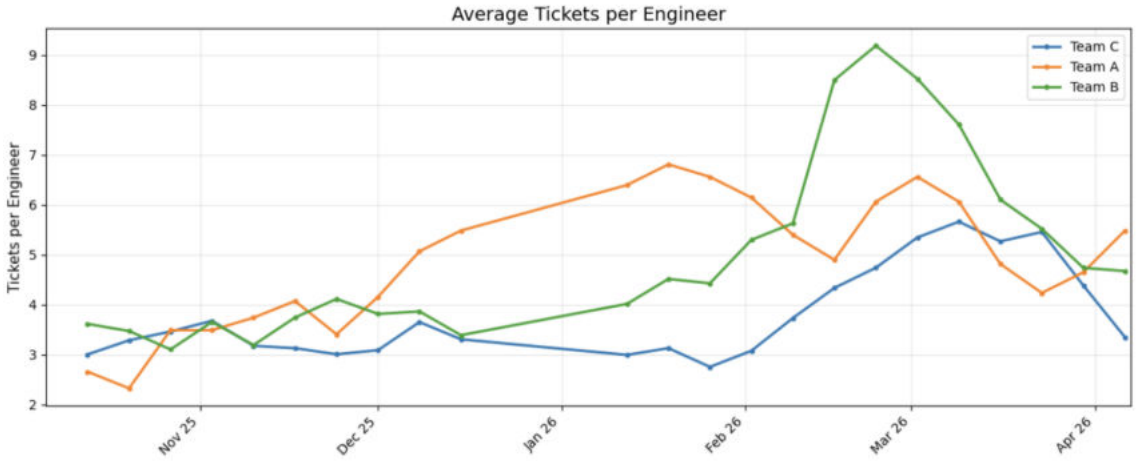

Figure 7. The progression of average number of Jira tickets completed per active engineer for the three teams covering the time period October 2025 - April 2026. Each data point represents the mean of Jira tickets completed per engineer in a specific team in a given week, with higher values reflecting increased task completion rates.

The number of tickets completed per active engineer follows three distinct patterns across all three teams, as shown in Figure 7. Team A shows a steady increase from October 2025, peaking in February before fluctuating towards the end of the study period. Team B follows an increasing trend but experiences a more dramatic spike in late February 2026, reaching the highest ticket completion rate of all three teams before declining. Team C remains relatively stable throughout the period with a modest increase towards the end of the study. Examining each team's respective adoption pattern together with the progression of average number of Jira tickets completed per active engineer will help explain why these distinct patterns occur. This discussion will be addressed in Section 6.3.

5.3.2 Findings Related to Software Quality

The metrics related to software quality examined in this study are PR Size (whether the team writes small, focused PRs or large ones, based on benchmarks for ideal PR size) and Review Intensity (average number of code reviews performed per active engineer). Figures 8, 9 and 10 present the weekly evolution of PR size distribution for each team during the study period. Overall, the results related to PR size distribution within each team show no indication that PRs have increased in size towards the end of the study period. This provides an important quality indicator that complements the findings related to development velocity in Section 6.3.

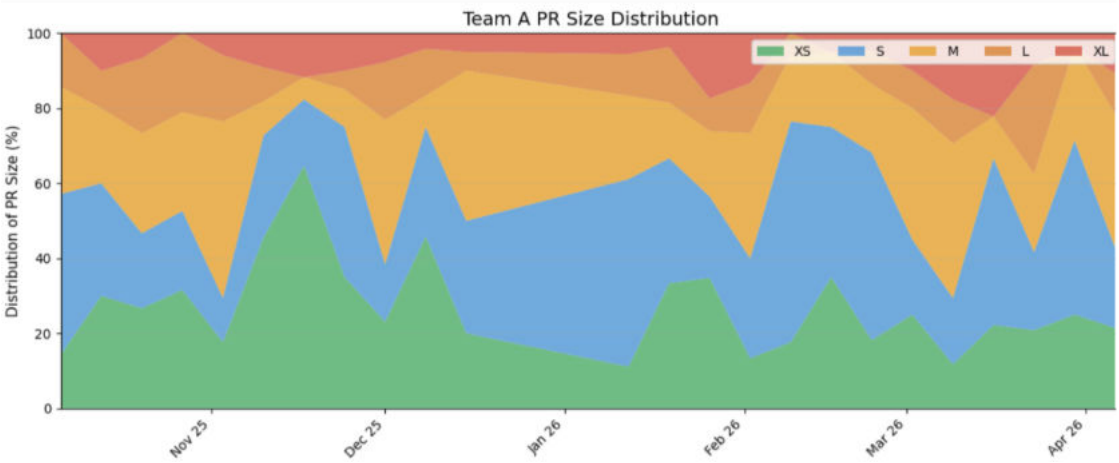

Figure 8. How the distribution of PR size in team A has evolved week to week covering the time period October 2025 - April 2026. PRs are categorised into five size categories based on lines of code (LOC): XS (≤10 LOC), S (11-50 LOC), M (51-200 LOC), L (201-500 LOC), and XL (>500 LOC).

Team A exhibits substantial variability in PR size distribution across the study period, with the proportion of smaller PRs (≤50 LOC) fluctuating considerably week-to-week. This pattern is exemplified during December 2025 - January 2026, when small PRs shift from representing 80% of submissions in early December to 40% by late December, before recovering to 75% in early January. Notably, while small and medium PR proportions vary substantially, larger PRs (>200 LOC) consistently constitute a minor fraction throughout the study period.

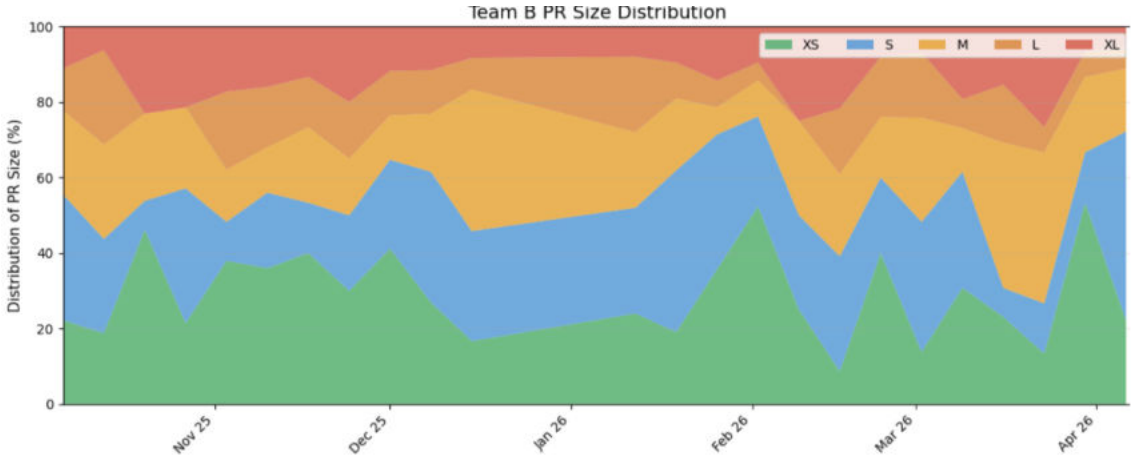

Figure 9: How the distribution of PR size in team B has evolved week to week covering the time period October 2025 - April 2026. PRs are categorised into five size categories based on lines of code (LOC): XS (≤10 LOC), S (11-50 LOC), M (51-200 LOC), L (201-500 LOC), and XL (>500 LOC).

Team B demonstrates week-to-week variability in PR size distribution throughout the study period. The team shows relatively stable patterns from October 2025 through January 2026, with smaller PRs (≤50 LOC) comprising 45-65% of submissions. A notable shift occurs in mid-February 2026, when small PRs spike to approximately 75% of total submissions, representing the highest concentration of small PRs during the study period, before fluctuating towards the end of the study. Regardless of these variations, larger PRs (>200 LOC) consistently remain a small fraction of total submissions.

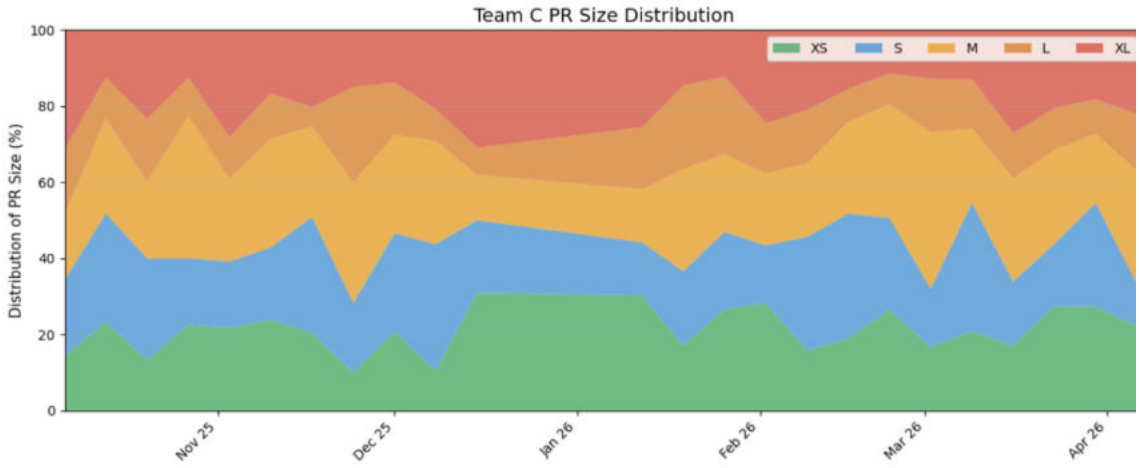

Figure 10: How the distribution of PR size in team C has evolved week to week covering the time period October 2025 - April 2026. PRs are categorised into five size categories based on lines of code (LOC): XS (≤10 LOC), S (11-50 LOC), M (51-200 LOC), L (201-500 LOC), and XL (>500 LOC).

Team C demonstrates a stable PR size distribution throughout the study period. Smaller PRs (≤50 LOC) consistently comprise 40-50% of weekly submissions, while medium PRs (51–200 LOC) maintain a steady 30-40% throughout most weeks. Larger PRs (>200 LOC) represent approximately 10-20% of submissions, which is slightly higher than the other teams but remains relatively constant across the entire period.

Figure 11 shows how the Review Intensity (average number of code reviews performed per active engineer) has evolved week by week during the six months in the three teams.

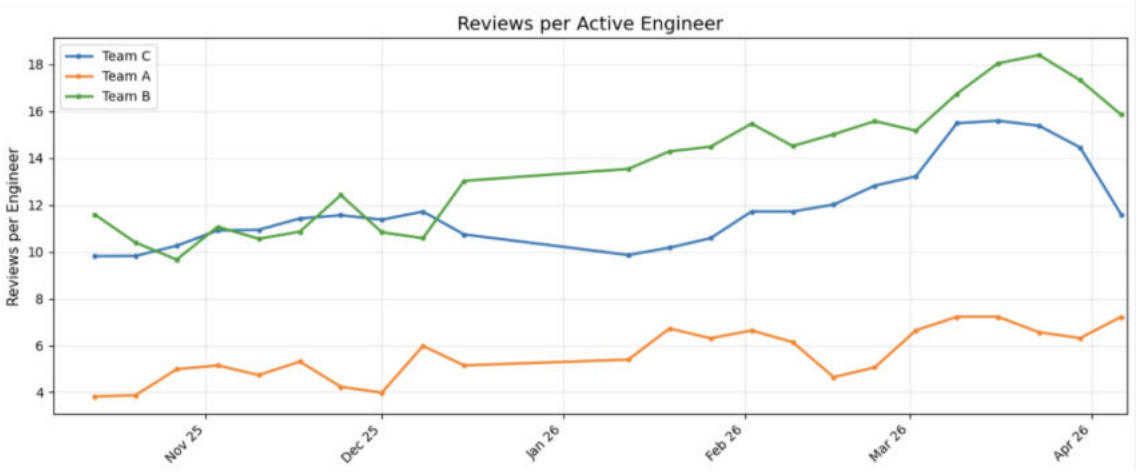

Figure 11: Number of reviews completed per active engineer for the three teams over the period October 2025 - April 2026. Each data point represents the average number of reviews for the specific team in a given week.

Both team B and team C began the period at around 10-12 reviews per engineer per week, a level that remained relatively stable until February 2026. From that point onwards, both teams showed a gradual increase, reaching 16-18 reviews per engineer per week. Team A maintained a significantly lower review rate throughout the entire period, with between 4 and 7 reviews per engineer, though showing a slight upward trend towards the end of the study. It may seem surprising that team A performs significantly fewer code reviews per active engineer than the other two teams. However, the more interesting point of analysis is how the progression of code reviews within each team has evolved over time with the use of agents, rather than how the teams compare to one another, as they may have different workflow patterns. The analysis covering these insights will be presented in Section 6.3.

6. Analysis and Conclusions

The previous chapter presented the empirical material in three layers: the broader adoption context of agentic coding systems, the qualitative picture of how each team works with coding agents, and the weekly outputs from the five productivity metrics. This chapter brings those layers together and analyses and interprets them in the order of the four research questions. Section 6.1 reflects on the framework itself (RQ1), Section 6.2 interprets the adoption patterns observed across the three teams (RQ2) and Section 6.3 examines the productivity trends the framework captured (RQ3) and how well those measured trends align with what developers themselves describe (RQ4).

6.1 Framework for Evaluating Agentic Coding Systems

The selection of productivity metrics in software engineering productivity research is rarely driven by what is theoretically optimal, it is constrained by what existing systems can capture. While the literature consistently identifies development velocity and software quality as the two core dimensions of software engineering productivity (Agarwal et al., 2026; Butler et al., 2021), the metrics used to capture these dimensions varies considerably across studies. As evident from the literature, identifying a metric or a combination of metrics that accurately represents software quality is what makes measuring software engineering productivity a complex task. Regardless of how metrics are selected, they will always constitute a simplification of a complex reality. For example, as discussed in the theoretical framework, Morisio, Muhammad, and Torchiano (2013) highlight defect density as a useful metric for software quality in organisations that systematically link bugs to specific code changes. In contrast, Agarwal et al. (2026) use static analysis warnings, duplicate line density, and code complexity to measure software quality, reflecting the capabilities of the source code analysis tool used in their study. This study faces the same constraint, the five metrics selected were shaped by what Solidafy could extract from GitHub and Jira metadata.

Measuring productivity through machine learning models trained to assess code commits similar to a panel of human experts as developed by SWEPR (2025) have the potential to provide a more objective measure of software quality without the need for simplification. However, the method is not yet viable for individual organisations to adopt independently, as it requires access to large-scale training data. This illustrates a broader challenge in the field of software engineering productivity, the methods best suited to capturing productivity are often the least accessible, while those that are more available tend to sacrifice measurement depth.

This trade-off is particularly important to be aware of when evaluating productivity from the use of autonomous coding agents. As stated by Chen et al. (2026) and Hassan, Li, and Zhang (2025), agents introduce quality risks as they tend to disregard code reuse opportunities, combine multiple tasks within a single PR, and generate code that may be syntactically correct but architecturally problematic. These tendencies suggest that

evaluating AI-driven development requires robust metrics for capturing software quality, metrics that metadata alone can struggle to provide. PR Size and Review Intensity used in this study can signal potential quality risks indirectly as Cohen (2006) suggests, but they cannot confirm whether code quality has actually deteriorated. A more direct approach would be to analyse the source code itself, tracking technical debt over time as Agarwal et al. (2026) did using source code analysis tools. However, this introduces a practical constraint that limits organisational adoption, it requires full access to the codebase. For many organisations, sharing proprietary code with a third-party intelligence platform is not feasible due to security and confidentiality concerns. Metadata-based approaches, by contrast, allow productivity analysis without exposing source code, making them more widely applicable despite their measurement limitations. This study reflects that trade-off; the five metrics selected rely on GitHub and Jira metadata rather than source code, prioritising broad feasibility and data security over measurement depth. The framework proposed is therefore not the most theoretically robust option, but it is one that organisations can realistically implement without compromising code confidentiality.

6.2 Patterns of Adoption Within the Three Teams

Following Gallivan (2001), the analysis distinguishes between primary adoption (the management team's decision to introduce the technology at an organisational or team level) and secondary adoption (how individual engineers actually use it). The first part of this section addresses primary adoption at the portfolio company while the following parts cover the three teams' adoption patterns.

6.2.1 Primary Adoption

As Gallivan (2001) argues, understanding the adoption patterns of an emerging IT technology within an organisational setting requires analysing both primary and secondary adoption. At the portfolio company, it is evident that the management team did not express any formal expectations of coding agent usage until early 2026. The primary adoption decision can therefore be associated with the turn of the year, even though some software engineers had been using agentic coding systems before this point. The longitudinal study design adopted in this thesis captures the critical phase that follows, the transition from the management team's decision to adopt the technology, to its actual use by individual software engineers.

The theoretical framework identifies three factors driving secondary adoption: managerial interventions, facilitating conditions, and subjective norms. The recurring AI workshops organised by Technology Strategy can be seen as managerial interventions intended to expedite secondary adoption. Furthermore, the fact that Technology Strategy has created channels for sharing best practices and is responsible for ensuring that all software engineers within the organisation have access to agentic coding systems can be understood as facilitating conditions. Subjective norms are more challenging to identify in the qualitative data. However, several respondents highlight

that they perceive usage as less optional than before January 2026 and that AI has become a recurring topic in both formal and informal settings in the office, which may nudge individual software engineers towards adoption.

Rogers' (2003) claim that individuals within a team adopt innovations at different times becomes evident in this study, particularly in team A and team C. It is therefore relevant to examine the adoption pattern of each respondent in order to gain a realistic picture of each team's overall adoption pattern. At the same time, the adoption journey described by the industry experts Hassle and Wahlström suggests that a meaningful analysis of agentic coding system adoption must go beyond when individuals begin using the technology. The analysis will therefore also examine how deeply coding agents are integrated into the teams' workflows.

6.2.2 Team A – Individually Driven Adoption

Team A is the smallest of the three teams and shows the widest within-team spread. One of the three engineers has not adopted coding agents at all, an example of the scepticism Elkhalili and Otoum (2026) describe as common in early adoption. The two adopters both describe a clear shift in usage intensity around December 2025 tied to the release of more capable AI models, exactly the kind of change in perceived relative advantage that Rogers (2003) identifies as a driver of accelerated adoption. As software engineers tried the new AI models released and realised that they performed significantly better compared to the previous models, they became more motivated engaging with coding agents.

Respondent 1 and Respondent 2 take different approaches to collaborating with coding agents to complete more complex work. Whereas Respondent 2 takes a more iterative approach, refining the output through additional prompts, the approach described by Respondent 1 is particularly noteworthy. Respondent 1 writes a detailed specification of the intended functionality requirements, including general context to guide the agent. This may reflect a lack of markdown files and skills containing the relevant domain context that Wahlström describes as necessary for agents to execute tasks effectively in line with company-specific requirements. Respondent 1 appears to compensate for missing repository-level context by providing it ad hoc within each specification. While Wahlström argued that such foundational work is often deprioritised in favour of product deliveries, Respondent 1 offers a different explanation. He is simply not convinced that markdown files and skills meaningfully improve the quality of agent contributions, based on what he has read on developer forums and heard from heavy AI users in his personal network. This points to an important insight in the adoption journey, there is no single accepted method for how to use coding agents effectively, and individual team members' perceptions and prior beliefs can have a substantial influence on team adoption patterns.

Taken together, the adoption pattern in the team A is best characterised as individually driven. One engineer remains a non-user, while the other two have moved beyond the

typical first use cases of isolated, well-defined tasks, applying agents to more complex work but in different ways. The case of team A illustrates how adoption patterns within a single team can be shaped less by organisational direction and more by individual preferences and prior beliefs.

6.2.3 Team B – Broad but Shallow Adoption

Team B presents a remarkably different picture to team A and thereby brings new findings to the analysis. The team's willingness to adopt coding agents has been strong throughout the studied period, and unlike team A, all software engineers have adopted coding agents to some degree. At the same time, the respondents consistently describe coding agents as a selectively chosen pair programmer rather than as a means of fully automating their work, which suggests that the team's adoption is characterised by breadth rather than depth.

Both Respondent 3 and Respondent 5 reflect on attempts to apply coding agents to more complex tasks at the beginning of the study period, spending significant time iterating with the agent only to receive code that did not meet their expectations. This aligns with what Hassle describes as a common early-adoption pitfall, applying agents to complex tasks without providing sufficient context. However, whereas Hassle's framing implies that the solution is to invest in the foundations needed to handle complexity, such as markdown files with necessary context and codebase conventions, team B has drawn a different conclusion. Rather than building the foundations that would allow agents to handle more complex work, the team has narrowed the scope of when they rely on agents. The team has done some ad hoc work writing markdown files for limited parts of their repositories, but Respondent 4 acknowledges that he still writes documentation with human developers in mind rather than with coding agents. Most likely coding agents can benefit from such documentation as well, but this indicates that the team does not reflect on providing agents with necessary context in a way that Wahlström describes as necessary for realising further productivity gains.

The team composition consisting of five team members, including a dedicated team lead, may impact the team's adoption patterns. Compared to team A consisting of three engineers with various adoption patterns, team B presents a more shared approach to how coding agents are used. In a team of five with a coordinating team lead, shared norms regarding the use of coding agents appear to emerge more naturally than in the smaller team A, where individual voices carry greater weight in shaping the collective adoption pattern. However, the team lead stands out from the shared approach by describing how he is able to fully automate parts of his own workflow, such as writing unit tests and drafting user stories from source code or RFC documents. This may reflect that the tasks typically performed by the team lead are easy to execute with agent involvement, or simply that he has experimented with agent usage finding ways to optimise his daily work. On a team level, the recent GitHub and Jira integration, allowing agents to generate a PR directly from a ticket, may be seen as a way to automate workflows. Even though it may seem like a small step, it may help the team to

explore new ways of working with coding agents that go beyond their current selective use, potentially shifting their adoption pattern over time.

In summary, the adoption patterns of team B appear to be more uniformly distributed among its team members and have remained relatively stable throughout the study period. What has changed, in the words of the respondents, is not how much they use agents but how precisely they understand when to rely on them. While this represents genuine learning, it also diverges from what industry experts suggest. Hassle argues that complex tasks can be handled by agents if the right foundations are in place, whereas team B has instead concluded that certain tasks are simply not well suited for agent involvement.

6.2.4 Team C – Foundationally Invested Adoption

Team C differs from the other two teams in both size and composition. The team consists of twelve people across multiple roles, including backend and frontend developers, a designer, a product owner, and testers. The team therefore offers a particularly relevant case for studying workflow adaptation around coding agents. Hassan, Li, and Zhang (2025) argue that the full potential of autonomous coding agents can only be captured when the functions collaborating with software engineers such as design, product management, and testing, adapt their workflows to the infrastructure built around these agents.

Examining the qualitative data from team C, it presents a conflicting picture of the team members' adoption patterns, highlighting that not everyone within a team adopts an innovation at the same time. While team B displayed a more shared approach to agent usage, the adoption patterns observed in team C do not follow the same pattern. This suggests that larger teams do not automatically lead to a more equal usage of coding agents within the team. Even though the qualitative data reveals a conflicting picture about how widely agentic coding systems are used among team members, cost data from GitHub Copilot and Claude Code suggest that adoption is relatively wide among users. Respondent 6 recalls that some team members initially dealt with a sense of uncertainty about the functionality of agent-generated code and were not confident in taking ownership of such code. This may reflect the kind of scepticism that Elkhalili and Otoum (2026) describe as common during the early stages of introduction. But at the same time, it could also be the case that these engineers are genuinely willing to adopt agentic coding systems but are concerned about the potential consequences of doing so.

Compared to the other two teams, team C stands out for its investment in skills and markdown files. According to Wahlström, this is the kind of foundational work that needs to be done to fully implement coding agents as part of the development workflows, providing agents with the domain-specific context they need to produce code aligned with company-specific requirements. The use of skills described by Respondent 9 is particularly noteworthy, as it extends beyond what Wahlström

describes. In team C, skills are used not only to provide context for code generation but also as safeguards, validating that no endpoints have been changed, flagging larger architectural changes, and giving engineers the confidence to take ownership of agent-generated code before it reaches production. This dual purpose, both generating code in line with company requirements and managing risk, suggests that team C has made significant effort to explore their use of agentic coding systems.

A factor worth noting is that team C works on a relatively new product within the broader platform. Wahlström observed that coding agents tend to perform significantly worse on tasks related to new products, reflecting the limited contextual foundation in these areas. However, none of the respondents from team C describe experiencing such limitations, which may indicate that the team's investment in skills and markdown files has been sufficient to compensate for the more limited conventions of a newer part of the codebase. Another possible explanation is that the team members have simply no reference point, while Wahlström works with various products, both existing and new products in his role as Head of Software.

Returning to the composition of the team, including various roles, Respondent 9 reflects the potential for closer collaboration between product managers and software engineers. He recognises that more precisely described tickets, written with the agent in mind, could improve the quality of agent-generated PRs. This observation aligns directly with what Hassle describes as a broader organisational change related to coding agents. While team C has not yet implemented such cross-functional adaptations, the recognition of the need for them suggests that the team is beginning to think in these terms.

While team C appears relatively mature in its adoption, with notable investments in foundational work and broad usage across the team, the larger team size makes its collective usage pattern harder to capture in a single description. Varying individual approaches emerge within the team, some engineers using agents in ways that Wahlström and Hassle describe as harder to obtain, others in more limited ways. What distinguishes team C from the smaller teams is how more progressive AI users appear to lift the team's overall adoption by contributing to the foundational infrastructure that everyone can draw on. This points to a team-level dynamic where individual leadership in agent usage translates into collective capability through shared structural investments.

6.3 Productivity Outcomes Across the Three Teams and Their Alignment with Adoption Patterns

The three teams, each representing a distinct adoption pattern, provide relevant cases for examining whether the productivity gains attributed to agentic coding systems in the literature actually materialise in practice. Returning to the productivity framework with its two key dimensions, development velocity and software quality, the five metrics covering these dimensions and the team's adoption patterns are analysed together.

Team A shows the lowest PR Cycle Time throughout the period, despite the team having one member who has not adopted agents at all. What is more interesting though is the development of the PR Cycle Time over the studied period, which remains roughly unchanged except for the apparent deviation during December 2025 and January 2026. The gradual increase in PRs merged per engineer and tickets completed per engineer accelerating clearly around December 2025, aligns with the respondent's perception that AI usage intensified around this time as more capable AI models were released. However, both figures show a peak in February 2026 following a subsequent decline in these metrics which raises the question of whether coding agents increase development productivity. Here it is important to take into consideration that team A consists of only two active agent users, meaning the data is heavily influenced by the development patterns of individual software engineers. For example, the team's Claude Code cost dropped to nearly 0 USD in March 2026, which may indicate that the engineer with Claude Code access was not actively coding during this period. Individual workload patterns, such as one engineer focusing on a particularly complex task for an extended period, can influence the team's average metrics, limiting the conclusions that can be drawn about agent-related productivity changes.

Team A shows no increase in PR size during the study period and generally maintains a high proportion of smaller PRs with only a few larger ones, which suggests that software engineers within the team have good conditions to perform effective code reviews. The considerably low Review Intensity throughout the period is likely a consequence of the team's workflow patterns. Having both a shorter PR Cycle Time and a lower Review Intensity throughout the study period indicate that the team has shorter release practices compared to the other two teams. Whether the observed productivity changes reflect agent adoption or other workflow changes cannot be determined from the data alone, which is a fundamental limitation of metadata-based productivity analysis. This interpretive challenge applies to all three teams but becomes particularly significant when analysing team A, where the adoption pattern is characterised as individually driven. For this team, the absence of a shared approach to how agents are used makes it harder to attribute productivity changes on a team level specifically to agent usage, as fluctuations in the metrics may equally reflect the work habits of a single engineer.

Team B, characterised by a more uniform adoption pattern that has remained relatively stable throughout the study period, shows development velocity and software quality progression broadly in line with these findings. The team appears to handle more work without necessarily finishing each task faster. PR Cycle Time remains relatively constant, while PRs merged per engineer increase gradually and tickets completed spike in late February 2026. It should be noted that the pronounced spike in the number of tickets is largely driven by a single week with 16.5 tickets per engineer, which inflates the following four data points due to the plotting method with a four-week rolling average. Furthermore, only 3 of the team's five members code full-time, which may understate the per-engineer metrics relative to the team's actual coding capacity. The

PR size distribution offers no indication that PRs have become larger towards the end of the study compared with earlier months, which suggests team members retain a similar opportunity to review the code effectively, minimising the risk of introducing bugs. The gradual increase in Review Intensity likely reflects the slightly higher development velocity and illustrates the more review-focused work that comes with coding using agents, as described by team members. Taken together, these patterns point toward gradual improvements in software development productivity. The gradual nature of productivity improvements in team B is consistent with the qualitative finding that the team uses coding agents as a selective pair programmer rather than as a means of fully transforming its development workflows.

Team C shows similar productivity patterns to team B. The team experienced a decrease in PR Cycle Time over the studied period, with the temporary increase around the beginning of 2026, which can be partly explained by examining the raw data. An outlier week with an average cycle time of 239 hours inflates the surrounding data points due to the plotting method with a four-week rolling average, producing a similar effect as the outlier observed in team B. The outlier may be an effect of calendar-related events, as the team left for the Christmas holiday during this period. PRs opened before the holiday likely remained open during the leave before being merged, increasing the average PR Cycle Time. The patterns observed in PRs merged and in tickets completed, with both metrics showing a more pronounced increase from February 2026 onwards correspond with the increased Claude Code spending for the team and the release of more capable models. Taken together, team C produces more code at a faster pace towards the end of the study compared with earlier months.

Looking at software quality, the absence of any notable shift in PR size distribution is particularly relevant. Even though the team has increased its development velocity by merging more PRs and completing more tickets in a shorter amount of time with the help of agents, the PRs do not increase in size, which is a healthy sign. The gradual increase in Review Intensity from February 2026 likely reflects the simultaneous rise in PRs merged to a targeted branch and is consistent with the respondents' observation that AI-generated code requires more careful inspection, with team members performing manual reviews as a complement to the automatic code reviews performed by agents

7. Discussion

This chapter draws together the findings of the study and discusses the implications for Monterro and other organisations seeking to evaluate the impact of agentic coding systems. Furthermore, the chapter presents the limitations of the study and proposes directions for future research.

7.1 Key Findings and Implications

The purpose of this study has been to propose and evaluate a framework for measuring productivity outcomes related to the adoption of agentic coding systems. To achieve this purpose, the study translated software engineering productivity metrics identified in the literature into a framework that can be realistically adopted by individual organisations. The framework was evaluated by applying it to three software engineering teams within a single organisation over a six-month period, analysing outputs from productivity metrics with qualitative insights into the teams' adoption patterns.

A pattern that emerges from examining previous attempts to measure software engineering productivity in the literature is that proposing a framework is always a balance between measurement depth and feasibility. The two key dimensions of software engineering productivity are development speed and software quality, where measuring software quality is considered a challenging task. The methods best suited for capturing software quality, such as source code analysis or machine learning analysis, require either full access to proprietary codebases or large-scale training data. Metadata-based approaches, by contrast, sacrifice measurement depth but can be implemented without exposing source code, making them more appropriate for an organisation like Monterro with the need to apply the framework across its 30 portfolio companies. The framework proposed in this study reflects this trade-off. The five metrics selected rely on GitHub and Jira metadata, prioritising feasibility and data security over theoretical robustness. As such, the framework is not the most accurate option for evaluating productivity related to agentic coding systems, but it is one that organisations can realistically implement without compromising code confidentiality, while still providing reliable indications of overall productivity trends.

The study shows that qualitative data is necessary to interpret how teams adopt agentic coding systems over time. Cost data alone can capture the intensity of usage but reveals little about the practices, workflows, and foundational investments that define the adoption. Positioning the three teams' adoption patterns in relation to the maturity journey described by industry experts and the early academic literature reveals that even within the same organisational context, teams progress in markedly different ways. Even within a single team, individual users approach new technologies in various ways. Team A displayed an individually driven adoption pattern, team B a broad but shallow one, and team C more foundationally invested pattern where progressive users contributed to a shared infrastructure that the rest of the team could draw on. These

findings suggest that team-level factors such as size, composition, and individual perceptions shape both the pace and the character of adoption, more so than organisational direction alone.

Turning to the productivity outcomes themselves, the findings partly confirm the picture provided by the qualitative analysis. Team C, characterised by substantial investments in foundational work, shows a clear upward productivity trend, completing more tickets and merging more PRs at a faster pace. Team B, characterised by a broad but shallow adoption pattern, shows similar improvements in the volume of work completed, although its development pace has remained stable throughout the study rather than accelerating. Both teams also maintain a low proportion of large PRs and an increasing Review Intensity, suggesting that the higher development velocity has not come at the expense of code review effectiveness. It is more difficult to draw firm conclusions about team A, where the individually driven adoption pattern and the small number of active agent users mean that team-level metrics are heavily influenced by individual workload patterns. The fluctuating results may reflect this dynamic, or they may simply indicate that the team has not become more productive as a result of its coding agent usage. This uncertainty showcases the limitations of a metadata-based approach, as the observed productivity patterns cannot in isolation be attributed to agentic coding systems usage. They may equally reflect unrelated factors such as release practices or the nature of the work itself.

An overarching conclusion of this study is that the metadata-based framework proposed can be applied across Monterro's 30 portfolio companies. However, its value depends on continuous qualitative input from each portfolio company that allows Monterro's AI team to maintain an accurate picture of how agentic coding systems are used. The framework's limited ability to capture software quality means that Monterro's AI team should pay particular attention to teams whose patterns indicate potential quality risks, to ensure that productivity gains do not come at the expense of code quality.

7.2 Limitations and Future Work

This thesis has proposed and evaluated a framework for measuring software engineering productivity, motivated by the recent shift toward agent-driven development. The study comes with some limitations as well as several implications for future research.

A first limitation emerged during the research process and was related to the interviews. Respondents had limited insight into how their team members' agent usage had evolved during the past six months. This is particularly relevant for team C, in which the researcher met with only four out of twelve software engineers from the team. Therefore, it is important to be aware that the results related to the adoption patterns of each team are highly dependent on subjective meanings. A second limitation relates to the definition of productivity. The framework considers only the output side of the productivity definition provided by Petersen (2011), while inputs, typically associated with the cost of producing software, fall outside the scope of this study. Finally, the

findings should be interpreted in the context of how software engineering teams are currently organised. As coding agents mature and engineers become more effective at delegating work, the structure of teams and the nature of engineering work itself can evolve, meaning that the framework needs to be further revised. For example, some of the teams experimented with the automated flow of creating a PR straight from a Jira ticket using agents, these PRs are not included in the Output metric currently.

The study suggests two directions for future research. During the interviews, it became clear that respondents struggled to separate their use of agentic coding systems from their broader use of generative AI. The majority of respondents had been using IDE tools based on generative AI such as GitHub Copilot before the introduction of agentic coding systems, which had familiarised them with prompting code solutions. Even though these earlier tools were not always tailored to the codebase and required engineers to integrate suggestions more actively, they may still have contributed to productivity improvements. A natural extension of this study would therefore be to expand the analysis to cover the earlier period in which engineers relied primarily on generative AI, allowing the productivity effects of generative AI and agentic coding systems to be examined as part of a longer adoption journey.

A second direction concerns the engineers who have chosen not to engage heavily with agentic coding systems, even when they themselves recognise the technology's value and describe it as the future of software development. Although this was not a focus of the interviews, the pattern recurred across respondents and represents an important area for further research. Understanding why engineers resist adoption despite acknowledging the technology's potential could provide valuable insights for organisations seeking to support adoption across their development teams.

References

Agarwal, Ritu (2000). Individual acceptance of information technologies. In: Zmud, Robert W. (Eds.) Framing the domains of IT management: Projecting the future through the past. 1. Ohio: Pinnaflex Educational Resources, s. 85-104.

Agarwal, S., He, H., Kästner, C., Miller, C. and Vasilescu, B., 2026. Speed at the Cost of Quality. https://arxiv.org/pdf/2511.04427 Accessed on 2026-03-26

Ahlawat, P., Bedard, J., Damani, S., Feldman, B., Lucero, E., Samdaria, A. and Sipperly, G. (2026). The Art of Scaling GenAI in Software. Available at: https://www.bcg.com/publications/2024/the-art-of-scaling-genai-in-software [Accessed 2026-03-9].

Akour, Mohammed and Alenezi, Mamdouh (2025). AI-Driven Innovations in Software Engineering: A Review of Current Practices and Future Directions. Applied Sciences, 15(3). pp. 1-26, https://doi.org/10.48550/arXiv.2601.12560

Aljabi, Anas, Kothari, Ezan, Torrez, Timothy and Williams, Kevin (2023). Chapter 42 - Longitudinal study: design, measures, classic example. In: Bakal, Jeffrey A., DeFroda, Steven F., Eltorai, Adam E.M. and Owens, Brett D. (Eds.) Translational Sports Medicine. 1. London: Academic Press, pp. 195-199. https://doi.org/10.1016/B978-0-323-91259-4.00060-6 [Accessed 2026-03-02].

Andoh-Arthur, Johnny (2019). Gatekeepers in qualitative research. In: Atkinson, P., Delamont, Sara, Cernat, Alexandru, Sakshaug, Joseph and Williams, Richard A. (Eds.) SAGE Research Methods Foundations. London: SAGE. https://www.researchgate.net/publication/336749771_Gatekeepers_in_Qualitative_Research [Accessed 2026-03-07].

Anthropic (2024). Building effective agents. Available at: https://www.anthropic.com/engineering/building-effective-agents [Accessed 2026-02-17].

Atlassian (2026). Jira. Available at: https://www.atlassian.com/software/jira [Accessed 2026-05-21]

Bacchelli, A. and Bird, C., 2013. Expectations, Outcomes, and Challenges of Modern Code Review. https://sback.it/publications/icse2013.pdf [Accessed 2026-03-29].

Balaji, S. and Murugaiyan, Sundararajan (2012). WATERFALL Vs V-MODEL Vs AGILE: A COMPARATIVE STUDY ON SDLC. International Journal of Information Technology and Business Management, 2(1). pp. 26-30, ISSN 2304-0777.

Bell, Emma, Bryman, Alan and Harley, Bill (2022). Business research methods. Oxford: Oxford University Press.

Benbasat, Izak, Goldstein, David K. and Mead, Melissa (1987). The Case Research Strategy in Studies of Information Systems. MIS Quarterly, 11(3). pp. 369-386, https://doi.org/10.2307/248684.

Bishop, Christopher M. (2006). Pattern recognition and machine learning. 1. Eds. New York: Springer.

Blankenagel, Michael and Hunziker, Stefan (2024). Longitudinal Research Design. In: Blankenagel, Michael and Hunziker, Stefan (Eds.) Research Design in Business and Management. 2. Wiesbaden: Springer, pp. 201-2020. https://doi.org/10.1007/978-3-658-42739-9 [Accessed 2026-03-05].

Braun, Virginia and Clarke, Victoria (2006). Using thematic analysis in psychology. Qualitative research in psychology, 3(2). pp. 77-101, https://doi.org/10.1191/1478088706qp063oa.

Bryman, Alan (2012). Social Research Methods (4th ed.). Oxford: Oxford University Press.

Bryman, Alan (2016). Social Research Methods (5th ed.). Oxford: Oxford University Press.

Butler, Jenna, Forsgren, Nicole, Houck, Brian, Maddila, Chandra, Storey, Margaret-Anne and Zimmermann, Thomas (2021). The SPACE of Developer Productivity. Communications of the ACM, 64(6). pp. 46-53, DOI:10.1145/3453928.

Buyya, R., G.R, G., V, A., 2026. Agentic Artificial Intelligence (AI): Architectures, Taxonomies, and Evaluation of Large Language Model Agents. https://arxiv.org/abs/2601.12560 Accessed on 2026-03-08

Canning, Mark, Cheng, Lan, Green, Collin, Jaspan, Ciera, Kammer, Elizabeth, Knight, Andrea, Murphy-Hill, Emerson and Zhang, Nan (2022). What Improves Developer Productivity at Google? Code Quality. In: Cadar, Cristian, Roychoudhury, Abhik and Kim, Miryung (Eds.) Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 1. New York: Association for Computing Machinery, pp. 1302-1313. https://doi.org/10.1145/3540250.3558940 [Accessed: 2026-03-29].

Cargan, Leonard (2007). Doing Social Research. London: Bloomsbury Publishing PLC.

Celikyilmaz, G., Dessi, R., Dwivedi-Yu, J., Grave, E., LeCun, Y., Lomeli, M., Mialon, R., Nalmpantis, C., Pasunuru, R., Raileanu, R., Rozière, B. and Schick, T., 2023. Augmented Language Models: a Survey. https://arxiv.org/abs/2302.07842 Accessed on 2026-02-18.

Chacon, Scott and Straub, Ben (2014). Pro Git. 2. Eds. Berkeley: Apress. https://git-scm.com/book/en/v2 [Accessed 2026-03-28].

Chen, L., Huang, H., Jaisri, P., Nakashima, S., Rodríguez-Pérez, G. and Shimizu, S., 2026. More Code, Less Reuse: Investigating Code Quality and Reviewer Sentiment towards AI-generated Pull Requests. https://arxiv.org/abs/2601.21276 Accessed on 2026-02-18

Clark, Tom, Foster, Liam, Sloan, Luke and Bryman, Alan (2021). Bryman's social research methods. Oxford: Oxford University Press.

Claude API docs (2026). Agent skills. Available at: https://platform.claude.com/docs/en/agents-and-tools/agent-skills/overview [Accessed: 2026-04-28].

Cocco, Luisanna, Mannaro, Katiuscia, Concas, Giulio and Marchesi, Michele (2011). Simulating Kanban and Scrum vs. Waterfall with System Dynamics. In: Albaladejo, Xavier, Bache, Emily, Hazzan, Orit and Sillitti, Alberto. (Eds.) Agile Processes in Software Engineering and Extreme Programming. Berlin, Heidelberg: Springer, pp. 117-131. https://link.springer.com/chapter/10.1007/978-3-642-20677-1_9 [Accessed 2026-02-16]

Cohen, J., 2006. Code Review at Cisco Systems. https://static0.smartbear.co/support/media/resources/cc/book/code-review-cisco-case-study.pdf Accessed on 2026-05-02

Collins, Brian J., Hillman, Amy J. and Winthers, Michael C. (2009). Resource Dependence Theory: A Review. Journal of Management, 35(6). pp. 1404-1427, DOI:10.1177/0149206309343469

Collins, Christopher, Dennehy, Dennis, Conboy, Kieran and Mikalef, Patrick (2021). Artificial intelligence in information systems research: A systematic literature review and research agenda. International Journal of Information Management, 60(2021). pp. 1-17, doi:102383

Dartmouth College (2026). The Research Conference Where AI Began. Available at: https://home.dartmouth.edu/about/artificial-intelligence-ai-coined-dartmouth [Accessed 2026-02-12]

Deissenboeck, Florian and Wagner, Stefan (2019). Defining Productivity in Software Engineering. In: Sadowski, Caitlin and Zimmermann, Thomas (Eds.) Rethinking Productivity in Software Engineering. 1. New York: Springer, pp. 29-38. https://doi.org/10.1007/978-1-4842-4221-6 [Accessed: 2026-02-15].

Denning, P., 1992. What is Software Quality? https://denninginstitute.com/pjd/PUBS/softqual92.pdf Accessed on 2026-04-14

Deschamps, Isabelle and Leonard-Barton, Dorothy (1998). Managerial Influence in the Implementation of New Technology. Management Science, 34(10). pp. 1252-1265, https://doi.org/10.1287/mnsc.34.10.1252.

Deventura (2025). Building the Future of Developer Coaching. Available at: https://www.deventura.com/about/ [Accessed 2026-02-05]

Drosos, I. and Sarkar, A., 2025. Vibe coding: programming through conversation with artificial intelligence. https://arxiv.org/abs/2506.23253 Accessed on 2026-03-03

Dutta, K. and Sahoo, S., 2024. Boardwalk Empire: How Generative AI is Revolutionizing Economic Paradigms. https://arxiv.org/abs/2410.15212 Accessed on 2026-02-16

Elkhalili, N. and Otoum, N., 2026. Methods and Techniques of Agentic Software Engineering: A Systematic Literature Review. https://ieeexplore.ieee.org/document/11343819 Accessed on 2026-03-12

Farbey, Barbara A. (1990). Software quality metrics: considerations about requirements and requirement specifications. Information and Software Technology, 32(1). pp. 60-64, https://doi.org/10.1016/0950-5849(90)90047-U

Farley, David and Humble, Jez (2010). Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. 1. Eds. Boston: Addison-Wesley Professional.

Fenton, Norman and Neil, Martin (1999). A critique of software defect prediction models. IEEE Transactions on Software Engineering, 25(5). pp. 675-689, 10.1109/32.815326.

Forsgren, N., Greiler, M., Noda, A., Tacho, L. and Storey, A. (2025). Measuring developer productivity with the DX Core 4. https://getdx.com/research/measuring-developer-productivity-with-the-dx-core-4/ [Accessed: 2025-02-20]

Gallivan, Michael J. (2001). Organizational Adoption and Assimilation of Complex Technological Innovations: Development and Application of a New Framework. The DATA BASE for Advances in Information Systems, 32(3). pp. 51-85, https://dl.acm.org/doi/pdf/10.1145/506724.506729

GitHub (2025). GitHub's Engineering System Success Playbook. Available at: https://github.com/resources/insights/engineering-system-success-playbook [Accessed 2025-02-01]

GitHub (2026). About pull requests. Available at: https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests [Accessed 2026-02-03]

GitLab (n.d). Branching strategies. Available at: https://docs.gitlab.com/user/project/repository/branches/strategies/ [Accessed 2026-03-28]

Google Cloud (n.d). DORA: ROI of AI-assisted Software Development report. Available at: https://cloud.google.com/resources/content/dora-roi-of-ai-assisted-software-development [Accessed 2026-02-19].

Google Cloud (n.d). Instrumentation and observability. Available at: https://docs.cloud.google.com/stackdriver/docs/instrumentation/overview [Accessed 2026-05-18].

Grady, Robert B. (1993). Practical results from measuring software quality. Communications of the ACM, 36(11). pp. 62-68, https://doi.org/10.1145/163359.163369.

Greiler, M., Noda, A. and Storey, M., 2023. An Actionable Framework for Understanding and Improving Developer Experience. https://arxiv.org/abs/2205.06352 Accessed on 2026-03-17

Gutowoska, A. (2026). What is agentic engineering? https://www.ibm.com/think/topics/agentic-engineering? [Accessed: 2026-02-18]

Hassan, A., Li, H. and Zhang, H., 2025. The Rise of AI Teammates in Software Engineering (SE) 3.0: How Autonomous Coding Agents Are Reshaping Software Engineering. https://arxiv.org/abs/2507.15003 Accessed on 2026-02-10

Heinrich, Kai, Janiesch, Christian and Zschech, Patrick (2021). Machine learning and deep learning. The International Journal on Networked Business, 31(3). pp. 685-695, https://link.springer.com/article/10.1007/s12525-021-00475-2

IBM (n.d.). What is Metadata? Available at: https://www.ibm.com/think/topics/metadata [Accessed 2026-02-12].

IBM (2025). What is AI-ready data? Available at: https://www.ibm.com/think/topics/ai-ready-data [Accessed 2026-04-14].

ISO (n.d.). Machine learning (ML): All there is to know. Available at: https://www.iso.org/artificial-intelligence/machine-learning [Accessed 2026-02-13]

ISO (n.d.). What is artificial intelligence (AI)? Available at: https://www.iso.org/artificial-intelligence/what-is-ai [Accessed 2026-02-13].

Jellyfish (2026). The Intelligence Platform for AI-Integrated Engineering. Available at: https://jellyfish.co/ [Accessed 2026-02-05].

Koza, John R., Bennett, Forrest H., Andre, David and Keane, Martin A. (1996). Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming. In: Gero, John S. and Sudweeks, Fay (Eds.) Artificial Intelligence in Design '96. Dordrecht: Springer, pp. 151-170. https://doi.org/10.1007/978-94-009-0279-4_9 [Accessed: 2026-02-26].

Lawrence, Michael J. (1981). Programming methodology, organizational environment, and programming productivity. The Journal of Systems and Software, 2(3). pp. 257-269, https://doi.org/10.1016/0164-1212(81)90023-6

Monterro (2026). Built for the AI Era. Available at: https://www.monterro.com/why-us/infuse-ai/ [Accessed 2026-04-17].

Morisio, Maurizio, Muhammad, Syed and Torchiano, Marco (2013). An Overview of Software Defect Density: A Scoping Study. 2012 19th Asia-Pacific Software Engineering Conference. pp. 406-415, 10.1109/APSEC.2012.93

Noda, A. and Tacho, L. (2025). Measuring AI code assistants and agents. https://getdx.com/research/measuring-ai-code-assistants-and-agents/ Accessed on 2026-01-18.

Orlikowski, Wanda J. (1993). CASE Tools as Organizational Change: Investigating Incremental and Radical Changes in Systems Development. MIS Quarterly, 17(3). pp. 309-340, https://doi.org/10.2307/249774

Petersen, Kai (2011). Measuring and predicting software productivity: A systematic map and review. Information and Software Technology, 53(4). pp. 317-343, https://doi.org/10.1016/j.infsof.2010.12.001

Rachitsky, L., 2026. This week on How I AI: How Stripe built "minions" - AI coding agents that ship 1,300 PRs per week + How to turn Claude Code into your personal life operating system. https://www.lennysnewsletter.com/p/this-week-on-how-i-ai-how-stripe Accessed on 2026-04-07.

Rogers, Everett M. (2003). DIFFUSION OF INNOVATIONS. 3. Eds. London: Collier Macmillan Publishers.

Routley, N. (2023). What is generative AI? An AI explains. https://www.weforum.org/stories/2023/02/generative-ai-explain-algorithms-work/ [Accessed: 2026-02-10].

Rubin, Kenneth (2012). Essential Scrum. 1. Eds. Ann Arbor: Edwards Brothers Malloy. https://libdoc.dpu.ac.th/eBook/113642.pdf [Accessed: 2026-03-28].

Ruef, M., 2014. Source Code Analysis - A Beginner's Guide. https://www.researchgate.net/publication/336838677_Source_Code_Analysis_-_A_Beginner's_Guide Accessed on 2026-02-12.

Ruhe, M. and Wagner, S., 2018. A Systematic Review of Productivity Factors in Software Development. https://arxiv.org/pdf/1801.06475 Accessed on 2026-02-15

Runeson, Per and Höst, Per (2009). Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering, 14(Dec). Pp. 131-164, https://doi.org/10.1007/s10664-008-9102-8

Sarker, Iqbal H. (2021). Machine Learning: Algorithms, Real-World Applications and Research Directions. SN Computer Science, 2(160). pp. 1-21, https://doi.org/10.1007/s42979-021-00592-x.

Solidafy (2026). Build analytics for anything. Available at: https://www.solidafy.com/ [Accessed 2026-02-05].

Swarmia (2026). Swarmia shapes the way great software is made. Available at: https://www.swarmia.com/about-us/ [Accessed 2026-02-05].

SWEPR (2025). AIE CODE 2025: AI Leadership ft Anthropic, OpenAI, McKinsey, Bloomberg, Google Deepmind, and Tenex. YouTube. https://www.youtube.com/watch?v=cMSprbJ95jg&t=9707s [Accessed 2026-01-18].

Wilson, Virginia (2013). Research Methods: Mixed Methods Research. Evidence Based Library and Information Practice, 8(2). pp. 275-277, https://doi.org/10.18438/B8801M.

Yin, Robert K. (2016). Qualitative Research from Start to Finish. New York: The Guilford Press.

Appendix

Interview Guide: Software Engineers

- Can you tell me about how the introduction of coding agents looked in the team?
- When did the team get access to coding agents?
- Was any specific expectation communicated regarding the team’s use of the coding agents?
- If so, how did you perceive that expectation?
- What did the first use cases look like?
- Were you given dedicated time to experiment with the coding agents?
- How would you describe the team's willingness to use these coding agents when they were introduced?
- Thinking about a typical workday today, to what extent do you use coding agents?
- What does the distribution look like within the team? Does everyone use them to roughly the same extent, or is there significant variation? If so, what do you think the variation is due to?
- How would you describe your and the team’s usage patterns of coding agents?
- If you compare a typical work week today with how things looked six months ago, how have the team’s usage patterns of coding agents changed?
- Can you identify any clear phases or shifts in how the team has used the agents during this period?
- What has driven those changes? Has it been about specific events, decisions, new tools, or has it happened more organically?
- Has the team worked on building up the necessary context for the agents to perform as well as possible? (documentation, skills, README files, for example)
- Have you experienced any change in the team’s workflow over the past six months as a result of using coding agents? If so, can you describe what a workflow looks like today compared to before, from receiving a task to the code being merged?

- How has your level of trust in the output from coding agents changed over time? Have you experienced any incidents that affected that trust positively or negatively?
- Do you feel that your role as a developer has changed since you started using coding agents? If so, how?
- If you could give one piece of advice to a team that is just about to start using coding agents, what would it be?
- Is there anything important about the team's use of coding agents that I haven't asked about?

Interview Guide: Agentic AI Experts

- Could you start by telling us a bit about the development team (number of developers, division of work, and methodology)?
- What is your role in relation to the development team?
- When did you start experimenting with coding agents (Copilot, Claude, etc.) at Company X?
- What prompted you to start experimenting with coding agents (e.g., a specific problem, curiosity, or a management initiative)?
- What did the first use cases for coding agents look like in practical terms?
- What worked well in the early use of coding agents, and what worked less well?
- Were there people on the team who were particularly driving the adoption of coding agents?
- What characterized these people (e.g., seniority, mindset, role)?
- How crucial were they to the fact that you actually started experimenting with coding agents?
- How did the use of coding agents spread within the team, and have you deliberately taken initiatives to increase this spread?
- Was there any resistance within the team? From whom and why? (feel free to describe without singling anyone out personally)
- Which agent solutions do you use today?
- What was your reasoning when choosing these particular solutions?

- Have you switched agent solutions along the way? If yes, why?
- How do you stay up to date on new possibilities and features within different agent solutions?
- What does your development process look like today?
- In what ways does it differ from how you worked before you started using coding agents?
- How have agents affected the way you work with coding, testing, code review, debugging, and documentation?
- Have you changed your way of working methodologically (you mentioned, for example, spec-driven development)?
- In which parts of the development process do you think agents provide the most value?
- How has the developer role at Company X changed since you started using agents?
- Have the requirements for competence and responsibility among developers changed? If yes, in what way?
- How has the use of agents affected junior developers compared to senior ones?
- Has it affected the structure or size of the team? If yes, how?
- Do you notice any difference in delivery speed?
- Do you feel that the quality of the code has changed?
- Do you have any concrete ways to measure delivery speed and quality, or is it mainly based on perception?
- How do you view cost versus value when it comes to the use of agents?
- What obstacles have you encountered on the journey toward becoming more agent-driven in your development?
- How have you worked around these obstacles?
- What would you have done differently if you started over today from a situation without coding agents?
- If you were to help another team make a similar journey to the one Company X has made, what are the most important prerequisites for success?

- What are your ambitions going forward when it comes to continuing your journey toward becoming more agent-driven in your development?
- What do you think will be required of you at Company X to achieve this vision?