# Artificial Intelligence in the Software Development Life Cycle: A Case Study in a Mid-Size Software Organization

Juuso Liljavirta

Haaga-Helia University of Applied Sciences
Degree Programme in Business Technologies
ICT Services and Systems
Master Thesis
2025

## Abstract

**Author(s)**
Juuso Liljavirta

**Degree**
Degree Programme in Business Technologies

**Report/thesis title**
Artificial Intelligence in the Software Development Life Cycle:
A Case Study in a Mid-Size Software Organization

**Number of pages and appendix pages**
63 + 2

This thesis examines the utilization of artificial intelligence (AI) within different phases of the software development life cycle (SDLC) in a mid-size IT consulting firm. The objective is to understand the current use of AI, identify future opportunities and highlight obstacles that prevent effective integration into SDLC processes, focusing specifically on the phases of requirements, development, testing and maintenance.

The research was based on existing academic literature that discussed AI applications, including machine learning, natural language processing and AI agents within software development. Qualitative data was collected through semi-structured interviews with nine employees and two external AI specialists, offering practical examples and experiences regarding AI's role and potential.

Findings showed that AI was primarily utilized during the development phase, particularly through coding assistants such as GitHub Copilot and conversational tools like ChatGPT. Developers reported benefits in productivity, reduced routine workload, and improved capability for identifying coding issues and optimization opportunities. In contrast, its application remained limited in the requirements engineering, testing, and maintenance phases, mainly due to the complexity of specifying business logic, insufficient data for robust models, and concerns over accuracy and traceability.

The study identified several future opportunities for AI across the SDLC. These included automating documentation, generating detailed test cases, and improving communication among project stakeholders. Interviewees also suggested advanced applications such as automated refactoring, predictive maintenance, and intelligent traceability mechanisms that could streamline processes and reduce human error. Agent-based solutions and retrieval-augmented tools were also discussed as promising developments for the future

The study also revealed multiple obstacles to broader adoption. These included data security and privacy concerns, lack of training and knowledge, technical limitations, and uncertainties about trust and responsibility. Organizational and process-related challenges were also noted.

This thesis concludes with general recommendations for software development organizations aiming to utilize AI more effectively. These include a focus on gradual experimentation, better knowledge management, and preparing for agent-based collaboration in the future.

**Keywords**
Artificial intelligence, generative AI, AI agents, software development, software development life cycle

## Table of contents

1 Introduction ... 1
1.1 Research objective and questions ... 1
1.2 Thesis structure ... 2
2 Theoretical framework ... 4
2.1 Software development life cycle ... 4
2.1.1 Requirements phase ... 6
2.1.2 Development phase ... 7
2.1.3 Testing phase ... 8
2.1.4 Maintenance phase ... 9
2.2 Artificial intelligence ... 10
2.2.1 Machine and deep learning ... 11
2.2.2 Language-based AI and retrieval-augmented generation ... 12
2.2.3 AI agents ... 12
2.3 Artificial Intelligence in SDLC ... 13
2.3.1 AI in Requirements phase ... 14
2.3.2 AI in Development phase ... 15
2.3.3 AI in Testing phase ... 17
2.3.4 AI in Maintenance phase ... 18
3 Case study of utilizing AI in software development life cycle ... 21
3.1 Organization X ... 21
3.2 The approach and methodology ... 21
3.3 The interview process ... 24
4 Findings ... 28
4.1 Current use of AI (RQ1) ... 28
4.1.1 Requirements phase ... 28
4.1.2 Development phase ... 30
4.1.3 Testing phase ... 31
4.1.4 Maintenance phase ... 32
4.2 Possibilities of AI use in the future of SDLC (RQ2) ... 34
4.2.1 Requirements phase ... 35
4.2.2 Development phase ... 37
4.2.3 Testing phase ... 39
4.2.4 Maintenance phase ... 41
4.2.5 Agent-based AI support across SDLC phases ... 43
4.3 Obstacles hindering use in SDLC (RQ3) ... 45

4.3.1 Data security and privacy concerns ... 45
4.3.2 Lack of knowledge and skills ... 46
4.3.3 Trust and reliability issues ... 47
4.3.4 Integration and technical limitations ... 47
4.3.5 Organizational challenges ... 48
5 Conclusion ... 50
5.1 Key findings ... 50
5.2 Recommendations based on findings ... 52
5.3 Future direction: AI agents and autonomous support ... 55
5.4 Ethics and reliability ... 56
5.5 Future research ... 57
5.6 Self-reflection ... 57
References ... 59
Appendices ... 64
Appendix 1. General question structure of semi-structured interviews ... 64
Appendix 2. Consent form (translated to English) ... 65

# 1 Introduction

Artificial intelligence (AI) is rapidly becoming a central topic in the software industry. As digital services grow in scale and complexity, organizations are looking for ways to improve productivity, reduce repetitive tasks, and maintain quality under increasing pressure. For IT companies, especially those working with clients across industries, the ability to deliver software efficiently while managing constant change is a key competitive factor. AI tools are often seen as part of the answer, offering support for tasks such as coding, testing, documentation, and requirements work.

Although AI has been successfully applied in fields like data analytics and image recognition, its everyday use in software development work is still emerging. Tools like ChatGPT and GitHub Copilot have made AI more accessible, but adoption varies widely across organizations. While some teams have integrated these tools into their daily routines, others are still unsure how to apply them effectively or safely in real projects.

The rapid development of generative AI and the increasing availability of commercial tools have created both opportunities and pressure for companies to act. At the same time, the lack of clear practices or shared understanding of how AI should be used in software development has left many organizations in a transitional state. These tensions make the topic both practically relevant and research-worthy.

This thesis investigates how AI is used across different phases of the software development life cycle, what opportunities it may offer and what obstacles currently limit its adoption. The aim is to move beyond general expectations and examine what kinds of benefits AI can concretely provide in everyday software work.

## 1.1 Research objective and questions

The research is based on semi-structured interviews conducted at Organization X, a mid-sized IT consulting firm with a global presence. The interviewees represented a range of roles within the software development life cycle and described both their current work and how AI has or could become part of it. These discussions provided insight into practical tasks and challenges, helping to identify areas where AI tools might support development work more effectively. In addition, two external AI specialists were interviewed to provide broader perspectives beyond the organization's internal practices. A literature review was also carried out to support the interpretation of findings and situate the study within existing research.

The focus is on practical uses of AI tools, potential ways AI could change development work and what kinds of obstacles organizations face when trying to adopt AI-based solutions. By combining

findings from earlier studies with interview data, the aim is to identify ways to increase productivity, improve software quality, and reduce inefficiencies across different phases of the SDLC. This is addressed through the following research questions:

**RQ1: How is AI used in different phases of the SDLC?**

This question explores current uses of AI-driven tools and methods in areas such as requirements gathering, development, testing, and maintenance. The goal is to identify practices that improve efficiency and strengthen outcomes.

**RQ2: How could AI be used in the future during these phases?**

Here, the focus shifts to potential future applications of AI in the SDLC. Drawing on both literature and interviews, this part considers new techniques that could improve adaptability and streamline development activities.

**RQ3: What obstacles hinder the effective use of AI in the SDLC?**

The third question addresses barriers to adoption, including technical, organizational and human-related challenges. Understanding these factors is important when considering how AI can be introduced in a sustainable and secure way.

By answering these questions, the thesis aims to build a clear picture of where AI fits into software development today and where it could go next. The intention is that the findings will offer useful insights for Organization X and other companies that are considering how to bring AI into their software development processes.

### 1.2 Thesis structure

This thesis examines how artificial intelligence is, or could be, used in the software development life cycle at Organization X. It begins with background and theoretical sections, followed by case study findings and final conclusions. The structure combines theoretical perspectives with practical insights gained through interviews. Following the introduction and research questions, the next part presents the theoretical framework. It outlines the SDLC, focusing on four key phases: requirements, development, testing, and maintenance. These were selected because they are the most relevant for understanding the role of AI in everyday software work and were reflected in the responsibilities of the interviewees. Although the design phase is generally considered a core SDLC component, it was not emphasized in this case and is therefore excluded.

In the same section, the concept of artificial intelligence is introduced. Key areas such as machine learning, deep learning, natural language processing, generative AI and AI agents are discussed. Their basic principles are explained, along with examples of how these technologies can support different tasks in software development. While the emphasis is on AI as a supportive tool rather than a replacement for human work, the practical applications are further elaborated in the following sections that focus on the SDLC context.

After the theory, the thesis moves to the case study of Organization X. This part provides a real-life setting for the research and introduces the organization's background. The research method is qualitative, using semi-structured interviews to collect data. This approach was chosen because the topic required deeper understanding, and gathering a reliable sample for a survey would have been difficult. The goal was to explore individual experiences and perspectives rather than to produce generalizable results.

The interview structure followed Anne Galletta's three-part model, including an opening to build rapport, a middle section for detailed discussion, and a closing segment to summarize the conversation. This structure allowed consistency across interviews while also supporting flexibility when needed.

The interview process is described in detail, including participant selection, scheduling, and the use of Microsoft Teams for remote meetings. Most participants had experience across multiple SDLC phases, which helped provide a broader view. Although the original plan was to find one person per phase, this turned out to be impractical. In addition to internal employees, two external AI specialists were also interviewed to gain a wider perspective. In the end, the group reflected the more flexible and overlapping roles typical of agile software teams.

The findings are organized around the three research questions. Each SDLC phase is examined regarding current AI usage and possible future applications. Challenges are grouped thematically, reflecting common patterns across different phases. Interviewee quotes are included to illustrate key points and provide a window into everyday software work. The analysis also draws attention to gaps between theory and practice, especially in areas where AI adoption is still developing.

The thesis concludes with main findings and practical advice for organizations engaged in software or internal development. These include improving documentation practices and beginning to explore more advanced uses of AI, such as AI agents and retrieval-augmented systems. The conclusion also reflects on how organizations might gradually build the capabilities needed for more autonomous forms of software development. In addition, the final chapter suggests future research directions and includes the author's self-reflection on the research process.

# 2 Theoretical framework

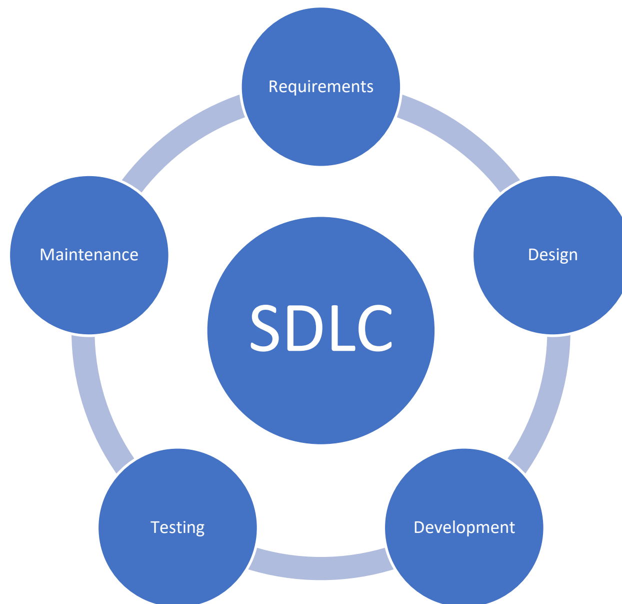
## 2.1 Software development life cycle

The software development life cycle is a conceptual framework that structures the phases of software development. While the model outlines the key phases, a methodology specifies the detailed actions to be carried out during each phase. Commonly used SDLC methodologies include the traditional waterfall model and more contemporary agile models. The waterfall model follows a linear, sequential approach, where each phase (e.g., requirements, design) must be completed before the next phase begins. In contrast, agile methodologies adopt an iterative approach, breaking the project into smaller, manageable parts and enabling continuous feedback and adaptation throughout the development process. (Ruparelia 2010, 8-9, 12-13.)

Within agile, frameworks such as Scrum and SAFe (Scaled Agile Framework) are commonly employed to implement iterative methodologies. These frameworks are tailored to different project sizes and complexities. For example, Scrum is often used in smaller projects due to its lightweight structure, while SAFe is designed for large, multi-actor projects requiring greater coordination. SAFe frameworks are further categorized into Essential, Large Solution, Portfolio, and Full SAFe, depending on the depth of implementation. For smaller projects, less comprehensive versions of SAFe are used to reduce administrative overhead. (Saklamaeva & Pavlič 2024, 1–3.) In organization X, agile methodologies are the primary choice for software development. Scrum is typically employed in smaller projects, while SAFe is preferred for larger, multi-vendor projects requiring synchronized timelines and sub-project management. Research indicates that over half of agile implementations worldwide now utilize SAFe, reflecting its growing popularity in large-scale software development (Saklamaeva & Pavlič 2024, 1).

Acharya and Sahu (2020, 169–171) outline the software development life cycle as comprising five key phases. However, variations exist in literature, with some sources breaking phases into smaller steps or adding additional phases. This thesis focuses on four core phases: requirements, development, testing, and maintenance. The design phase is not addressed separately, as design-related activities in iterative practices such as those used in Organization X are typically carried out as part of the requirements and development phases. This selection also reflects the scope and structure of the interview framework used in this study.

Figure 1: SDLC phases (Acharya & Sahu 2020, 170)




This thesis follows the SDLC description but focuses primarily on the activities within each phase rather than on the phases themselves. In practice, the same team members often work across multiple phases, making it difficult to clearly separate them. This kind of overlap is especially common in agile teams, where development cycles are shorter and iterative. For example, although testing is considered a distinct phase in the SDLC, it also occurs in several other phases. Developers test their own modifications, and maintenance tasks often require testing compatibility after updates.

Another overlapping activity is deployment. While the first release to production might be viewed as the transition to the maintenance phase, deployments to testing environments can extend over months and are equally important for smooth operations. Therefore, although the following chapters describe these activities as part of specific SDLC phases, the interview participants in the research section may discuss and explain their experiences primarily through the activities they perform.

The following chapters describe each SDLC phase in general terms. Although artificial intelligence is discussed in detail later in this thesis, short references are included at the end of each SDLC phase to hint at possible areas of application. These connections help contextualize the later analysis.

### 2.1.1 Requirements phase

The requirements phase is the first and one of the most important phases of the software development life cycle, as it defines the client's needs and the software's objectives, functionality, and constraints. It provides the foundation for the rest of the process by ensuring that development is aligned with stakeholder expectations. (Acharya & Sahu 2020, 171; Batarseh et al. 2020, 4–5.) Accurate requirement definition is critical, as errors or ambiguities at this stage often lead to costly rework in later phases.

Identifying and involving all relevant stakeholders is essential in the requirements phase. These stakeholders may include customers, end users, authorities, and software developers themselves. Effective communication among these groups ensures mutual understanding of what the software will and will not do. Additionally, this approach makes sure that everyone is committed to common objectives, minimizing misunderstandings among stakeholders as the project progresses. (Acharya & Sahu 2020, 171; Pargaonkar 2023, 123.)

Various methods and tools are used to define requirements, including the Unified Modeling Language (UML). UML provides a set of visual tools for analyzing, designing, and implementing software-based systems. Additionally, requirements can be expressed using natural language, diagrams, or business process models. Selecting methods and tools appropriate to the project's scale, complexity and stakeholders is important for effective documentation and communication. (Bourque & Fairley 2014, 39–42.) For example, smaller projects may rely on straightforward user stories, while larger projects often benefit from structured modelling techniques.

Iterative agile methodologies make managing requirements more challenging. In agile projects, not all requirements are fully understood at the start and changes are frequent. Effective change management processes should be used to ensure that these adjustments are systematically incorporated while maintaining project control. (Bourque & Fairley 2014, 43–44.) Agile practices in Organization X emphasize flexibility but also reveal opportunities for improvement through automation and AI-based solutions.

In Organization X, requirements are typically documented in natural language as user stories. These are collaboratively created with clients, requirements engineers, and developers. However, this process has its challenges. Frequent virtual meetings and email exchanges often lead to expanding requirements, which can complicate traceability. When issues such as a defective feature arise, it is often cumbersome to locate the original requirement or track changes made over time.

This shows the need for effective tools and methods to ensure transparency and traceability throughout the process.

AI can offer practical ways to manage challenges in the requirements phase. For instance, natural language processing tools can analyze and flag ambiguities or inconsistencies in requirement documents. AI-driven traceability tools link user stories to their corresponding features, making it easier to track changes and identify their impacts. Implementing such tools in Organization X could simplify requirements management, improve communication among stakeholders, and reduce the time spent resolving ambiguities.

### 2.1.2 Development phase

The development phase of software engineering is where the actual construction of the software occurs. This phase primarily involves coding, whether implementing new features or modifying existing ones. Various technologies and tools are utilized to ensure software quality and functionality. These include programming languages, integrated development environments (IDEs), version control systems and testing tools. Ensuring code quality and readability is critical in this phase, as it supports future maintenance and assists collaboration among developers. (Sharma 2017, 521–522.)

Integrating software components with other systems is a common requirement during the development phase. This makes sure that the software interacts seamlessly with the larger system and meets compatibility requirements. Reusing code and other assets helps avoid duplication of effort and reduces the time spent on repetitive testing. By reusing pre-tested components, teams save time and improve reliability. (Bourque & Fairley 2014, 72.)

The development phase in agile is iterative, allowing teams to adjust and refine their work when the project requirements change due various reasons. In some contexts, the term "development phase" encompasses broader activities in the software lifecycle, such as design, implementation, and testing. However, this thesis focuses on the implementation phase, where the software is written and assembled. In Organization X, architects are responsible for designing the initial infrastructure and tools. Developers also have an important part in later stages of the design process, reflecting a collaborative and iterative approach used by the organization.

For Organization X, AI can help with the collaboration between architects and developers. This integration could improve the overall quality and adaptability while supporting the organization’s iterative development approach. AI can also automate repetitive coding tasks and provide suggestions on how to cope when design requirements evolve during lifecycle.

### 2.1.3 Testing phase

The testing phase focuses on ensuring the quality and functionality of the software. During this phase, various tests are conducted to identify and fix errors or deficiencies. The main goal of testing is to verify that the software behaves as intended and meets its defined requirements. Testability is an important consideration during this phase. This refers to how easily tests can achieve full coverage and detect errors effectively. In Organization X, developers perform initial testing during the development phase, but dedicated testers often carry out more extensive evaluations such as regression testing and testing for new implementations. Test automation is commonly applied for regression testing, although it is not used in all projects. In some cases, clients conduct regression testing or user acceptance testing (UAT) before new features are integrated into production systems.

The testing phase uses a variety of methods and techniques depending on the software's objectives and characteristics. Functional testing, for example, checks if the software performs its intended functions accurately. Other areas of testing include performance testing, which focuses on speed and scalability, and reliability testing, which examines system stability. Usability testing, on the other hand, evaluates the quality of the user experience. Selecting the right methods and tools for testing is required for achieving thorough and effective results. (Bourque & Fairley 2014, 83–88.)

Testing is conducted at several levels, each addressing different aspects of software quality. Unit testing evaluates individual components for correctness. Integration testing examines how modules or systems interact with one another, while system testing validates the functionality of the entire application. Additionally, acceptance testing ensures that the software meets the expectations of end-users or clients. The choice of testing levels depends on the complexity and scale of the software as well as the stage of the development process. (Bourque & Fairley 2014, 43, 69–71.)

Success in the testing phase depends on thorough planning and sufficient resource allocation. Clear objectives need to be defined, suitable methods and tools selected, and testers equipped with the necessary skills. Testing efficiency is often assessed by using metrics like defect density, test coverage and reliability. These metrics provide valuable insights into testing progress and help identify areas for improvement. (Swanlund, Wilson, Kapoor, Gabbard & Srinath 2024, 8.)

In Organization X, artificial intelligence could help reduce time and effort in the testing phase by automating repetitive tasks and analyzing test data. It may also support resource optimization by focusing on high-risk areas, potentially shortening the testing cycle without compromising coverage or quality.

### 2.1.4 Maintenance phase

The maintenance phase is the longest phase in the software lifecycle, beginning after the software is delivered and continuing until it is retired. This phase focuses on keeping the software operational, dependable, and capable of adapting to changing requirements and environments. However, maintenance has traditionally been given less emphasis than the development stages. (Bourque & Fairley 2014, 104.)

Maintenance encompasses four key activities. Corrective maintenance addresses identified defects, while adaptive maintenance modifies the software to meet new environmental requirements, such as updates to operating systems or hardware. Perfective maintenance involves enhancing existing features or adding new ones to meet user needs, and preventive maintenance aims to mitigate latent issues before they become operational problems. (Bourque & Fairley 2014, 106.) Together these activities help the software evolve and continue meeting user needs.

The costs of maintenance account for a significant share of total software lifecycle expenses. Bourque and Fairley (2014, 106) state that over 80 % of maintenance costs are spent on tasks unrelated to defect correction, such as performance tuning, system optimization, and feature enhancements. These expenses are influenced by several factors, including the software's operating environment, organizational practices, and its inherent maintainability. Maintainability, defined as the software's ability to be modified and extended efficiently, is a critical characteristic that should be prioritized during development to reduce long-term costs and complexity. (Bourque & Fairley 2014, 106–107.)

Effective maintenance relies on a proactive approach that includes planning during the development phase and allocating appropriate resources throughout the software lifecycle. Anticipating future maintenance needs, ensuring high software quality, and building a supportive organizational culture are all helpful for successful maintenance efforts. Retaining skilled personnel, maintaining motivation, and providing clear processes for change management further improve maintenance efficiency and consistency. (Pargaonkar 2023, 124; Sharma 2017, 521–522.)

One of the primary tasks during maintenance is to analyze the impacts of changes made over the software's lifecycle. Understanding how modifications affect functionality, and maintainability helps organizations make better decisions about updates and future adaptations. This analysis also contributes to the sustainability of the software in ever-changing conditions. (Angerer, Grimmer, Prähofer & Grünbacher 2019, 418.)

Artificial intelligence may support maintenance by enabling more predictive and automated processes. Predictive maintenance uses data analysis to anticipate issues, which can help reduce

downtime and costs. In Organization X, AI could assist by automating routine tasks, improving efficiency, and helping the software adapt to evolving user needs.

### 2.2 Artificial intelligence

The term "artificial intelligence" was first introduced by John McCarthy at the Dartmouth Conference in 1956 (Grzybowski, Pawlikowska–Łagód & Lambert 2024, 225). Initially, AI was conceived as the simulation of human intelligence, enabling machines to perform tasks such as learning, reasoning, and problem-solving. McCarthy and his colleagues proposed that intelligent machines might eventually improve themselves, marking self-improvement as a key research direction in early AI (McCarthy 2006, 12). Stuart Russell and Peter Norvig (2016) define AI in terms of four categories: thinking humanly, acting humanly, thinking rationally and acting rationally. Acting humanly refers to the ability of AI to replicate human-like behavior, such as in the Turing Test, where the objective is to determine whether a human can distinguish between interacting with another human and interacting with an AI. If the AI successfully convinces the person that they are engaging with a human, it is said to have passed the test. This approach is particularly relevant to natural language processing, where AI systems engage in human-like interactions, maintaining contextual understanding over longer conversations. Thinking humanly, on the other hand, involves AI's reasoning processes aligning with human cognitive patterns. This category is closely tied to psychology and cognitive science, fields that continue to explore the complexities of human cognition. In the context of AI, thinking humanly emphasizes transparency, allowing users to trace the steps the AI took to reach a specific conclusion. (Russell & Norvig 2016, 2–4.) Transparency is also useful in applications requiring traceable decision-making, although this thesis focuses on software development tasks rather than socially critical domains. By understanding how AI reaches its decisions, stakeholders can improve algorithms and adjust their behavior to achieve more favorable outcomes.

Rational thinking differs slightly from human-like reasoning by emphasizing strict adherence to logical principles. This dimension represents reasoning akin to that of a highly logical and systematic individual. While humans are capable of irrational thinking, AI systems designed for rationality focus exclusively on logical and structured thought processes. Finally, acting rationally extends beyond reasoning to emphasize autonomy. AI systems in this category can perceive their environment, interacting with it, and making independent decisions to achieve specified goals. This dimension positions AI as an autonomous agent, capable of functioning independently and adapting to its environment. Acting rationally is viewed as the broadest and most central approach to AI, as it encompasses the widest range of applications. (Russell & Norvig 2016, 4–5, 34.)

In this thesis, artificial intelligence refers primarily to generative AI solutions that can create or synthesize new content, such as code, documentation or test cases, by simulating aspects of human

reasoning, learning and decision-making. These systems are characterized by autonomy, self-improvement and the ability to adapt dynamically to changing environments.

The emphasis is on practical applications of large language models and related generative technologies that support knowledge work within the software development life cycle. This includes assisting with requirements clarification, code generation, testing and maintenance tasks. While the broader definition of AI can include rule-based systems, basic automation, and traditional machine learning models, such approaches are excluded from the scope of this thesis. The focus is specifically on AI tools that contribute to efficiency, accuracy and knowledge sharing through content generation or context-aware interaction.

### 2.2.1 Machine and deep learning

Machine learning (ML), a subfield of artificial intelligence, is sometimes referred to as pattern recognition in scientific literature (Kämäräinen 2023, 10–11). Kelleher (2020, 15–17) describes ML as the development of algorithms that learn from data. They process input data, detect patterns, and generate corresponding outputs. Unlike traditional algorithms, ML does not rely on predefined rules; instead, the algorithm learns by being trained on large datasets or samples. ML training can be categorized into supervised, unsupervised and reinforcement learning.

In supervised learning, the dataset is labelled, enabling the algorithm to learn from predefined examples. For instance, the algorithm could be trained with images of cats and dogs, learning to classify these correctly and even identifying images that do not match either category. Unsupervised learning, in contrast, involves unlabeled data, requiring the algorithm to independently identify patterns and categorize data, such as determining which images represent cats or dogs. Semi-supervised learning combines elements of both methods, using a partially labelled dataset for training. Reinforcement learning employs a reward-based system, where the algorithm learns through interaction with its environment. Desired outcomes are rewarded, while undesired outcomes are penalized, allowing the system to improve without requiring a pre-existing dataset. (Kelleher 2020; Kühl, Schemmer, Goutier & Satzger 2022, 2237–2238.)

Deep learning (DL), a subset of ML, focuses on neural networks inspired by the human brain. Just as the brain comprises billions of interconnected neurons, neural networks consist of layers that process and transmit information. The first layer, the input layer, receives data, while the final layer, the output layer, provides results. Between these, multiple hidden layers perform processing and assign weights to information received from the previous layer. DL is particularly effective in unsupervised learning scenarios, where its ability to process vast amounts of unstructured data is advantageous. (Kelleher 2020, 64–69.)

### 2.2.2 Language-based AI and retrieval-augmented generation

Deep learning has significantly advanced natural language processing (NLP), improving the performance of tasks such as translation, classification, and language modeling (Kelleher 2020, 165–166). NLP supports rapid translation and enables intuitive interfaces allowing users to interact with AI systems through natural conversation-like interfaces. While text-based applications are most common, speech processing can also be part of NLP systems.

NLP applications are often divided into two categories: natural language understanding (NLU) and natural language generation (NLG). NLU involves interpreting and extracting meaning from text, enabling systems to understand user queries or classify documents. NLG, on the other hand, focuses on creating coherent and contextually appropriate text, often based on structured input such as databases or predefined templates. (Dong et al. 2023, 2.)

Generative AI goes beyond language and can create new content, ranging from text and images to audio and programming code. These systems learn and replicate patterns from training data, generating outputs that approximate human creativity. Instead of merely predicting outcomes or categorizing data, generative AI produces novel content. Known examples are ChatGPT for text generation and DALL-E 2 for image creation. (Feuerriegel, Hartmann, Janiesch & Zschech 2024, 111–112.)

Retrieval-augmented generation (RAG) is emerging as an important extension of generative AI. By pairing a large language model with an external retrieval component, RAG systems can pull up-to-date, context-specific information from internal or external knowledge bases during generation. This design improves factual accuracy, offers explanatory background, and lowers the risk of hallucinations that commonly affect standalone language models (Zhang et al. 2024, 416–417.) In a software-development setting, RAG could automatically surface relevant code snippets, requirements documents, or best-practice guidelines and weave them into AI outputs, thereby grounding responses in real project data. Access to such contextual information supports tasks such as keeping documentation current, clarifying requirements, and onboarding new developers, all of which depend on precise domain knowledge. As organizations broaden their use of generative AI, retrieval augmentation may become an important approach for delivering reliable, knowledge-intensive assistance (Zhang et al. 2024, 416–417, 421).

### 2.2.3 AI agents

AI agents are autonomous or semi-autonomous systems capable of perceiving inputs, reasoning about tasks, and taking actions toward predefined goals. Unlike task-specific AI tools, agents are typically designed to operate over sequences of actions, often adapting their behavior based on

environmental feedback or collaboration with other agents or users. (Suri et al. 2023, 1855–1856.) This structure enables agents to support more complex, multi-step processes compared to conventional automation scripts or standalone models.

Many recent AI agents have been built on large language models, which allow them to understand natural language, process documents, and carry out interactions with humans or software systems. Depending on the framework, agents may be equipped with memory, tool use capabilities, or access to vector databases for retrieval-augmented tasks. These capabilities can be combined in orchestration frameworks, where multiple agents adopt specialized roles and collaborate toward shared outcomes. (He Junda, Treude Christoph & Lo David 2024, 1–2, 16.) For example, agents may take on roles such as planner, coder, tester, or analyst, with their tasks managed either by a coordinating system or through interactions among the agents (Sami et al. 2024, 1).

### 2.3 Artificial Intelligence in SDLC

The integration of artificial intelligence into the software development life cycle has been researched a lot in recent years for its potential to improve productivity, enhance quality, and rationalize workflows across various phases of software development. While the concept of applying AI to SDLC has been actively researched in recent decades, its exploration dates to as early as the 1960s. Since then, research has evolved to explore a wide range of AI applications in software engineering. For example, Zhang and Tsai (2003, 96–98) emphasize the potential of AI tools and techniques, particularly machine learning, to enhance quality and productivity in SDLC. They highlight its applications in predicting software development costs, identifying potential defects, and facilitating the reuse of components in new projects. Expanding on this, Shafiq et al. (2021, 140901–140902) argue that AI brings predictive and optimization capabilities that are especially helpful in the later phases of SDLC, such as testing and maintenance. According to their findings, AI is not typically viewed as a replacement for human input but rather as a tool to assist and augment human decision-making. Similarly, Jarrahi (2018, 578) points out the collaborative potential of AI, describing it as an augmentation technology that enhances human creativity and productivity by taking over routine and repetitive tasks. This distinction between automation and augmentation is important in understanding how AI supports human expertise and works across the SDLC.

Despite its advantages, integrating AI into SDLC does have its challenges as well. Giffari et al. (2024, 81–82) note that implementing AI tools often requires extensive organizational changes, including training personnel and acquiring knowledge about specific tools. Similarly, Shafiq et al. (2021, 140902) argue that AI's adoption in SDLC remains fragmented, often limited to specific tasks such as automated testing or requirements analysis. Additionally, the rapid pace of AI

development can overwhelm organizations, making it difficult to identify and adopt the most appropriate tools for their needs.

Recent studies suggest that AI could help reduce inefficiencies in the software development life cycle. For example, Kokol (2024, 10) highlights how by applying AI to do repetitive tasks, it reduces cognitive load and improves employee well-being. Similarly, Shafiq et al. (2021, 140902) argue that AI-driven tools are increasingly viewed as collaborative aids rather than replacements for human expertise. These tools enable teams to redirect their focus toward innovation and strategic decision-making, enhancing overall productivity.

Hymel (2024, 1, 4) writes that in the future AI tools evolve into unified systems that communicate across SDLC. In such an "AI-native SDLC," tools would interact seamlessly, verifying each other and automating processes throughout the lifecycle. While this idea sounds prominent and could help everyone to do their job better, its realization would require rigorous testing and safeguards, particularly in environments where human oversight is reduced.

The following subchapters examine how artificial intelligence can support specific phases of the software development life cycle. Each section focuses on one phase: requirements, development, testing or maintenance. The aim is to describe both current applications and future possibilities based on existing research literature. These sections provide a foundation for the later empirical analysis by outlining findings and themes presented in earlier literature.

### 2.3.1 AI in Requirements phase

Requirements phase is the first phase in the software development life cycle that defines the goals, functionalities, and constraints of a software system. Integrating artificial intelligence into this phase has been increasingly explored in academic and industry settings, with researchers identifying potential benefits, limitations and future directions. AI techniques offer tools to automate labor-intensive processes, improve requirements quality, and bridge communication gaps among diverse stakeholders. (Shafiq et al. 2021, 140901; Sofian, Yunus & Ahmad 2022, 51027.)

**Improving clarity with NLP and ML**

Natural language processing (NLP) and machine learning (ML) models are particularly effective in identifying unclear or ambiguous requirements. When multiple stakeholders, such as clients, developers, and project managers, are involved, miscommunication can arise due to varying technical expertise or inconsistent formats used to communicate needs. AI tools can analyze and aggregate inputs from diverse platforms, transforming scattered data into a cohesive and clear set of requirements. This reduces ambiguity and improves understanding across teams. (Shafiq et al. 2021,

140897, 140901; Zhang & Tsai 2003, 100–101.) These tools can detect vague language, repetitive phrasing, and incomplete statements, which helps reduce the risk of misunderstandings early in the development cycle. Samek et al. (2017, 2) have warned that without sufficient contextual input, AI-based systems may generate misleading or inaccurate results, especially when applied to loosely structured content.

**Prioritizing and allocating requirements**

Another application of AI in the requirements phase is the prioritization and classification of tasks. In agile projects, requirements are often prioritized based on their value and the effort needed for completion. Machine learning algorithms can streamline this process, saving time and improving team productivity. (Akshatha Nayak, Swarnalatha & Balachandra 2022, 2; Quba, Al Qaisi, Althunibat & Alzu'Bi 2021, 685) For example, in SAFe frameworks, tasks are divided before increments begin to ensure balanced workloads across teams. AI can help allocate tasks efficiently by considering individual expertise and ensuring an even distribution of work, even in complex multi-vendor environments (Barenkamp, Rebstadt & Thomas 2020, 4-5). As development environments become more dynamic and cross-functional, the ability to prioritize and allocate requirements with precision grows increasingly important. AI may not replace the need for human judgment in planning, but it can provide a structured foundation that supports more consistent decisions, especially when coordination must happen quickly and across organizational boundaries.

**Tracing requirements and managing dependencies**

In long and complex projects, tracing code back to its original requirements can become increasingly challenging. The problem becomes more difficult when the original team members are no longer available or when documentation is outdated or incomplete. AI tools can assist by identifying and managing dependencies between features, ensuring seamless integration and maintainability. These tools also improve traceability, which is essential for addressing changes effectively throughout the software lifecycle. (Batarseh et al. 2020, 5–6.)

### 2.3.2 AI in Development phase

The development phase of the software development life cycle involves translating requirements and designs into functional software. Of all the SDLC phases, development likely offers the widest range of AI applications, including improving coding efficiency, reducing errors and streamlining development workflows (Batarseh et al. 2020, 15).

**Automated code generation**

One significant application of AI in development is automated code generation. AI tools enable developers to save time by generating repetitive or boilerplate code, while allowing them to focus on solving more complex problems. NLP models can translate ordinary language input into code, while even visual elements like pictures or graphs can be processed into code structures. Furthermore, AI tools can pre-emptively suggest solutions during coding, acting as intelligent assistants to improve developer efficiency. These tools are designed to collaborate with developers rather than fully replacing manual work, reinforcing AI’s role as a supportive tool. (Odeh, Odeh & Mohammed 2024, 726, 730.)

This view is supported by empirical findings as well. Vaithilingam et al. (2022) found that while developers appreciated the speed and convenience of code generation tools like GitHub Copilot, many faced challenges in understanding, editing and debugging the generated code. Their study emphasized that AI was most useful when it offered a starting point or assistance, rather than trying to produce complete, flawless solutions. This underlines the importance of treating AI-based tools as collaborative aids, particularly for tasks involving judgment or code quality assurance. (Vaithilingam et al. 2022, 5.) This suggests that the success of such tools may depend as much on developer habits and critical thinking as on the tools themselves.

**Enhancing code quality**

Beyond generating code, AI can be applied to improve code quality. Machine learning algorithms can analyze code to detect vulnerabilities, inconsistencies and inefficiencies early in the development process. Predictive models identify potential bugs and recommend fixes, making sure that the software is robust and secure. AI can also be used in pre-implementation code reviews, helping developers refine their work before integrating it with existing features. Additionally, AI can assist junior programmers by suggesting best practices and improving their coding skills. For instance, AI tools can detect "code smells", which are patterns in code that, while not outright bugs, may lead to maintainability or functionality issues in the future. Machine learning models are capable of recognizing these maintainability-related patterns, enabling early intervention. (Saklamaeva & Pavlič 2024, 8–9; Zhang & Tsai 2003, 140901.)

**Code refactoring and documentation**

Refactoring is a necessary part of long-term software maintenance, especially as requirements change or new technologies emerge. These updates help maintain performance and compatibility but are often time-consuming and prone to errors, particularly in large systems. AI tools can detect areas in need of refactoring by analyzing code structures and comparing them to UML diagrams.

These tools also aid in documenting changes, a step that is often overlooked, and ensure thorough regression testing is performed in affected areas. This reduces the likelihood of introducing new errors while refactoring. (Sidhu, Singh & Sharma 2022, 2–4, 11.)

**Promoting Code Reusability**

AI can also support code reusability by identifying opportunities to reuse existing components across projects. For developers, following the "Don't Repeat Yourself" (DRY) principle is key to maintaining efficiency and reducing unnecessary work. AI tools identify reusable components across projects, saving time and ensuring consistency. Reusing existing, tested code not only enhances quality but also ensures that updates to shared components are effectively synchronized. AI tools help developers find relevant code snippets more efficiently by analyzing the context of their work and suggesting examples that fit the current task. This approach makes it easier to locate reusable solutions compared to traditional manual searches, which often involve browsing through documentation or external sources. (Batarseh et al. 2020, 15.)

### 2.3.3 AI in Testing phase

The testing phase in the SDLC is where the developed software is evaluated against both functional and non-functional requirements to confirm it behaves as intended. During this phase, defects are identified, functionality and performance are verified, and overall quality is assessed. However, as software systems grow increasingly complex, testing has become more costly and time-consuming, often consuming up to 50 percent of the total development costs. This makes it a phase that is sometimes overlooked, which can lead to poor-quality software and expensive consequences. Several AI applications aim to make testing more efficient, comprehensive and less dependent on manual input. (Durelli et al. 2019, 1189; Hu 2024, 2.) If AI is used extensively in earlier phases, such as development with autogenerated code, it also increases the importance of testing and requires more thorough test coverage and greater attention to security risks (Hymel 2024, 2).

**AI in test planning and case generation**

Planning is a critical part of testing, as test cases must accurately represent real-world scenarios. It is not enough to design tests for ideal use cases but scenarios where users act in unexpected or unintended ways must also be considered. One key challenge in testing is defining what counts as a correct outcome. Test oracle techniques aim to solve this by automating the process of determining expected results, which are especially valuable in complex or unpredictable scenarios. Machine learning and natural language processing techniques are particularly useful in automating test case creation and prioritizing critical tests, saving time and allowing testers to focus resources on the most critical cases. These tools help testers manage the complexity of modern software features

and allocate testing efforts more effectively. (Amalfitano, Faralli, Hauck, Matalonga & Distante 2023, 8, 21–22; Kotti, Galanopoulou & Spinellis 2023, 17.) However, the effectiveness of these methods often depends on how well the underlying models capture domain-specific logic and user behavior.

**Fault prediction and defect localization**

Fault prediction and defect localization are areas where AI has shown practical value in software testing. By analyzing historical defect data, machine learning models can identify fault-prone modules, enabling testers to address potential issues proactively. For example, Özakıncı and Tarhan(2018, 236) report improved fault prediction accuracy using ML-based techniques, while Hu (2024, 12–13) demonstrates how random forest neural networks can classify test cases and analyze code with high precision. Similarly, Zhou et al. (2019, 204–205) show how models trained on project histories can detect patterns of recurring defects and forecast which components are most likely to fail in future iterations. These methods support more efficient debugging and contribute to greater software reliability. Such approaches are typically most effective when sufficient historical data is available to train the models.

**AI in performance testing**

Performance testing often involves simulating real-world usage patterns to uncover bottlenecks and scalability issues, and AI tools can support this process by adapting to varying conditions automatically. Barenkamp et al. (2020, 7) describe AI-powered load testing tools that dynamically adjust workloads based on real-time performance metrics, enabling testers to identify bottlenecks more effectively. Similarly, Batarseh et al. (2020, 17, 21) highlight the use of AI in stress testing, where it evaluates system resilience under extreme conditions. These applications help ensure that software can respond reliably to both expected and unexpected demands, strengthening overall robustness.

### 2.3.4 AI in Maintenance phase

The maintenance phase of the SDLC focuses on keeping software reliable, adaptable, and performant after its initial deployment. AI technologies can support this phase through techniques such as predictive maintenance, adaptive updates, automated bug handling, and more efficient deployment. These approaches help teams address issues proactively, manage changes more smoothly and maintain software quality as systems evolve.

**Predictive maintenance**

AI supports maintenance by helping teams identify and address issues before they escalate, reducing disruptions and supporting more stable system operation. Machine learning models trained on historical defect data and real-time metrics can detect fault-prone modules, allowing teams to prioritize their efforts and allocate resources efficiently. These tools offer actionable insights that enable targeted interventions, helping to lower maintenance costs and minimize downtime. Batool et al. (2022, 3) and Mariani et al. (2023, 11) emphasize these benefits, while Barenkamp et al. (2020, 2–3) note that the effectiveness of predictive models depends heavily on the availability of high-quality, domain-specific data. In real-world settings, refining and accessing such data can be challenging, especially when information is fragmented or poorly documented.

**Adaptive maintenance**

For adaptive maintenance, AI evaluates the impact of updates on existing systems, ensuring seamless integration of changes. Machine learning aids in tasks like automated refactoring and modularization, simplifying complex code structures and enhancing maintainability (Batool et al., 2022; Barenkamp et al., 2022). These tools are especially valuable as modern software systems must adapt to evolving user and business requirements.

**Automated bug resolution**

AI-powered recommendation systems can analyze error logs and trace the root causes of software issues, helping reduce the time spent on manual debugging. Mariani et al. (Mariani et al. 2023, 2) note that these tools apply natural language processing and machine learning techniques to suggest actionable fixes. However, Sofian et a. (2022, 51031) caution that excessive reliance on automated solutions may diminish engineers' problem-solving skills over time.

**Deployment**

Artificial intelligence can support earlier stages of development in achieving faster and more reliable deployments by improving code quality, accelerating workflows, and helping ensure that features meet defined requirements. However, as software systems grow more complex and clients push for shorter release cycles, the deployment process can become increasingly demanding. In Organization X, multiple environments are used for releasing new features before production to ensure that new updates do not introduce system-breaking changes. AI can also support the deployment phase itself.

Although continuous integration/continuous deployment (CI/CD) generally refers to a fully automated development pipeline, in Organization X it primarily refers to automating the deployment process (the CD component), which can involve numerous steps and configuration variables depending on the environment. Singh and Singh (2023, 7057–7058) suggest several ways AI can enhance deployment. First, AI can assist in build optimization, making deployments faster and more resource efficient. This is particularly relevant for large deployments, which can be time-consuming. In Organization X, deployments frequently occur outside normal office hours, creating additional costs for both the organization and the client. Second, AI can help schedule release timings based on user feedback and usage patterns, minimizing disruptions.

Another potential improvement is that AI can compare code changes, estimate weaknesses, and pinpoint where features may fail. Lastly, software often relies on various code libraries and dependencies, and AI can track these to detect any compatibility issues that require attention. (Mohammed, Saddi, Gopal, Dhanasekaran & Naruka 2024, 531.)

# 3 Case study of utilizing AI in software development life cycle

## 3.1 Organization X

Organization X is a pseudonym used in this thesis to refer to a mid-sized technology and consulting company operating in multiple countries. The organization was established approximately two decades ago and has since expanded its presence across several continents. Its Finnish operations began over ten years ago and currently employ around 200 professionals across several cities.

The company delivers digital solutions and consulting services for both public and private sector clients. In Finland, it has participated in various large-scale projects, including public-sector information system implementations and customized software solutions in domains such as telecommunications, manufacturing and energy. The organization’s service model combines local expertise with international delivery capabilities, which enables it to support complex, multi-stakeholder environments.

In recent years, the company has demonstrated steady growth in its Finnish operations. Industry observers have ranked it among the more rapidly expanding software service providers in the national market. This growth is primarily attributed to its focus on digital transformation initiatives and long-term partnerships with both public and private sector clients.

At the time of the interviews for this study, the company launched a global campaign aimed at enhancing awareness and understanding of AI. The initiative allocated part of employees' personal development budgets to AI-related learning. Staff were encouraged to define personal AI-related goals and reflect on how AI could support their daily tasks. Although the program was still in its early stages during data collection, it reflected a clear strategic intent to promote AI adoption and awareness throughout the organization.

The idea for this research was influenced by observations made within this organizational context, where the adoption of AI tools, especially generative AI, was increasing among developers. However, broader implementation across different roles remained limited. These circumstances indicated a need to examine not only the current state of AI use but also the attitudes, obstacles, and future opportunities related to its integration in software development processes.

## 3.2 The approach and methodology

This thesis uses a case study approach. The primary aim of a case study is to generate fresh insights and ideas that support the improvement of existing practices. A particular case is selected to

collect data and demonstrate a potential model that, if successful, can be adapted more broadly within the organization. The case is examined within its specific context, allowing the phenomenon to be studied thoroughly and in depth. Case studies often rely on qualitative methods such as interviews to gain a better understanding of the situation and the surrounding environment. Other methods, including brainstorming sessions and surveys, may also be used to support the development of ideas. In many cases, the research focus is not fully defined at the beginning but becomes clearer as the study progresses and more information is revealed. A detailed understanding of the current situation is essential for identifying areas that require development. (Moilanen, Ojasalo & Ritalahti 2022.)

In this case, artificial intelligence is still at a very early stage in Organization X and depends on individual employees' interests. Therefore, a good basic understanding needs to be established about what kinds of options there are. Then, it is good to gather information about the attitudes and readiness to utilize AI in everyone's work. The interviews will hopefully clarify these questions, and it may be possible to find some ways to identify effective approaches. In the end, the study will also present suggestions on what an organization can do as the first steps toward AI-powered software development.

The method of this thesis is qualitative research conducted through semi-structured interviews. Surveys were considered but ultimately excluded, as prior internal observations suggested that participation rates within the organization might remain low, despite a relatively large number of professionals working in the software development life cycle. Reaching a sufficient sample size for quantitative analysis could therefore have been difficult. Semi-structured interviews were chosen instead to obtain in-depth insights from selected individuals. This approach also allowed flexibility to adapt follow-up questions based on participants' responses, which was useful given the limited prior knowledge about how AI is currently used or perceived within the organization. Understanding attitudes was also considered important, as these can reveal potential starting points for introducing AI into different phases of the SDLC.

As a novice in conducting interviews, developing the necessary skills for qualitative data collection presented a key challenge during this thesis process. To support the learning process and build confidence in qualitative research, Anne Galletta's book *Mastering the Semi-Structured Interview and Beyond* (2013) was used as a very useful and approachable guide. The book is written in a handbook style, offering clear and concrete advice on how to plan, structure, and carry out semi-structured interviews. It includes practical examples that clarified the different stages of an interview, what kinds of preparations are needed, and how to remain flexible during the interview situation itself. Galletta not only covers technical aspects such as question types and follow-up

techniques but also discusses the researcher’s role and the importance of listening actively and responding thoughtfully to the interviewee's input.

One key point Galletta (2013) emphasizes is the importance of a thorough and focused literature review before constructing the interview guide. A well-conducted literature review helps the researcher to identify gaps in existing knowledge, refine the research topic, and formulate relevant research questions. These questions, in turn, influence the structure and content of the interview. In this study, the literature review supported an understanding of the broader context of artificial intelligence in the software development life cycle and helped in developing questions that were both grounded in previous research and open enough to allow new ideas to emerge. (Galletta 2013, 11–13.)

Galletta (2013) also notes that while a literature review provides the foundation, the interviews themselves may introduce novel insights that have not been discussed in earlier studies. This is especially valuable in a case study context, where the goal is often to gain a deep understanding of a specific situation rather than to test a hypothesis. By combining insights from existing literature with data gathered through interviews, it becomes possible to form a more complete and nuanced understanding of the research topic. In this thesis, that meant identifying not only how artificial intelligence is currently perceived and used in the case organization, but also exploring attitudes, expectations, and concerns that may not be visible in published research. (Galletta 2013, 14–16)

The interview itself is divided into three segments: the opening, the middle, and the end. In the first part, the main goal is to help the interviewee relax and feel comfortable. The questions should be broad and open-ended, allowing the interviewee to describe their experiences with minimal interruption. The interviewer should listen carefully, as the answers given in this phase often form the basis for more precise questions later on. Probing is especially important here, meaning the interviewer should ask for further details or clarification on relevant points. (Galletta 2013, 45–49.)

The middle segment is the most detailed part of the interview. In this phase, the interviewer asks more in-depth questions, often linking them to themes that emerged during the opening segment. There is more variation in the questions at this stage, as they depend on the earlier responses. The aim is to explore the key themes more fully. Looping back to earlier topics helps the interviewer connect responses and deepen understanding. This is also the stage where vague or unclear points from earlier in the interview can be clarified. (Galletta 2013, 49–50.)

The final segment is the concluding part, where the discussion is wrapped up and the interviewee is given an opportunity to address any topics they feel were left unfinished or to share any new thoughts. This is also the point where the interviewer can summarize the discussion and thank the

participant for their contribution. One purpose of this segment is also to allow the participant to return to a more relaxed state, especially after discussing potentially complex or personal topics. In longer interviews that may take place over multiple sessions, this part may last longer. (Galletta 2013, 51–53.)

The structure of the interview questions (Appendix 1) follows the three-part model described by Galletta (2013), which divides a semi-structured interview into an opening, a middle, and a concluding segment. Each interview began with a brief explanation of the purpose of the research and giving the interviewee a chance to ask questions. This part was generally informal and not recorded, as it often included casual conversation to build rapport and help the interviewee feel at ease. Participants were always informed when the actual interview recording would begin.

The first recorded segment included the interviewees introducing themselves, describing their work experience, and outlining their current tasks. The purpose here was to understand their role and link it to the appropriate phases of the software development life cycle. This segment also included a general question about their experiences and attitudes toward artificial intelligence. These early responses helped form a baseline and were referred back to later during more detailed discussions.

The second segment formed the core of the interview and focused on the current and potential future use of AI in different phases of the software development life cycle. This part varied the most between participants, as their experiences and areas of involvement differed. Although the original plan was to interview one person per phase, the discussions proved more meaningful and complete when participants were encouraged to reflect on multiple phases. This approach naturally resulted in longer interviews than anticipated but provided a more comprehensive view of how AI is perceived across the development process.

The final segment focused on wrapping up the conversation. In most cases, this part remained relatively brief. In some cases, participants shared additional thoughts or ideas, which led to a short general discussion about broader AI-related topics. Participants were asked to share feedback on the interview experience itself and discuss any thoughts they had about the process or the topic. These off-the-record conversations also informed reflection and improvement of the interview structure and flow in subsequent sessions.

### 3.3 The interview process

Semi-structured interviews were conducted in February 2025 using Microsoft Teams due to its integrated recording and transcription features. This was convenient, as participants were accustomed to remote meetings, and the digital format provided a relaxed environment, with some even

participating in informal settings such as home offices or while walking. Using Microsoft Teams was beneficial not only logistically but also methodologically. Remote interviews allowed participants to speak in environments where they felt most comfortable, which appeared to encourage openness and reduce potential discomfort, resulting in richer and more candid responses.

The original aim was to interview 10 individuals, each representing a distinct phase of the software development life cycle. However, early discussions indicated that interviewees typically had overlapping experience across multiple phases rather than expertise in only one. Consequently, most participants shared insights on several SDLC phases, which enriched the data collected.

The participants' IT experience varied significantly, ranging from 3 to 28 years. The average experience was approximately 10 years, while the median was 6 years. This broad experience range allowed for capturing both the perspectives of highly seasoned professionals, who could reflect deeply on historical industry developments, and relatively new professionals with fresh insights and expectations about emerging technologies like AI.

Several interviewees were career changers who transitioned to IT roles from entirely different professional backgrounds. Their perspectives provided unique views on AI integration, as they were able to contrast traditional IT workflows with approaches in other industries. This enhanced the depth of the discussions and provided practical viewpoints on the realistic adoption of AI-driven tools and processes.

Although the initial aim was to include equal representation across all SDLC phases, it proved difficult to secure participants who identified testing as their primary area of specialization, despite the presence of QA testers in Organization X. Nevertheless, testing was broadly addressed by other participants, as it is often a cross-cutting activity in agile methodologies. This helped compensate for the underrepresentation of dedicated testers and allowed robust insights into current testing practices.

Table 1: Interviewee matrix

| # | Title | Years of experience | Require-ments | Develop-ment | Testing | Mainte-nance | Primary field |
|---|---|---|---|---|---|---|---|
| **1** | Software developer | 3 | X | X | X | | Development |
| **2** | Senior software de-veloper | 14 | X | X | X | | Development |
| **3** | Consultant | 5 | X | X | | | Development |
| **4** | Product owner | 9 | X | X | X | X | Requirements |
| **5** | Business process developer | 18 | X | | | X | Requirements |
| **6** | Project manager | 28 | X | | | | Requirements |
| **7** | Business consultant | 4 | | | X | X | Maintenance |
| **8** | Software developer | 6 | | X | X | X | Development |
| **9** | Application devel-oper | 3 | | X | | X | Maintenance |

Due to time constraints, all interviews were scheduled within a single month. While this created some pressure in coordinating the sessions, it also brought benefits; conducting the discussions in rapid succession enabled the researcher to reflect on earlier conversations more effectively, adjust in real time, and explore emerging topics more deeply with subsequent participants.

To support open discussion, interviews were conducted in a manner that encouraged comfort and trust. Interviewees appeared candid, openly sharing their experiences and perspectives. Each session began with an introduction to the research objectives and a clear explanation of confidentiality practices. Participants were informed that their identities, specific projects, and Organization X itself would not be explicitly mentioned in the thesis. Consent was confirmed via email prior to each interview, following the distribution of a consent form (Appendix 2), and no objections were raised.

Many participants initially expressed reservations about their own AI expertise. To address this, participants were reassured that detailed AI expertise was not required. The focus was on gathering their opinions and perceptions about AI's future potential within their respective roles. When interviewees found it difficult to provide concrete examples independently, use cases derived from the literature review (Appendix 1) were introduced as conversation starters. This approach

facilitated engaging discussions and enabled participants to express informed views even without extensive prior AI experience.

The data obtained through these interviews was both detailed and diverse. Nevertheless, certain limitations were identified in the research process. The researcher's professional background as a software developer may introduce some bias, particularly regarding the development phase, as greater familiarity exists with development tasks compared to, for example, maintenance or requirements definition. Another potential limitation stems from the rapidly evolving nature of AI. The specific examples introduced from the literature review may become quickly outdated or less relevant over time, given the pace of technological change.

In addition to internal interviewees, two external AI specialists were interviewed to provide broader expert perspectives on AI's current and future role in software development. These specialists were Kalle Mäkelä (AI Lead, Eficode) and Jani Salomaa (AI Capability Lead, Tietoevry). Their contributions offered complementary viewpoints to those gathered within Organization X and helped contextualize the findings in relation to broader industry developments.

.

# 4 Findings

Interviews revealed that employees at Organization X use AI to some extent. Many participants reported using generative AI in both personal and professional contexts, although this use is generally occasional and task-specific rather than systematic. AI is primarily applied in situations where the benefits are immediately apparent, rather than as part of a broader transformation of processes.

This chapter summarizes key insights from the interviews regarding the role of AI in different phases of the software development life cycle. The structure follows the study's three research questions: current AI utilization in the SDLC (RQ1), potential future applications (RQ2), and obstacles to adoption (RQ3).

## 4.1 Current use of AI (RQ1)

This section presents the current use of artificial intelligence across the four selected phases of the software development life cycle in Organization X: requirements, development, testing and maintenance. The findings are based on semi-structured interviews with employees and specialists and structured according to the SDLC phases. Where appropriate, the results are reflected against the literature presented in Chapter 2, to highlight whether existing research aligns with actual practices or if there are differences.

### 4.1.1 Requirements phase

In the requirements phase, which involves gathering and defining what the software should do, AI usage at Organization X is minimal so far. Requirements are typically collected through stakeholder meetings and documented as user stories in natural language. According to the product owner (Interviewee 4), there is no dedicated AI system for automating requirements gathering or analysis at present. However, individuals have started to experiment with AI on their own initiative in small ways.

Interviewee 4 mentioned using a language model-based tool to help clarify and check her notes after client meetings. She explained, "*I sometimes paste my notes into an AI tool to summarize them or highlight possible ambiguities. It's not an official process, but it helps me double-check that we didn't miss anything obvious.*"

This reflects a personal, ad-hoc use of natural language processing to support requirements clarification. These informal uses align with earlier research suggesting that AI can support quality

checks in requirements documentation (Shafiq et al. 2021, 140897). However, such support remains limited to individual initiative, without systematic integration into the requirements process.

Another minor use of AI in this phase has involved requirements classification. Interviewee 5, a business process manager, described an internal experiment with a prototype that categorized user stories by complexity using machine learning. "*We had a prototype, but it was just an internal demo. We haven't deployed it in real projects yet*," he noted. While the system could process large volumes of data and suggest related components, its effectiveness relied heavily on complete and consistently formatted documentation. Without that, the AI sometimes produced suggestions that were not grounded in the actual project context. Prior research has highlighted that AI-generated outputs may become unreliable if the underlying input lacks sufficient structure or context (Samek et al. 2017, 2).

Interestingly, the interviewee 6, a requirements analyst with the longest career in IT among the participants, stated that she had never knowingly used artificial intelligence in either her professional or personal life. Her encounters with AI were limited to internet-based applications, such as AI-generated images or similar content. She expressed a notably skeptical stance toward AI use, stating, "*I don't use AI, as clear as that*." Nevertheless, she acknowledged that AI could potentially assist with complex issues or provide support in certain contexts. Her uncertainty about its relevance to requirements work contrasted with other interviewees, who viewed AI as more suited to routine tasks. While some participants expressed hope that AI could eventually support early-phase activities, most believed that interpreting stakeholder needs and clarifying business logic should remain human-led tasks for now.

Overall, current AI involvement in the requirements phase is limited and informal. It relies on the initiative of employees rather than a mandated organizational tool. The main reason cited for this cautious adoption is the critical importance of requirements accuracy: mistakes early on can be costly later. Participants felt that AI tools for requirements are still immature. For example, AI might misinterpret context or nuance in stakeholder inputs, leading to concerns about trust. These findings are consistent with literature that stresses the importance of precision and traceability in early SDLC phases (Acharya & Sahu 2020, 170). Another practical barrier is data availability. Effective AI applications would require a large, curated dataset of past requirements and outcomes. This is something Organization X has not yet developed. Although some employees have found limited benefits, such as summarizing notes or detecting ambiguity, there remains a strong reluctance to rely on AI for more central activities. This hesitation, largely driven by trust and risk awareness, was a recurring theme across interviews, particularly in the early stages of the SDLC where early decisions affect the entire project.

### 4.1.2 Development phase

In the development phase, where the software is built through activities such as coding, refactoring and code review, the use of AI is more visible compared to the earlier phases. At Organization X, developers have begun to adopt AI-assisted tools that support their day-to-day work. The most frequently mentioned tools were AI coding assistants, such as GitHub Copilot. According to the senior software developer (Interviewee 2), some team members use these tools within their integrated development environments to autocomplete code or generate routine structures.

"*I use an AI coding assistant for about 20 % of my coding tasks, mostly to generate boilerplate code or unit test skeletons. It's like having an extra pair of hands for the tedious parts. It won't design the architecture for me, but it speeds up the small stuff*," he explained.

This quote illustrates how AI currently plays a supporting role by handling repetitive or straightforward tasks, enabling developers to focus more on complex or creative work. Literature supports this pattern. According to Odeh, Odeh & Mohammed (2024, 734) AI tools are increasingly applied to automate basic coding tasks and improve productivity in professional software teams.

Many developers noted that generative AI assistants integrated into development environments have partly taken over tasks previously done via search engines and coding help sites like Stack Overflow. They find AI a faster way to get answers. However, they also pointed out that AI typically provides only one answer, making it the developer's responsibility to verify its relevance and accuracy for the task at hand. Some developers expressed particular concern that junior developers might rely too heavily on AI-generated solutions. A consultant with five years of experience commented on using Copilot:

*"It's easy to use Copilot because it's already integrated into the IDE, like Visual Studio Code. I don't have to switch to Google to ask something. But sometimes it takes a few tries to write a good prompt, and for the AI to understand what I really want."*

This points to a learning curve that has also been discussed in the literature. Vaithilingam et al. (2022, 5) observed that developers often need time to develop effective prompting strategies, which impacts productivity gains in the early stages of adoption.

Another way generative AI was seen as helpful is in explaining unfamiliar concepts, such as a programming language the developer is not proficient in, or in suggesting how to approach a development task. There are often multiple ways to solve a problem, and it can be time consuming to decide which approach is best. Sometimes the developer may not even be aware of a better alternative. In such cases, AI can offer a fresh perspective. This can also be useful in situations where the

client does not allow the use of AI for code generation. The developer can still benefit from AI by using it to support their thinking. As Interviewee 1 explained: *"I mostly use AI to think through how I could improve something I've already built myself. I fine tune it with AI, and of course, when I run into issues, it is an excellent tool to find solutions."*

This aligns with Saklamaeva and Pavlič (2024, 11) who argue that AI-based tools are most effective when used as assistive systems that enhance, rather than replace, developer judgement.

Although not all developers in the organization use AI tools, every interviewee who participated and worked in development had used AI in some form. Adoption remains voluntary and based on personal preference. Some developers have embraced these tools enthusiastically, while others remain cautious or prefer traditional methods. Trust, familiarity, and past experience appear to shape attitudes toward adoption.

In summary, AI in the development phase at Organization X is used in a supportive, assistive capacity. Developers described benefits such as faster routine coding, improved learning, and quicker detection of errors during code review. However, they also noted challenges. These include occasional false positives, variation in trust toward AI-generated output, and the need to integrate these tools smoothly into existing workflows. There is also a learning curve involved, which means that time and training are required before AI tools can be widely and effectively adopted.

### 4.1.3 Testing phase

As mentioned earlier, testing occurs throughout the software development lifecycle. Developers test their code during implementation, and regression tests are typically performed during refactoring. In most projects at Organization X, a dedicated testing phase follows, involving both internal testers and client-side approval testing. In addition, testing continues during the maintenance phase, for example through automated tests executed after releasing a new version to the production environment.

Throughout the interviews, participants described testing as necessary but unappealing work. Developers, in particular, noted that it is time-consuming and often undervalued. Interviewee 1 mentioned that if discipline is lacking in the project, testing is one of the first tasks to be neglected or performed only superficially. This aligns with prior research noting that insufficient testing coverage is a common issue in agile teams, where tight schedules and limited resources often lead to shortcuts (Kotti et al. 2023, 17).

Despite these challenges, AI is not widely utilized during the testing phase at the moment. Only Interviewee 2 reported using AI, specifically when creating use cases for unit tests. For instance,

there might be similar test cases, and once he creates one manually, the AI can generate the rest based on that example. However, AI-generated tests can still contain errors, which subsequently require additional time to correct. Interviewee 2 explained:

*"I have used AI to generate some unit tests and test data. I ran those tests, and they appeared accurate, but I still felt the need to verify them manually. If AI were used for extensive data generation, it would require a great deal of diligence to thoroughly check everything. However, I believe I would tend to just glance over them or skip this verification step altogether."*

Nevertheless, he also emphasized that having some tests, even those generated by AI, is preferable to having none at all. This observation supports Interviewee 1's statement that test cases can easily be overlooked when there is a lack of discipline in the project.

This trade-off between speed and reliability reflects findings by Hu (2024, 2) who emphasized that AI can improve testing productivity but often lacks reliability in more complex scenarios. Interviewee 2’s point also supports Interviewee 1’s earlier statement that test cases are easily overlooked when accountability is weak. In this context, AI was seen not as a replacement for careful test design but as a tool that could offer minimal support when testing discipline is lacking.

In summary, the testing phase or related activities appeared to be the least utilized part of the software development life cycle in terms of AI support. It was also often overlooked in general discussions. This may indicate a potential area where AI could provide substantial value. Automating repetitive testing tasks, generating test cases, or identifying edge cases are all areas where AI tools could improve efficiency and reduce human error. These ideas are supported by earlier research, which highlights the potential of AI-based systems to automate test case generation and prioritization (Amalfitano et al. 2023, 21–22) as well as assist in fault detection and localization through historical data analysis (Hu 2024, 12–13). As such, testing may represent one of the most promising areas for further AI adoption within the software development life cycle.

#### 4.1.4 Maintenance phase

The maintenance phase received notable attention from interviewees, which was somewhat unexpected, as fewer responses had been anticipated regarding this stage. One reason for this is that development and maintenance often overlap. Developers are still involved during maintenance when bugs found in the production environment need to be fixed or when small-scale development tasks are required. Despite this overlap, many interviewees emphasized the importance of distinguishing maintenance as a separate phase, along with its specific activities. The use of AI during maintenance was minimal, even though the phase was seen as important and, in many cases, the longest in the SDLC.

One of the clearest examples of AI use in maintenance was in resolving issues found in production. Interviewee 9 explained that troubleshooting often involves collecting small bits of information from different sources, which can be time-consuming. He noted that AI helped him identify potential causes more efficiently than traditional tools: "*Google can find the error message, but AI can explain how to fix it*." This points to a practical role for AI in narrowing down possible causes and speeding up investigations.

Interviewee 7 also described issue resolution as a frustrating task when the system is unfamiliar and saw AI as potentially helpful in listing recently changed components. While such capabilities are not yet deeply integrated into workflows, some are already in limited use. For example, Interviewee 7 mentioned that AI-generated summaries are occasionally created when closing tickets, although their usefulness depends on the availability of contextual information: "*It's already generating summaries, but if there are no good comments, the result isn't very useful*."

Interviewee 9 saw potential in using AI to support identifying and analyzing bugs in the production environment. When bugs appear in live environments, AI could support maintenance work by analyzing logs, comparing recent changes, or suggesting possible causes. He noted that while tools like Google can help decipher error codes, AI might go further by correlating logs with version history*: "AI could assist in identifying where something broke by looking at logs or recent changes. That's where I see potential."*

Documentation access was another commonly mentioned challenge. Interviewee 8 described the difficulty of locating relevant materials when maintaining or extending features and suggested that AI could assist by retrieving documentation or surfacing previous conversations. Interviewee 4, who regularly prepares release communications, pointed out the challenge of summarizing recent work for clients. She expressed the need for "*a feature to gather together all user stories that have been done and present them clearly in client-understandable language*." While this theme also arose in the requirements phase, here the emphasis was on after-release communication, which plays a distinct role in maintenance.

These findings are in line with earlier studies, which identify fault diagnosis, log analysis and documentation as areas where AI can reduce manual effort and support comprehension (Batool & Khan 2022, 3; Bourque & Fairley 2014, 106). The ability to trace and explain system behavior using past data is particularly valuable in maintenance work, where the original developers may no longer be involved and information may be scattered across systems.

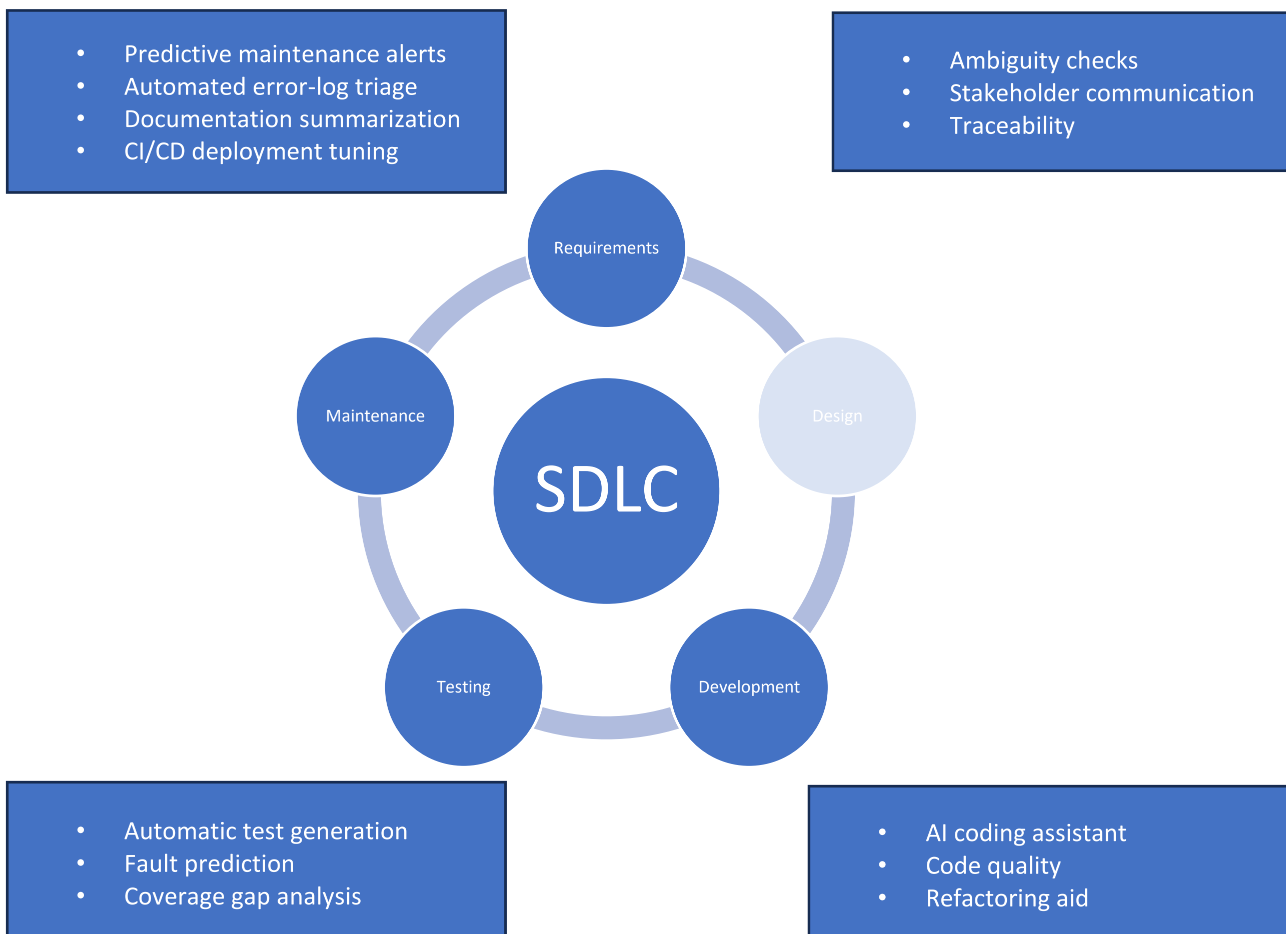
Overall, AI is already playing a limited but concrete role in maintenance at Organization X, particularly in helping interpret error messages, locate information, and summarize completed work. Its

use still depends on individual initiative and is not part of any shared or consistent process. Still, the examples show that AI can support some of the everyday tasks involved in keeping systems running and communicating changes clearly. Although the current role remains small, several interviewees also brought up ideas that point toward a broader use of AI in maintenance. These possibilities are discussed in more detail in the next section.

### 4.2 Possibilities of AI use in the future of SDLC (RQ2)

This section focuses on how artificial intelligence could be used in different phases of the software development life cycle in the future. The findings are based on interviews with employees and AI specialists who reflected on possible use cases, improvements and new ways of working. These ideas are compared with earlier research to highlight where future applications may emerge. Each subchapter is structured around the same four SDLC phases used in the previous section: requirements, development, testing, and maintenance. The figure below presents the main themes identified in each SDLC phase covered in this thesis, highlighting areas where AI has the potential to add value.

Figure 2: AI themes in SDLC



#### 4.2.1 Requirements phase

Many interview participants shared concrete ideas for how AI could support the requirements phase in the future. Their suggestions focused mostly on improving clarity, saving time and reducing the risk of misunderstandings. Several of these ideas built directly on small experiments already done at Organization X or on current pain points like vague requirements and scattered documentation. While AI is not yet widely used in this phase, interviewees had a fairly practical view of how it could help.

One of the most common themes was using AI to detect ambiguity in user stories and requirement documentation. Interviewee 5 described how unclear inputs during meetings or in emails often lead to misunderstandings later: "*In the future, AI could detect inconsistencies or missing information in requirements documentation, saving us a lot of time and confusion down the road*." This concept is consistent with previous research on natural language processing in the context of software development. NLP techniques have been shown to detect vague or contradictory language, improve requirement clarity, and reduce the manual effort of quality assurance reviews (Shafiq et al. 2021, 140901; Zhang & Tsai 2003, 100). These tools are especially useful in agile environments where time and documentation may be limited.

Another frequently mentioned topic was traceability. Several interviewees believed that AI could help create and maintain links between requirements, design elements, tests and implementation. Jani Salomaa viewed this as a realistic use case, where AI could help maintain consistency across documentation and development. Interviewee 2 described the potential as follows: "*AI could significantly improve traceability, automatically linking requirements to their implementation and testing outcomes, ensuring everything stays synchronized even as we make changes*."

This idea is supported by earlier research, which highlights how maintaining traceability links manually is both time-consuming and error prone. Researchers have suggested that AI can assist by continuously updating traceability matrices, helping project teams understand how requirement changes affect downstream work (Pargaonkar 2023, 123; Batarseh et al. 2020, 5–6). Especially in dynamic environments, this type of automated support could reduce overhead while improving transparency.

Kalle Mäkelä brought up the issue that many requirements are not written down formally but are instead hidden in informal communication. He pointed out that many important decisions and ideas are shared in Teams chats or emails rather than in structured documentation. Several interviewees said they had used automatic meeting summaries and found them helpful, even if imperfect. Mäkelä suggested that AI tools could help surface this kind of unstructured information, making it

easier for new team members to understand past decisions or for analysts to collect input from scattered sources.

Other interviewees built on this point by suggesting that AI tools could extract potential requirements or feature requests directly from emails, chat messages, or even transcripts of stakeholder interviews. In this setup, AI could act almost like a junior business analyst, picking up on client needs and drafting initial requirements from everyday communication. While this is not something currently practiced at Organization X, some interviewees said that tools like Teams summaries already make it easier to capture what was discussed. Even if the summaries are incomplete, they are faster to clean up than starting from a blank page.

This stands in contrast to how requirements work is often framed in the literature. Many studies reviewed by Shafiq et al. (2021, 140897) rely on structured inputs such as formal requirements specifications or annotated datasets. Their review demonstrates how NLP can extract requirements from clearly defined documents. However, participants in this study described a more fragmented reality, where important input may be hidden in emails, chats, or loosely written notes. AI tools intended to support requirements work should be able to operate across these layers, from formal documentation to informal conversation. Some participants discussed moving toward more autonomous AI assistance in the future. For example, Salomaa described how independent AI components might handle different tasks, such as generating content, creating test cases, and validating results. Building on this idea, one possible direction is the use of agents that monitor and interpret project communication. These agents could assist in identifying potential requirements, clarifying inconsistencies, or preserving context as it develops over time.

A few participants brought up classification and tagging as an area with clear potential. One team at Organization X had already tested AI-based classifiers, and interviewees saw a path toward automating requirement triage. They envisioned AI categorizing incoming requirements by type, complexity, or criticality, helping analysts spot gaps or prioritize effort. Research by Akshatha et al. (2022, 685) suggests that machine learning can already support requirement classification in academic settings, and interviewees saw a clear path to applying these techniques in real projects.

In several cases, AI's role was imagined as part of a broader support function. For example, Salomaa mentioned that "*AI tools are often useful just by reminding you of things you forgot to consider*", referring to how AI-generated suggestions can help avoid overlooking key areas in planning. Others saw potential in AI-powered documentation support. This included retaining context when team members change or when work continues across multiple sprints. The idea also connects with knowledge management, where AI could help make project history and rationale more accessible over time.

### 4.2.2 Development phase

Interviewees described a wide range of possibilities for how AI could change the development phase in the near future. While many employees at Organization X already make use of AI-driven development and conversational tools, they view these as early signs of a broader transformation. Developers anticipate that such technologies will shift from being occasional helpers to more constant participants throughout the software development workflow. The general hope is that AI will take over routine coding tasks, support quality assurance, assist with architectural work, and even help with documentation and team communication.

A frequently mentioned idea was that AI could generate much of the basic or repetitive code that currently consumes a large share of development time. One senior developer described this as "*the boring 80 percent*," referring to tasks like implementing standard CRUD operations, formatting data, or creating basic forms. Developers would prefer to focus their time on the more complex 20 percent, such as algorithm design, system integration, or performance tuning. Current tools already allow code generation from comments or short natural language descriptions, but interviewees hope that future tools will support full module creation with less manual intervention. This aligns with findings from Vaithilingam et al. (2022, 5), who note that intent-based programming and natural language interfaces are expected to become more prominent in development environments.

However, interviewees also pointed out a major limitation of today's tools: they lack awareness of the full project context. Copilot and similar assistants can suggest code based on general training data or the current file, but they do not understand project-specific constraints or business domain logic. Interviewee 9 noted that "*sometimes it just throws in something completely off-base*," and suggested that connecting AI tools more deeply with project repositories, documentation, and existing APIs could reduce errors. Several developers hoped for AI models trained on internal codebases to avoid irrelevant outputs and ensure alignment with company practices. A possible solution to this problem came up in Kalle Mäkelä's interview, where he pointed out that generic AI models often lack the necessary project-specific knowledge and can produce outdated or misleading suggestions. As a solution, he proposed using retrieval-augmented generation (RAG) techniques, where the AI accesses internal documentation or code to generate context-aware responses. Such an approach could improve the accuracy and usefulness of code assistance, particularly in domain-specific projects.

Jani Salomaa also proposed the idea of using lightweight AI agents to solve subproblems independently. He described a setup where several agents each generate different solution options for a task, and the developer then selects the best one. "*Instead of one big AI model*," he explained, "*you could have multiple smaller ones that each do one thing well*." This agent-based structure

would allow broader exploration of alternatives while still keeping the developer in charge. While speculative, this approach was seen as technically feasible and desirable for tasks like prototyping or experimenting with multiple design paths.

Improving code quality and maintainability was another recurring theme. Interviewee 2 imagined AI systems that continuously review code and highlight areas with high break risk or poor practices. Rather than relying solely on static analysis or unit tests, future AI tools could act as dynamic reviewers that learn from historical defect patterns and give feedback before problems reach production. Zhou et al. (2019, 204–205) support this direction, showing how machine learning models can predict bug-prone areas based on past data.

Participants also discussed AI's potential role in supporting architectural and design work. While current tools rarely operate at this level of abstraction, several developers hoped that AI could one day propose architectural patterns, generate component diagrams, or even produce system-level documentation. Interviewee 9 described a scenario where AI could "*inspect the whole project and produce the application architecture documentation*," which would reduce manual work and support onboarding. Kalle Mäkelä raised a related concern: as AI speeds up development, there is a risk that architectural thinking becomes fragmented or overlooked. He suggested that AI tools should also assist in preserving architectural integrity, not just code productivity. For organizations managing multiple client systems, as is the case with Organization X, these features could improve onboarding and system understanding.

Learning and mentoring were also mentioned as areas where AI might contribute. Interviewee 8 observed that AI could act as a teacher, especially for junior developers, by explaining code or suggesting better practices. However, she also warned that tools must be designed in a way that supports learning rather than enabling passive use. Samek et al. (2017, 5–6) similarly stress the importance of explainability in AI, especially in educational or support scenarios where users need to understand why a suggestion was made. Developers hoped that future systems would not only generate code but also explain their reasoning and refer to the best internal practices or documentation.

On the process level, AI might support coordination and reporting. Interviewee 9 imagined an agent that observes development work and automatically updates tickets or generates summaries. For example, an AI might track file changes, correlate them with tasks in Jira, and generate progress updates for team leads or clients. While this is not strictly a coding task, it is closely related to development workflows and could reduce the administrative burden on developers.

Salomaa also suggested that AI could help bridge the gap between roles such as developers, analysts and architects by providing shared context and surfacing relevant constraints. Interviewee 4 supported this idea, noting that "*we often lose time because the knowledge is not written down or not where it should be*." A well-integrated AI system that understands both technical and business layers could help surface relevant documentation, requirements, or decisions during development. Rather than acting only as a coding tool, AI could serve as a shared knowledge interface that improves collaboration across different roles.

Throughout the interviews, the expectation was that AI will not replace developers but change their role. Developers might spend less time writing code from scratch and more time reviewing, validating and directing AI-generated output. The concept of developers as supervisors of AI aligns with recent studies that describe human-AI collaboration in software tasks as a shared process rather than a handover (Jarrahi 2018, 583). Interviewees believed that this shift could improve productivity, reduce time spent on repetitive tasks and promote better knowledge sharing.

### 4.2.3 Testing phase

The interviewees largely agreed that the testing phase offers considerable opportunities for improvement using AI. Compared to the development phase, where AI tools are already quite prevalent, testing was described as less automated and more labor-intensive. Participants felt AI could increase efficiency and help overcome challenges such as limited test coverage, high maintenance overhead of tests, and slow diagnostic processes. Although existing AI-based testing tools are still evolving, interviewees expressed strong expectations that advancements will be rapid in the near future.

Creating automated test cases has become an exciting and promising area of application. Interviewee 8, who has considerable experience in testing, pointed out that creating thorough unit tests is straightforward in theory but often impractical due to time constraints. He noted that AI could significantly alleviate this workload, particularly for areas like user interface testing, which involve numerous variations. Interviewee 8 stated, "*Creating test cases takes time… AI could significantly increase test coverage."* Others expressed as well that AI could generate test scenarios from feature descriptions or requirement documents, allowing testers to concentrate on reviewing and refining tests rather than creating them from scratch. This is similar with Amalfitano et al.'s (2023, 30–31) findings, where AI systems using natural language processing and machine learning effectively generate test cases based on input-output specifications.

Interviewees also highlighted the potential of AI to identify the parts of a system that are most prone to failure. Interviewee 2 imagined an AI assistant that analyzes project history to detect

error-prone components and recommends targeted regression tests. This would help teams prioritize their efforts and reduce the need for broad, system-wide testing. Similar ideas have been presented in earlier research on fault prediction and defect localization, where historical test data is used to guide testing focus and streamline debugging efforts (Özakıncı & Tarhan 2018, 236).

The maintenance of existing test scripts was another area identified for AI assistance. Participants frequently mentioned the brittleness of automated tests, particularly their tendency to break due to minor interface changes. In this context, AI could detect such alterations and update test scripts accordingly. Instead of failing due to changes like button identifiers, an AI might recognize new labels or positions and adapt the tests. Literature supports this notion, with Amalfitano et al. (2023, 8) describing AI methods that dynamically adjust test scripts, thereby reducing manual effort.

Identifying the cause of test failures was described as particularly challenging. Interviewee 9 suggested that AI could help interpret error logs more effectively than current tools, especially by correlating failures with recent changes. AI systems that can explain failures clearly and point to likely root causes could shorten debugging cycles significantly. The importance of clear explanations also came up in discussions about documentation and learning, where AI transparency was seen as key to building trust and improving test reliability (Amalfitano et al. 2023, 8; Hu 2024, 13).

AI’s role in improving test documentation was another recurring theme. Interviewees noted that documentation is often left incomplete or becomes outdated, which creates issues in traceability, future maintenance and team communication. AI was seen as helpful in generating and updating documentation related to test cases, results, and changes. Interviewee 1 remarked that documentation “*remains incomplete or partially done*,” while Interviewee 2 described it as tedious, arguing that AI-generated documentation could greatly reduce the time and effort required. Kalle Mäkelä noted that AI tools could help track and document changes as they occur, improving consistency in fast-moving development environments.

Jani Salomaa also touched on AI’s future role in testing. He envisioned lightweight agents that could generate test cases, analyze their output and highlight problems. These agents could work alongside developers and testers, supporting early-stage validation without requiring deep integration into the main development pipeline. This type of support could be particularly useful in large or modular systems where responsibility for testing is shared across teams.

Despite these positive expectations, interviewees consistently emphasized the necessity for human oversight. Interviewee 8 cautioned against "*giving AI completely free rein*," emphasizing that human review remains essential. This perspective was shared broadly, envisioning a future where testers have a more strategic role, focusing on reviewing, supervising, and correcting AI-generated

results rather than manual test execution. This aligns with Jarrahi's (2018, 584–585) view that the optimal use of AI occurs in collaboration with human expertise, preserving human control and judgment.

Finally, interviewees discussed AI's potential as a learning tool within the testing domain. Like in development, AI systems could explain their reasoning when suggesting tests, helping junior testers develop their understanding. Referring to earlier test cases, known issues, or domain-specific rules could accelerate onboarding and skill-building. Vaithilingam et al. (2022, 4) support this idea, noting that contextual feedback provided by AI systems can help users understand why a suggestion was made and how to improve it.

### 4.2.4 Maintenance phase

Interviewees described the use of AI during the maintenance phase as relatively limited compared to its application in development and testing. However, participants saw substantial potential for AI to support tasks like debugging, documentation, and understanding legacy systems. Maintenance was often characterized as detail-oriented, time-consuming, and dependent on tacit knowledge, prompting interviewees to suggest that AI could alleviate these burdens by providing quicker access to system knowledge and assisting with fault diagnostics.

Troubleshooting production issues was seen as a significant pain point. Interviewee 9 mentioned that while traditional search methods like Google might help decode error messages, AI could offer more sophisticated assistance. He provided an example where an AI could analyze logs, identify recent changes, and correlate them with errors to pinpoint the cause efficiently. This suggestion aligns with recent research by Samek et al. (2017), which indicates that explainable AI can aid in understanding complex outputs, such as error traces and stack dumps. If AI tools were integrated with version control and deployment histories, they could swiftly link failures to specific code commits or configuration changes, thereby expediting debugging.

Onboarding new team members to unfamiliar systems emerged as another challenging aspect in maintenance. Interviewees pointed out difficulties new developers face when trying to grasp what modules do, how components interact, and where common issues occur. They proposed that AI could help by summarizing system documentation and explaining code behavior interactively. For instance, AI could directly answer queries such as "What does this API accomplish?" or "Which code module manages authentication?" Specialist Kalle Mäkelä emphasized a similar direction, suggesting that AI could serve as a knowledge interface that reduces reliance on individual experience. Literature supports this view, with Jarrahi (2018) and Samek et al. (2017) highlighting AI's potential to enhance decision-making when integrated with specialized knowledge bases.

Several interviewees mentioned the challenges involved in modifying legacy code, suggesting AI could assist in code comprehension and refactoring. Interviewee 4 specifically highlighted how outdated documentation complicates maintenance efforts and increases risk. AI tools capable of summarizing functionalities, identifying obsolete code, and suggesting refactoring options were viewed as highly beneficial. Additionally, these tools could proactively spot outdated dependencies or potential security vulnerabilities. This aligns with findings from Zhou et al. (2019), indicating AI-driven code analysis has successfully identified problematic code areas and opportunities for simplification.

Deployment was another area where participants saw detailed opportunities for AI application. Interviewees suggested AI could streamline deployment processes by automatically assessing release readiness through predictive analytics and past deployment data. For example, an AI system could evaluate the risks of deploying specific changes based on historical failure patterns, current system load, and dependencies between software components. AI-driven deployment tools might also automate rollback procedures if anomalies or performance issues are detected immediately after deployment, thereby reducing downtime and improving overall reliability. Additionally, integration of AI with CI/CD pipelines could enhance automation, minimizing manual intervention and human errors during deployments.

Ticket handling emerged as another area where AI could possibly improve efficiency. Interviewees envisioned AI systems automatically categorizing, prioritizing and assigning maintenance tickets based on their severity and complexity. By using historical data and machine learning algorithms, AI could predict the urgency and potential impact of reported issues, ensuring faster response times for critical problems. Additionally, AI could suggest relevant solutions or previous similar cases to assist maintenance personnel in resolving tickets more swiftly and accurately. This capability would not only streamline the ticket management process but also allow maintenance teams to allocate resources more effectively.

There was broad agreement that full automation of maintenance tasks is unrealistic, though partial automation could significantly ease cognitive loads and build confidence when implementing changes. Like the testing phase, participants underscored that AI-generated suggestions would still require careful human review to ensure alignment with architectural considerations and business logic. The maintainer’s role would increasingly involve validating AI outputs, framing contexts, and coordinating efforts, shifting away from manual code inspections.

This view was also highlighted by Salomaa, who noted that the maintenance phase is particularly vulnerable to the effects of technical debt. He suggested that AI could help maintainers by visualizing system dependencies and flagging high-risk areas before changes are made. When paired with

human oversight, this kind of support could allow safer and more efficient updates in complex systems.

#### 4.2.5 Agent-based AI support across SDLC phases

A growing discussion in the field concerns whether AI will evolve from today's assistive tools toward more autonomous, agent-like systems. One possible direction, raised especially in discussions with Kalle Mäkelä, is that AI tools could gradually take on more independent roles. Mäkelä described a scenario where AI agents act almost like virtual team members, capable of performing multi-step development tasks either independently or alongside humans. These agents would not only respond to instructions but also proactively monitor project work, generate suggestions, and connect fragmented pieces of information. For example, an AI agent might receive a high-level requirement and generate related user stories, code stubs, and draft test cases, all while drawing context from internal documents and past project data.

In discussions with Mäkelä, a key takeaway was that current large language models are limited by outdated or overly generic data. To address this, he emphasized the importance of real-time and context-specific data flows, especially if AI agents are to support decision-making or take on broader responsibilities in the SDLC. One approach Mäkelä supported is the use of retrieval-augmented generation, where an agent consults internal documentation and project history to provide tailored suggestions. This means that the success of future AI agents depends not only on model performance but also on the quality, structure, and availability of internal data.

Mäkelä also highlighted knowledge management as a key requirement for agent-based development, and this was echoed in several of the interviews. Participants often described fragmented information, missing documentation and weak traceability as barriers to applying AI in requirements and maintenance work. At Organization X, project information is frequently scattered across chats and emails, rationales behind decisions are not recorded, and links between requirements, code and tests are incomplete. Mäkelä framed this as part of a broader need for digital maturity. For AI agents to contribute meaningfully, the working environment must support access to the full context behind changes and decisions, both for humans and for AI systems.

He also raised the concern that accelerating development with AI could lead to weaker architectural coherence. According to Mäkelä, future tools should not only help write code faster but also assist in maintaining system-wide consistency and design quality. This suggests that AI agents could eventually support not just implementation tasks but also architectural validation and cross-checking across different layers of the system.

Another important aspect is the relationship between humans and agents. Mäkelä noted that agents may soon be able to generate multiple alternative solutions to a problem, which humans then evaluate. This kind of collaboration can broaden the range of design options without adding to the manual workload. However, it also introduces challenges related to explainability, boundaries, and trust. If an agent modifies a requirement or updates code, it must be able to explain what it did and why, ideally linking its actions to existing documentation or past decisions. Mäkelä emphasized that humans must remain in control and act as reviewers, not just recipients of AI output. For this to work, oversight mechanisms and clear responsibilities are needed so that agents operate within well-defined limits.

Although the long-term vision of autonomous agents is ambitious, early steps will likely involve more limited and practical applications. Mäkelä illustrated examples such as agents that refine user stories, clean up outdated documentation, or check for dependency conflicts during release planning. These agents do not require deep autonomy. What they need is good access to project data and a clear role. Starting with well-scoped pilots would allow teams to build familiarity and identify areas where agents can add value without disrupting core processes.

In a broader sense, Mäkelä proposed a hybrid model of development, where agents handle tasks in a structured, rapid way that resembles a micro–waterfall approach, while humans maintain final decision-making within agile workflows. This model would shift the human role away from routine execution and toward directing, reviewing and integrating the work done by AI. Developers could focus more on architecture, system-wide thinking, and managing AI-enabled workflows, rather than carrying out each individual task themselves.

Ultimately, adopting AI agents could free professionals from some of the most repetitive parts of software development. Although most interviewees did not specifically refer to agents, they shared the view that AI should support, not replace, their roles. Developers hoped to reduce time spent on tasks like scanning logs or updating documentation and instead focus more on solving complex problems. Mäkelä emphasized that this transition will not happen by itself. Organizations will need to rethink roles, onboarding and performance expectations to accommodate new ways of working.

If approached thoughtfully, this evolution could lead not only to more efficient development processes but also to more meaningful and satisfying work. The future outlined by Mäkelä is not one where AI takes over, but one where it becomes a capable, consistent assistant that helps humans achieve better results. The role of AI in the SDLC will ultimately depend not just on technical progress, but on how organizations like Organization X choose to address cultural, structural and knowledge-related challenges as they explore these possibilities and redefine how people and AI collaborate in practice.

### 4.3 Obstacles hindering use in SDLC (RQ3)

Although interviewees expressed strong interest in the future potential of AI, they also identified several obstacles that currently limit its effective use across different phases of the software development life cycle. These challenges are not tied to a specific phase but span the entire SDLC. Based on the interviews, the main themes include data security and privacy concerns, lack of AI-related knowledge and training, trust and reliability issues with AI outputs, technical limitations of current tools such as poor context awareness, and organizational and process-related barriers. This section discusses each of these themes to clarify what kinds of challenges Organization X, and likely similar companies, face when attempting to integrate AI more extensively into their software practices.

#### 4.3.1 Data security and privacy concerns

One of the most frequently mentioned obstacles was the concern about data security and confidentiality when using AI tools. Many AI-powered services such as cloud-based generative AI tools require sending code or data to external servers for processing, which raises alarms regarding sensitive information. At Organization X, developers and managers are wary of exposing client data or proprietary code to AI systems that operate as third-party cloud services. "*I hesitate to use these AI tools because we don't know where our data might end up*," admitted one interviewee, reflecting a common sentiment that uncertainty about data handling limits AI adoption. In some cases, the barrier is not just internal policy but client-imposed restrictions. A consultant gave an example that "*some clients do not allow using AI for code generation at all, due to contractual confidentiality requirements*". This means even if the team sees a clear benefit to using an AI tool, they might be forbidden because the client fears data leakage. Indeed, data privacy concerns are widely reported as a top barrier to adopting generative AI in industry.

The core issue is trust in the AI provider and the risk of unintended data exposure. If a developer pastes a piece of code into an AI assistant to get help, could that code, which might contain sensitive business logic, be seen by others or used to further train the AI without consent? With high-profile incidents of AI tools inadvertently exposing user queries, the caution at Organization X is understandable. This has led to interim measures like only using AI on nonsensitive tasks or anonymizing data before sending it to an AI, which in turn reduces the usefulness of the tool. From a compliance perspective, Organization X also must consider regulations. For example, GDPR if or AI Act if any personal data is involved and intellectual property protection. Without robust guarantees like encryption, on-premises AI solutions, strict privacy policies, data security fears act as a brake on AI adoption.

Additionally, integration with secure environments is a challenge. Much of the company's development and maintenance happens in secure networks or VPNs. Cloud-based AI might not easily integrate into such locked-down environments, or doing so might open new security holes. This forces a choice between convenience and security that the organization resolves in favor of security, thereby sidelining many AI tools. As AI adoption grows, vendors are starting to offer on-premises or private instances to mitigate these issues, but the interviews indicated that until Organization X can use AI with full control over data, many team members will remain uncomfortable leveraging it for real work.

### 4.3.2 Lack of knowledge and skills

A recurring theme in the interviews was the lack of sufficient knowledge or experience in using AI tools effectively. Several participants acknowledged that they did not possess specialized expertise in AI. This gap was not related to the technical or mathematical foundations of AI, but rather to the ability to apply AI tools in practical, everyday tasks. Some participants mentioned they were initially hesitant before the interviews because they did not feel they had what they described as "detailed AI expertise." This suggests a confidence gap, where employees might avoid using available tools due to uncertainty or limited familiarity with effective usage practices.

One specific aspect of this knowledge gap concerns interaction with AI systems, commonly referred to as prompt engineering when using generative tools such as ChatGPT. One developer commented that "*using AI requires learning how to ask the right questions*." This illustrates the need for training on how to formulate queries and problems in ways that AI systems can interpret effectively. Without this skill, early experiences may produce irrelevant results, causing users to abandon the tools. Several interviewees described situations where colleagues tried AI tools a few times, received unhelpful outputs due to unclear prompts, and concluded the tools were not useful. In contrast, those who continued to use the tools and learned how to adjust their input reported improved results. Targeted training and support may therefore be necessary to encourage sustained and effective use.

Another facet of this theme was the limited understanding of AI's capabilities and boundaries. Some interviewees lacked awareness of what current AI systems can and cannot do, leading either to underestimation or overestimation. For example, one developer did not know that AI tools can be used to generate test cases, whereas another assumed AI could handle complex business logic and was disappointed when it failed to deliver reliable results. This mismatch between expectations and actual performance can reduce confidence in the tools. Educating teams about AI's strengths, such as pattern recognition and natural language processing, and its limitations, such as contextual awareness or reasoning in unfamiliar situations, could help ensure more realistic use.

One interviewee mentioned being unsure about how large language models work and expressed hesitation about trusting them. These comments suggest that even a basic level of AI literacy, including understanding known behaviors such as generating inaccurate output or the importance of high-quality input data, is needed for users to incorporate AI tools effectively in their work.

#### 4.3.3 Trust and reliability issues

Even when data security is managed and users know how to operate AI tools, trust remains a central barrier to adoption. Many interviewees expressed hesitation about relying on AI for critical tasks. While related to knowledge, this issue goes further and reflects psychological and organizational reluctance. A recurring view was that AI can assist but should not be in charge. As one developer put it, *“AI is a good servant, but a bad master.”*

Several participants mentioned that AI’s outputs can be inconsistent. One noted that the tool might give a helpful answer in one instance, but something random the next. This unpredictability makes users cautious, particularly in tasks where errors have consequences. Since AI tools rarely explain their reasoning unless prompted, users may find it difficult to assess the reliability of results. Compared to traditional methods such as web searches that present multiple sources, AI often provides a single response, increasing the need for independent verification. This constant need to double-check results was seen as mentally taxing. Some developers reported that using AI sometimes takes more effort than completing the task manually, especially when the output cannot be trusted without scrutiny. A few also pointed out that this risk increases when many suggestions are accepted without review, especially under time pressure.

Concerns about over-reliance were also raised. One senior developer warned that less experienced team members might copy AI-generated code without fully understanding it. This could result in undetected errors or poor design choices. Trust must therefore be balanced: not so low that tools are unused, but not so high that they are followed uncritically. Maintaining human oversight appears important to ensure the reliability and accountability of AI-assisted tasks.

#### 4.3.4 Integration and technical limitations

In addition to human-related challenges such as trust and skills, interviewees also identified several technical and integration-related obstacles. These include compatibility with legacy systems, the limitations of AI in understanding business context, lack of domain-specific data, and difficulties embedding AI tools into established workflows.

One issue raised was that not all projects at Organization X use modern technology stacks, which can affect how easily AI tools can be integrated into daily work. As Interviewee 6 noted, “*in our*

*current project we have outdated technology so can we even utilize A*I?" While tools such as ChatGPT or GitHub Copilot are platform-independent and can be used in most environments, their usefulness may be limited in older systems, particularly where development relies on niche languages, lacks IDE support, or involves minimal automation. For example, integrating AI directly into a continuous delivery pipeline or receiving real-time suggestions in an unsupported editor may not be feasible. In such cases, AI assistance is still possible but may require more manual effort or may be restricted to general tasks like code refactoring, documentation, or exploratory analysis. This creates an uneven landscape in which AI is more accessible in modern greenfield projects and less so in legacy-heavy maintenance work.

A second technical limitation relates to AI's difficulty in understanding organization-specific business logic. While AI can assist with general programming tasks, several participants noted that it often fails when dealing with domain-specific requirements. For example, Interviewee 6 expressed doubt that AI could interpret complex rules tied to her work. Others mentioned the challenge of even formulating useful prompts for tasks involving detailed internal logic. This reinforces earlier concerns about AI's lack of contextual awareness.

Related to this is the absence of relevant training data. As noted in section 4.1, Organization X currently lacks curated datasets of historical requirements, code, or incidents. Without such data, AI tools operate as general-purpose models rather than being adapted to the company's needs. One interviewee remarked that AI would require "*a comprehensive history to produce estimates*." Collecting and preparing this data would take time and resources, especially if privacy or client confidentiality are involved.

#### 4.3.5 Organizational challenges

Beyond individual skills and technical compatibility, some of the challenges identified by interviewees were organizational in nature. These include the absence of clear internal policies, lack of process integration for AI tools, and limited formal support for experimentation or structured rollout.

A key issue mentioned by several participants was the lack of consistent guidance on when and how AI tools should be used. While Organization X had taken steps such as acquiring GitHub Copilot license, there were few shared practices or official recommendations. As one interviewee noted, the company encourages experimenting with AI, but "*in the project itself we can't use it right now*," referring to the absence of a defined framework within the client context. This creates a situation where enthusiasm at the organizational level is not always matched with process-level support in projects.

Interviewees also noted the absence of change management or onboarding processes related to AI. New tools are often introduced informally or by individual initiative rather than as part of a planned transition. This can result in uneven adoption across teams and confusion about best practices. One interviewee pointed out that integrating AI into daily routines often requires adjustments to workflows, yet there was little support available for making those changes. Without structured support, AI tools may be adopted in ad hoc ways, if at all.

In some cases, client processes or contracts also acted as constraints. As a consulting company, Organization X must often align with client practices, and several participants mentioned that client-side restrictions or lack of readiness sometimes prevented AI use. This adds a layer of complexity that internal efforts alone cannot always resolve. Without clear internal processes, even capable individuals struggle to integrate AI meaningfully into workflows. Addressing organizational readiness may therefore be just as critical as solving technical issues.

# 5 Conclusion

This chapter brings together the main insights from the study, focusing on how organizations involved in software development can make more effective use of artificial intelligence across different phases of the software development life cycle. It summarizes the responses to the research questions and presents practical recommendations related to documentation, practices, and skills. It also reflects on how more autonomous, agent-based AI systems may shape development work in the future. Throughout, the emphasis remains on practical ways to support human-led work with AI, based on how it is currently used and how its role may develop in the future.

## 5.1 Key findings

This chapter summarizes the main findings of the case study, responding to the three research questions. The first question (RQ1) examined how AI is currently used in different phases of the software development life cycle. The second question (RQ2) asked how AI could be utilized in the future, and the third question (RQ3) explored factors hindering effective AI use.

The study found AI use at Organization X is heavily concentrated in the development phase. Developers frequently rely on AI-assisted tools to support tasks such as debugging, code generation, and documentation, helping to streamline routine aspects of their work.

By contrast, AI use is minimal and inconsistent in requirements, testing and maintenance. Some participants saw potential for AI to assist in clarifying requirements or generating test cases, but this usage was mostly experimental. In maintenance, AI occasionally helped analyze logs or interpret unfamiliar code, but most tasks were still done manually.

When looking at future possibilities (RQ2), many interviewees believed that AI could have more impact in underutilized areas like testing and maintenance, especially if documentation and traceability were improved. Some participants mentioned the potential of using AI for log analysis, test case suggestions, or document generation, but these ideas remained mostly theoretical or early-stage experiments.

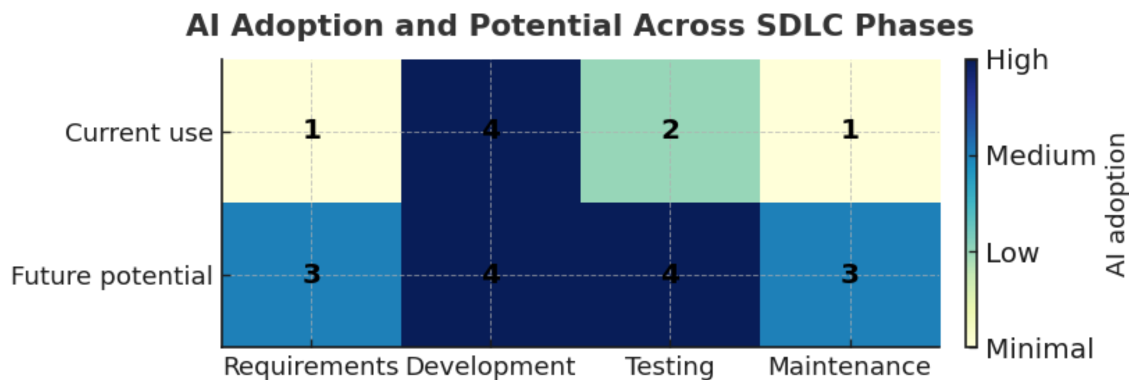
Across all phases, AI use was mostly driven by individuals rather than teams or processes. There were no unified tools, frameworks, or company-level practices in place. Some teams had developers using AI regularly, while others did not use it at all. This led to AI being used in isolation, without shared learning or systematic improvement. Several interviewees felt this was a missed opportunity. Without internal structures such as training, guidelines, or even a safe environment to test AI tools, these efforts remained isolated. This observation supports earlier research showing that AI in

software development is often adopted enthusiastically by individuals but not yet embedded at the organizational level (Giffari et al. 2024; Shafiq et al. 2021).

In response to RQ3, the case study also identified clear barriers that help explain why AI has not taken off more broadly. Security and privacy concerns came up in nearly every interview. Many employees hesitated to send code or client data into external AI systems. This concern was especially strong when tools were hosted in public clouds or lacked clear data handling policies. Some customers had strict rules that explicitly banned the use of AI tools in their projects. These kinds of restrictions meant that even when there was motivation to use AI, external policies made it impossible. In addition, several employees said they simply did not trust the output. AI sometimes makes mistakes or suggests technically correct but practically bad solutions. Without strong validation, these errors could easily end up in production. Combined with a lack of training and confidence, many developers chose to stay cautious.

Poor documentation and missing traceability emerged as key limitations for AI, especially in phases that rely on historical knowledge. These issues will be further addressed in the next section on recommendations. Participants described situations where requirements were only communicated verbally or scattered across emails, Teams messages, and outdated documents. In such cases, it was difficult for either humans or AI systems to determine what the original goals were or why specific changes had been made. In maintenance work, the lack of links between requirements, code changes, test cases, and business context made troubleshooting slow and manual. This reflects what has also been pointed out in earlier research (Shafiq et al. 2021), which emphasizes the role of traceability in making AI tools genuinely useful. When development knowledge is fragmented or undocumented, AI tools struggle to provide meaningful help. Development work, on the other hand, tends to be more localized and self-contained, especially when tasks are limited to individual files or modules. This may partly explain why AI is easier to apply in that phase, even when broader context is missing. The heatmap below illustrates both the current use and identified future potential of AI across different phases of the software development life cycle as identified in this case study.

Figure 3: Heatmap: AI adoption and potential in SDLC phases




**Current use:**

- **Requirements, Testing, Maintenance**: Minimal or low. AI use is rare or experimental, and not yet systematic.
- **Development**: High. AI-assisted tools are already widely used for coding, debugging and documentation.

**Future potential:**

- **Requirements**: Medium. Interviewees saw more potential here than in their current use, but success depends on improvements in knowledge management.
- **Testing, Maintenance**: High. Many interviewees expect AI to deliver much greater value in these areas, especially with better documentation and traceability.
- **Development**: Remains high; AI is expected to continue supporting productivity and automation.

These values are based on a qualitative synthesis of interviews rather than direct quantitative measurements. The heatmap provides a visual overview of where AI is already established and where significant growth is anticipated if organizational and technical barriers are addressed.

### 5.2 Recommendations based on findings

Considering these findings, it can be concluded that AI at Organization X is generally seen as a supportive aid rather than a replacement for human expertise. Developers and other professionals find current AI tools helpful for automating repetitive or tedious tasks, such as generating boilerplate code or sorting through logs. However, they do not entrust AI with high-level decision-making or creative work. This perspective aligns with the concept of augmentation in the AI literature: AI is used to support human work by handling well-defined subtasks, which can improve productivity without replacing the human role entirely. Many interviewees noted that critical activities such as

understanding customer needs, designing architecture, and making final judgments should remain human-led.

Trust and organizational culture play a big role here. Since the technology is still relatively new to the organization, a cautious attitude remains. People are interested in using AI but need time and experience to build confidence that it will behave reliably and securely. At this stage, human–AI collaboration is clearly led by the human, who reviews and validates AI-generated results. Over time, trust may grow if the tools consistently deliver reliable and transparent outcomes. This balanced approach to trust, which avoids both blind reliance and total skepticism, matches earlier findings that successful AI integration depends on maintaining human oversight and accountability.

From a practical standpoint, the research leads to several recommendations that can be applied in organizations of various types and sizes that are involved in software development. These focus on improving readiness, addressing existing barriers and enabling broader adoption:

**Improve documentation and knowledge sharing**

A key step is to systematically capture and organize internal knowledge. This includes keeping requirements documents, design decisions, and code comments up to date, along with recording the rationale behind changes. Better documentation and traceability create the data environment needed for more advanced AI use. In the future, techniques such as retrieval-augmented generation could help AI assistants retrieve and apply relevant project context. However, this kind of solution only works if those documents and records exist in the first place.

AI tools can also support documentation work directly. For example, they can automate the generation of detailed technical descriptions, summarize project changes, and improve traceability by linking related documents and code changes. These features can reduce the manual effort needed to maintain quality documentation and make knowledge easier to access during development and maintenance.

Effective AI-driven documentation and traceability tools depend on robust, comprehensive, and accurately maintained records. Proactively investing in structured project archives and knowledge bases can therefore create a positive feedback loop. As documentation quality improves, AI systems can work more effectively, which in turn helps keep documentation up to date. Finding suitable AI tools to support these tasks should be considered an early priority when planning AI adoption in software development.

**Establish clear AI usage guidelines and secure tools**

Organizations should define guidelines, policies and technical frameworks for safe and productive AI use. These should address questions like: What types of data can be entered into AI tools? Which services have passed security reviews? How should teams validate AI-generated outputs before using them? Addressing these concerns helps reduce uncertainty and hesitation. Security is especially important. Considering privacy risks, the company could explore on-premises or private AI solutions that ensure sensitive code and client data stay within internal systems. For example, an internal coding assistant running on secure servers could allow developers to use AI confidently. Similarly, sandbox environments can be used to safely test new tools. With well-defined rules and technical support in place, AI use can shift from personal experimentation to a reliable part of daily workflows.

**Build employees' AI skills and grow trust**

Organizations should invest in training, internal knowledge sharing and change management. Workshops, peer support, or short demo sessions can help employees learn how to use AI effectively and avoid common mistakes. Improved AI literacy helps teams understand both what AI can do and where it still falls short. It is just as important to address cultural concerns. Using AI should be seen as a way to improve work, not as a threat to job value. Leadership should communicate that AI is meant to reduce routine tasks so that employees can focus on higher-value work. Highlighting team successes where AI helped meet deadlines or solve problems can encourage wider adoption. At the same time, expectations should be clear. Human review is still necessary. Joint review practices can help teams build trust together and reduce overreliance on AI. As people gain experience, confidence grows, and AI can start to feel like a trusted co-worker.

**Pilot AI in targeted, low-risk projects**

A good way to proceed is to start with small, clearly defined pilots in specific parts of the SDLC. For instance, one pilot could focus on generating test data. Another could involve analyzing logs in maintenance to flag likely issues. A third might use AI to summarize requirements or design documents to help with onboarding or reporting. These pilots should have limited scope so that risks and mistakes remain manageable. Each pilot should also include criteria for success and a feedback loop. Teams can then evaluate what worked and what did not and use that experience to improve processes. Early positive results can help build momentum. Challenges can be used to refine internal guidelines. Through this step-by-step approach, organizations can move from scattered personal use toward consistent, organization-wide AI adoption while still managing risks and expectations.

### 5.3 Future direction: AI agents and autonomous support

Looking ahead, many of the tools discussed in this thesis may evolve toward more autonomous and coordinated forms of support. Instead of operating as isolated assistants answering individual prompts, future AI systems may act more like agents with defined goals and roles across the software development life cycle. These agents could be handle specific tasks such as generating implementation options, creating test cases, reviewing documentation, or validating design choices, and they could operate either in sequence or in parallel depending on the context.

This shift would mean moving from reactive usage, where a human asks a question and receives an answer, toward proactive systems that can take initiative based on available context and data. In the development phase, one agent might write boilerplate code while another reviews it for known issues or style inconsistencies. In testing, an agent could automatically suggest coverage improvements or identify risky areas based on historical bug patterns. In the requirements phase, agents could help transform vague business goals into more concrete user stories by drawing from organizational knowledge bases and previous solutions.

Realizing this capability requires advances in model design, access to contextual data, and a shift in how organizations approach AI adoption. Agents will need rules, boundaries, and clear accountability structures to ensure their outputs are transparent and correct. Human–AI collaboration will remain essential. Even if agents handle many technical tasks, the ability to interpret goals, manage ambiguity, and make final decisions will still rely on people. The diagram below outlines one possible division of responsibilities between human developers and AI agents across key SDLC phase.

Table 2: Example division of responsibilities between human experts and AI agents across SDLC phases

| SDLC phase | Human expert | AI agent |
|---|---|---|
| **Requirements** | • Clarifies requirements<br>• Writes user stories | • Checks requirements<br>• Suggests missing details using NLP |
| **Development** | • Codes features<br>• Reviews AI-generated code | • Generates code<br>• Proposes refactoring |
| **Testing** | • Designs manual test cases<br>• Executes tests | • Generates test cases<br>• Analyzes results<br>• Identifies risky areas |
| **Maintenance** | • Applies fixes<br>• Reviews AI suggestions | • Predicts incidents<br>• Suggests documentation improvements |

If developed and deployed carefully, agent-based AI could improve both the efficiency and quality of software development. It could reduce manual work, support more consistent practices, and help teams manage increasing complexity. The long-term success of these systems, however, will depend on more than technical progress. Organizations will need to invest in documentation, establish secure infrastructure, and cultivate a mindset that supports experimentation while maintaining human oversight. These foundations will be essential as AI transitions from a tool into an active participant in software work.

## 5.4 Ethics and reliability

The research process is described as clearly as possible throughout this thesis, covering the interview structure, analysis methods, and ethical considerations. Semi-structured interviews were used, and all participants received the same question framework, although the discussion naturally varied based on each person's background and their involvement in different SDLC phases. This approach made it possible to gather a wide range of perspectives while still allowing answers to be compared across roles. For analysis, a thematic approach was applied. Responses were organized according to the three research questions and the relevant SDLC phases. Often, interviewees touched on several phases at once, so the material was grouped in a way that made sense for each theme. All phases of the SDLC were addressed in the analysis to provide a balanced view of AI’s role throughout the software life cycle.

Regarding ethics, participants were fully informed about the purpose of the study and gave their consent before the interviews began. They were reminded of their right to withdraw at any time, though no one did. Interviews were recorded and anonymized and no sensitive or identifiable information was included in the thesis. The material will be deleted within three months of the thesis being approved. Discussions mostly remained on a general and professional level, with participants accustomed to talking about their work without sharing confidential details. Participants were not asked to review the transcripts or summaries after the interviews. This decision was made because the topics remained professional and non-sensitive. Everyone was informed that they could reach out afterward if needed, but no such requests were made.

Although efforts were made to keep the analysis clear and balanced, the researcher’s professional background may have influenced which themes were noticed or emphasized. As with many qualitative studies, the findings reflect one organizational and time-specific context and are not intended to be generalized beyond similar environments. The findings contribute to a broader understanding of AI’s role in software development work.

### 5.5 Future research

Based on the findings of this thesis, there are several directions that future research could take. First, it would be valuable to conduct similar studies in other types of organizations, especially outside the IT consulting sector. Companies with their own in-house development may face different challenges and opportunities when adopting AI, so comparative research could help clarify which findings are generalizable and which are context-specific.

Second, there is a need for more empirical research on the concrete impact of AI adoption in different phases of the software development life cycle. While this thesis highlighted potential benefits, there is limited real-world evidence on how much AI actually improves areas such as productivity, quality, documentation, or onboarding. Carefully designed case studies or quantitative assessments could provide clearer answers about what changes when AI is introduced in practice.

Finally, future research should pay attention to organizational change. Many of the obstacles identified in this study relate to processes, structures, or workplace culture rather than technology itself. Understanding how organizations adapt their ways of working, manage change, and support their employees during AI adoption could help both practitioners and researchers address these broader challenges.

Research in these areas would help build a more complete understanding of how AI can be used effectively in software development and what kinds of changes are needed to realize its potential.

### 5.6 Self-reflection

Before starting this thesis, I had limited experience with qualitative research methods. Learning to design, conduct and analyze interviews was new and sometimes demanding. Altogether, I carried out nine IT-expert interviews and two specialist interviews on AI in a relatively short period. Processing the material was time-consuming, but choosing a topic that genuinely interested me made the effort easier to manage.

One of the most rewarding aspects was discussing AI with professionals from different roles in the SDLC. As someone who identifies primarily as a developer, I found it particularly interesting to talk with people involved in requirements engineering and hear about their perspectives and day-to-day challenges. These conversations broadened my own understanding and have made me consider expanding my skill set toward business analysis. This has also influenced my motivation to pursue an MBA degree, as I am interested in acting as a bridge between technical and business roles.

I was also struck by the depth of thought my interviewees brought to the topic. Even those who initially claimed to have little direct experience with AI had clearly considered its implications and were following developments closely. This reinforced my view that practical experience is only one dimension of expertise; curiosity and reflection are equally important.

As the project progressed, I became more comfortable with conducting interviews. After the first few, I started to recognize recurring themes during the conversations themselves, which made it easier to guide the discussion and ask follow-up questions. My ability to structure and analyze qualitative data clearly improved over the course of the study.

The pace of development in AI is remarkable, and it shows no signs of slowing down. While writing this thesis, new topics such as AI agents and retrieval-augmented generation appeared in rapid succession, moving quickly from technical blogs into mainstream discussion. Staying up to date became a task in itself, and I chose to include these emerging themes in the study. For anyone working in software development, it is clear that AI will only become more deeply integrated into everyday work. Rather than waiting for changes to arrive, we need to prepare and keep learning continuously. I hope this thesis encourages others to approach AI not only with curiosity, but also with a readiness to adapt and develop new ways of working as the field evolves.

## References

Acharya, B. & Sahu, K. 2020. Software Development Life Cycle Models: A Review Paper. International Journal of Advanced Research in Engineering and Technology, 11, 12, pp. 169–176. URL: https://doi.org/10.34218/IJARET.11.12.2020.019. Accessed: 19 September 2024.

Akshatha Nayak, U., Swarnalatha, K.S. & Balachandra, A. 2022. Feasibility Study of Machine Learning & AI Algorithms for Classifying Software Requirements. MysuruCon 2022 - 2022 IEEE 2nd Mysore Sub Section International Conference. URL: https://doi.org/10.1109/MYSURUCON55714.2022.9972410. Accessed: 25 November 2024.

Amalfitano, D., Faralli, S., Hauck, J.C.R., Matalonga, S. & Distante, D. 2023. Artificial Intelligence Applied to Software Testing: A Tertiary Study. ACM Computing Surveys, 56, 3. URL: https://doi.org/10.1145/3616372. Accessed: 13 September 2024.

Angerer, F., Grimmer, A., Prähofer, H. & Grünbacher, P. 2019. Change impact analysis for maintenance and evolution of variable software systems. Automated Software Engineering, 26, 2, pp. 417–461. URL: https://doi.org/10.1007/S10515-019-00253-7/FIGURES/17. Accessed: 2 December 2024.

Barenkamp, M., Rebstadt, J. & Thomas, O. 2020. Applications of AI in classical software engineering. AI Perspectives, 2, 1. URL: https://doi.org/10.1186/s42467-020-00005-4.

Batarseh, F.A., Mohod, R., Kumar, A. & Bui, J. 2020. The application of artificial intelligence in software engineering: a review challenging conventional wisdom. Data Democracy: At the Nexus of Artificial Intelligence, Software Development, and Knowledge Engineering, pp. 179–232. URL: https://doi.org/10.1016/B978-0-12-818366-3.00010-1. Accessed: 13 September 2024.

Batool, I. & Khan, T.A. 2022. Software fault prediction using data mining, machine learning and deep learning techniques: A systematic literature review. Computers and Electrical Engineering, 100. URL: https://doi.org/10.1016/j.compeleceng.2022.107886.

Bourque, P. & Fairley, R.E. 2014. Guide to the software engineering body of knowledge : SWEBOK, version 3.0. IEEE Computer Society. [Washington, D.C.] : Accessed: 20 November 2024.

Dong, C., Li, Y., Gong, H., Chen, M., Li, J., Shen, Y. & Yang, M. 2023. A Survey of Natural Language Generation. ACM computing surveys, 55, 8, pp. 1–38. URL: https://doi.org/10.1145/3554727. Accessed: 4 January 2025.

Durelli, V.H.S., Durelli, R.S., Borges, S.S., Endo, A.T., Eler, M.M., Dias, D.R.C. & Guimarães, M.P. 2019. Machine learning applied to software testing: A systematic mapping study. IEEE Transactions on Reliability, 68, 3, pp. 1189–1212. URL: https://doi.org/10.1109/TR.2019.2892517. Accessed: 13 September 2024.

Feuerriegel, S., Hartmann, J., Janiesch, C. & Zschech, P. 2024. Generative AI. Business and Information Systems Engineering, 66, 1, pp. 111–126. URL: https://doi.org/10.1007/S12599-023-00834-7. Accessed: 4 January 2025.

Galletta, Anne. 2013. Mastering the semi-structured interview and beyond : from research design to analysis and publication. New York University Press. New York : Accessed: 5 April 2025.

Giffari, R., Ridho, M.M., Sensuse, D.I., Hidayat, D.S. & Purwaningsih, E.H. 2024. Analyst's Perception on the Use of AI-based Tools in the Software Development Life Cycle. Jurnal Sistem Informasi, 20, 1, pp. 73-87–73–87. URL: https://doi.org/10.21609/JSI.V20I1.1399. Accessed: 19 September 2024.

Grzybowski, A., Pawlikowska–Łagód, K. & Lambert, W.C. 2024. A History of Artificial Intelligence. Clinics in Dermatology, 42, 3, pp. 221–229. URL: https://doi.org/10.1016/J.CLINDERMATOL.2023.12.016. Accessed: 10 December 2024.

He Junda, Treude Christoph & Lo David 2024. LLM-Based Multi-Agent Systems for Software Engineering: Literature Review, Vision and the Road Ahead. ACM Transactions on Software Engineering and Methodology. URL: https://doi.org/10.1145/3712003. Accessed: 12 May 2025.

Hu, Y. 2024. Research on Artificial Intelligence-Assisted Software Test Automation Methods. Applied Mathematics and Nonlinear Sciences, 9, 1. URL: https://doi.org/10.2478/amns-2024-2874.

Hymel, C. 2024. The AI-Native Software Development Lifecycle: A Theoretical and Practical New Methodology. URL: https://arxiv.org/abs/2408.03416v3. Accessed: 20 November 2024.

Jarrahi, M.H. 2018. Artificial intelligence and the future of work: Human-AI symbiosis in organizational decision making. Business Horizons, 61, 4, pp. 577–586. URL: https://doi.org/10.1016/J.BUSHOR.2018.03.007. Accessed: 27 November 2024.

Kämäräinen, J. 2023. Koneoppimisen perusteet. Otatieto. [Helsinki] : Accessed: 4 January 2025.

Kelleher, J.D. 2020. Syväoppiminen. Terra Cognita. Helsinki : Accessed: 4 January 2025.

Kokol, P. 2024. The Use of AI in Software Engineering: A Synthetic Knowledge Synthesis of the Recent Research Literature. Information (Switzerland), 15, 6. URL: https://doi.org/10.3390/INFO15060354. Accessed: 13 September 2024.

Kotti, Z., Galanopoulou, R. & Spinellis, D. 2023. Machine Learning for Software Engineering: A Tertiary Study. ACM Computing Surveys, 55, 12. URL: https://doi.org/10.1145/3572905/ASSET/9385A36B-0633-4AA3-B4EE-7265EC6F3691/ASSETS/GRAPHIC/CSUR-2021-0747-F06.JPG. Accessed: 14 September 2024.

Kühl, N., Schemmer, M., Goutier, M. & Satzger, G. 2022. Artificial intelligence and machine learning. Electronic Markets, 32, 4, pp. 2235–2244. URL: https://doi.org/10.1007/S12525-022-00598-0. Accessed: 10 December 2024.

Mariani, M.M., Machado, I., Magrelli, V. & Dwivedi, Y.K. 2023. Artificial intelligence in innovation research: A systematic review, conceptual framework, and future research directions. Technovation, 122, pp. 102623. URL: https://doi.org/10.1016/J.TECHNOVATION.2022.102623. Accessed: 28 November 2024.

McCarthy, J. 2006. A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence: August 31, 1955. The AI magazine, 27, 4, pp. 12–14. URL: https://doi.org/10.1609/aimag.v27i4.1904. Accessed: 18 May 2025.

Mohammed, A.S., Saddi, V.R., Gopal, S.K., Dhanasekaran, S. & Naruka, M.S. 2024. AI-Driven Continuous Integration and Continuous Deployment in Software Engineering. 2024 2nd International Conference on Disruptive Technologies, ICDT 2024, pp. 531–536. URL: https://doi.org/10.1109/ICDT61202.2024.10489475. Accessed: 6 January 2025.

Moilanen, T., Ojasalo, K. & Ritalahti, J. 2022. Methods for development work : new kinds of competencies in business operations. BoD - Books on Demand. Helsinki, Finland : Accessed: 5 April 2025.

Odeh, A., Odeh, N. & Mohammed, A.S. 2024. A Comparative Review of AI Techniques for Automated Code Generation in Software Development: Advancements, Challenges, and Future Directions. TEM Journal, 13, 1, pp. 726–739. URL: https://doi.org/10.18421/TEM131-76. Accessed: 13 September 2024.

Özakıncı, R. & Tarhan, A. 2018. Early software defect prediction: A systematic map and review. Journal of Systems and Software, 144, pp. 216–239. URL: https://doi.org/10.1016/J.JSS.2018.06.025. Accessed: 21 November 2024.

Pargaonkar, S. 2023. A Comprehensive Research Analysis of Software Development Life Cycle (SDLC) Agile & Waterfall Model Advantages, Disadvantages, and Application Suitability in Software Quality Engineering. International Journal of Scientific and Research Publications, 13, 8, pp. 121. URL: https://doi.org/10.29322/IJSRP.13.08.2023.p14015. Accessed: 19 September 2024.

Quba, G.Y., Al Qaisi, H., Althunibat, A. & Alzu'Bi, S. 2021. Software Requirements Classification using Machine Learning algorithm's. 2021 International Conference on Information Technology, ICIT 2021 - Proceedings, pp. 685–690. URL: https://doi.org/10.1109/ICIT52682.2021.9491688. Accessed: 18 May 2025.

Ruparelia, N.B. 2010. Software development lifecycle models. ACM SIGSOFT Software Engineering Notes, 35, 3, pp. 8–13. URL: https://doi.org/10.1145/1764810.1764814. Accessed: 19 November 2024.

Russell, S.J. & Norvig, P. 2016. Artificial intelligence : a modern approach. Pearson Education Limited. Harlow : Accessed: 4 January 2025.

Saklamaeva, V. & Pavlič, L. 2024. The Potential of AI-Driven Assistants in Scaled Agile Software Development. Applied sciences, 14, 1, pp. 319-. URL: https://doi.org/10.3390/app14010319. Accessed: 25 November 2024.

Samek, W., Wiegand, T. & Müller, K.-R. 2017. Explainable Artificial Intelligence: Understanding, Visualizing and Interpreting Deep Learning Models. URL: https://arxiv.org/pdf/1708.08296. Accessed: 4 May 2025.

Sami, M.A., Waseem, M., Rasheed, Z., Saari, M., Systä, K. & Abrahamsson, P. 2024. Experimenting with Multi-Agent Software Development: Towards a Unified Platform. URL: https://arxiv.org/abs/2406.05381v1. Accessed: 11 December 2024.

Shafiq, S., Mashkoor, A., Mayr-Dorn, C. & Egyed, A. 2021. A Literature Review of Using Machine Learning in Software Development Life Cycle Stages. IEEE access, 9, pp. 140896–140920. URL: https://doi.org/10.1109/ACCESS.2021.3119746. Accessed: 13 September 2024.

Sharma, M.K. 2017. A study of SDLC to develop well engineered software. International journal of advanced research in computer science, 8, 3. Accessed: 23 November 2024.

Sidhu, B.K., Singh, K. & Sharma, N. 2022. A machine learning approach to software model refactoring. International Journal of Computers and Applications, 44, 2, pp. 166–177. URL: https://doi.org/10.1080/1206212X.2020.1711616. Accessed: 27 November 2024.

Singh, A. & Singh, P. s.a. LEVERAGING ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING IN SOFTWARE ENGINEERING: CURRENT TRENDS AND FUTURE DIRECTIONS. URL: https://doi.org/10.56726/IRJMETS40422. Accessed: 6 January 2025.

Sofian, H., Yunus, N.A.M. & Ahmad, R. 2022. Systematic Mapping: Artificial Intelligence Techniques in Software Engineering. IEEE Access, 10. URL: https://doi.org/10.1109/ACCESS.2022.3174115.

Suri, S., Das, S.N., Singi, K., Dey, K., Sharma, V.S. & Kaulgud, V. 2023. Software Engineering Using Autonomous Agents: Are We There Yet? 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1855–1857. URL: https://doi.org/10.1109/ASE56229.2023.00174<BR>. Accessed: 12 May 2025.

Vaithilingam, P., Zhang, T. & Glassman, E.L. 2022. Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models. Conference on Human Factors in Computing Systems - Proceedings. URL: https://doi.org/10.1145/3491101.3519665/SUPPL_FILE/3491101.3519665-TALK-VIDEO.MP4. Accessed: 4 May 2025.

Zhang, D.U. & Tsai, J.J.P. 2003. Machine Learning and Software Engineering. Software Quality Journal, 11, pp. 87–119. Accessed: 13 September 2024.

Zhang, R., Du, H., Liu, Y., Niyato, D., Kang, J., Sun, S., Shen, X. & Poor, H.V. 2024. Interactive AI with Retrieval-Augmented Generation for Next Generation Networking. IEEE Network. URL: https://doi.org/10.1109/MNET.2024.3401159. Accessed: 22 May 2025.

Zhou, T., Sun, X., Xia, X., Li, B. & Chen, X. 2019. Improving defect prediction with deep forest. Information and Software Technology, 114, pp. 204–216. URL: https://doi.org/10.1016/J.INFSOF.2019.07.003. Accessed: 4 May 2025.

# Appendices

## Appendix 1. General question structure of semi-structured interviews

### 1. Introduction and background

- Could you tell me a little about your role, background, and current responsibilities?
- How many years of experience do you have in software development, and specifically in your current role?

### 2. Current AI awareness and usage

- How familiar are you with artificial intelligence in general?
- Have you encountered or used AI either professionally or personally? Could you give examples?
- What's your general attitude towards AI? Do you view it positively or negatively, and why?

### 3. AI in Software Development Life Cycle (Repeated individually for Requirements, Development, Testing, and Maintenance phases)

- Which phases of the SDLC are you primarily involved in?
- Are you currently using AI technologies or methods in this phase?
- If yes: Could you describe how AI is being used and provide specific examples?
- If no: Do you think there are opportunities for using AI in this phase, and why hasn't it been adopted yet?
- What specific benefits have you observed (or anticipate) from using AI in this phase?
- What challenges or limitations have you encountered (or foresee) with AI in this phase?

### 4. Future potential and opportunities for AI

- How do you envision the use of AI developing in your area of expertise in the near future?
- Are there specific tasks or activities you wish AI could support or automate more effectively?
- What changes would be needed to realize these future opportunities?

### 5. Organizational context and attitudes

- How does your organization currently approach the adoption and implementation of AI?
- Do you have any concerns related to AI use?

### 6. Conclusion and additional thoughts

- Do you have anything additional you'd like to share regarding AI use in your work or software development generally?

## Appendix 2. Consent form (translated to English)

### Consent form

I give my consent to participate in the interview for Juuso Liljavirta's research, which investigates the utilization of artificial intelligence during the software development lifecycle. The interview will be conducted via Teams and recorded. Teams' automatic transcription tools will be used to assist. All identifying information about the interviewee, such as their name, will be removed from the transcription. Similarly, if the interviewee mentions anything sensitive during the interview, they can request its removal or anonymization. This includes names of client companies and individuals. The researcher will keep the files on their own computer.

The interview and transcription will be deleted no later than three months after the thesis has been approved.

I have had the opportunity to ask questions and have received sufficient answers to all my queries.

I understand that participation in the research is voluntary. I am aware that I can withdraw my consent at any time without giving a reason and, for example, interrupt the interview if I wish.

Consent can be withdrawn via email. Please note that once the research results have been analysed, the contribution of an individual participant cannot be removed.

For more information about the research, please contact the researcher: Juuso Liljavirta, email, phone number.

By signing, I confirm that I give my consent to participate in the research.

______________________________

Name of consent giver

______________________________ ______________________________

Date Signature