TECHNISCHE UNIVERSITÄT
CHEMNITZ

Programmer's Block at Development Handoffs:

An Exploratory Study of Developer Experiences and AI-Assisted Problem Solving

Master Thesis

Submitted in Fulfilment of the
Requirements for the Academic Degree M.Sc.

Study Programme: Web Engineering
Faculty of Computer Science
Professorship of Software Engineering

Submitted by: Kunal Rokde
Student ID: -
Date: 17.08.2026
First examiner: Prof. Dr. JanetSiegmund
Second examiner/Supervisor: Belinda Schantong, M.Sc.

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Professor Janet Siegmund and Belinda Schantong, for giving me the opportunity to explore this research topic and for their continuous guidance, feedback, and support throughout the thesis. Their thoughtful questions and constructive comments helped me refine my ideas, approach the research more critically, and find direction whenever I was uncertain.

I am also grateful to everyone who participated in the survey and interviews. Their time, openness, and willingness to share their experiences made this research possible and provided the foundation for this thesis.

Finally, I would like to thank my family and friends for their constant encouragement, patience, and support throughout this journey. Their support helped me stay motivated and carry this thesis through to completion.

Abstract

Context: Software development involves frequent handoffs between activities such as requirements understanding, design, implementation, debugging, and integration. During these handoffs, developers may become unable to make meaningful progress even when they understand the general task and are actively attempting to continue. Programmer's Block provides a construct for examining such situations, while prior research suggests that the causes of a block may originate outside the immediate act of writing code.

Objective: This thesis investigates Programmer's Block at development handoffs. It examines how developers experience and attempt to overcome blocking situations, how AI-based tools are perceived and used during these episodes, and how programming experience appears to influence developers' responses to being blocked and their use of AI-assisted support.

Method: The study followed an exploratory empirical software engineering design combining a descriptive survey of 85 respondents with nine semi-structured interviews. The survey provided contextual information about recent blocking experiences, difficult development handoffs, AI use, and perceived AI helpfulness. The interviews formed the main qualitative data source and were analysed using thematic analysis supported by STGT-informed procedures of open coding, constant comparison, and memoing. The analysis resulted in 62 consolidated master codes and five final themes.

Findings: In the survey, 52 of 85 respondents reported a recent blocking experience, and conceptual or pre-code handoffs accounted for 61.2% of the handoffs identified as most difficult. When blocked, 72.9% of respondents reported using AI tools often or sometimes, although their mean perceived helpfulness was 3.18 out of 5. The interviews showed that blocking frequently emerged when developers possessed partial understanding but could not transform it into a sufficiently safe next action. Fragmented or unusable technical and organisational context was an important contributing condition. Developers recovered by reconstructing context and making system behaviour or their own reasoning more visible through decomposition, experimentation, documentation, explanation, and social support. AI was most useful as supervised sense-making and implementation support rather than as an autonomous problem solver. Experience appeared to influence developers' recovery repertoires, confidence, help-seeking, and strategy selection more clearly than whether blocking occurred.

Conclusion: The findings suggest that Programmer's Block at development handoffs can be understood as a socio-technical phenomenon in which progress becomes difficult when the understanding available to a developer cannot be converted

into a sufficiently justified next action. AI can support this process by reducing some comprehension and implementation effort, but it also shifts reasoning toward context provision, verification, and supervision. Because no interview participant met the study's novice classification, experience-related findings should be interpreted as patterns within intermediate and experienced participants and retrospective earlier-career accounts rather than as a direct novice–experienced comparison.

Keywords: Programmer's Block, development handoffs, software engineering, generative AI, developer experience, qualitative analysis

Contents

List of Figures

List of Tables

List of Abbreviations

AI Artificial Intelligence

API Application Programming Interface

CORS Cross-Origin Resource Sharing

IDE Integrated Development Environment

LLM Large Language Model

PDF Portable Document Format

RQ Research Question

SDK Software Development Kit

SE Software Engineering

SQL Structured Query Language

STGT Socio-Technical Grounded Theory

URL Uniform Resource Locator

VPN Virtual Private Network

1. Introduction

Modern societies depend on software to operate services in healthcare, transport, finance, communication, public administration, and industrial infrastructure. The reliability and continued evolution of these services depend on software engineers being able to understand, modify, test, and integrate complex systems safely [28]. Difficulties that repeatedly interrupt this work are therefore not only local productivity concerns. When they accumulate, they can delay maintenance, slow the delivery of needed functionality, and increase the risk of changes being made without sufficient understanding.

Against this background, software development often becomes difficult not at a single isolated line of code, but at moments when developers have to move from one kind of work to another. A requirement has to become a concrete implementation decision, a design idea has to become executable code, a failing test has to become a diagnosis, and a code review comment has to become a safe change. These movements are part of ordinary software development, yet they can create uncertainty because the developer must reconstruct context, decide what matters, and transform partial understanding into a sufficiently safe next action.

A useful concept for examining such moments is Programmer's Block. Schantong et al. define and empirically investigate Programmer's Block as the inability to write or change code even though the task is known, the necessary programming skills and infrastructure are available, and the developer is willing to work [25]. Their interview study further shows that the cause of a block may originate outside the immediate act of code writing, including in requirements clarification, design, testing, or debugging. This finding creates a direct connection to the focus of this thesis: if blocks can originate where one development activity must be transformed into another, then development handoffs provide a relevant setting in which to investigate how such blocks emerge and how developers attempt to recover. In this thesis, a development handoff refers to movement from one software development activity to another. The term includes handoffs between people, but is used more broadly to include cognitive handoffs within an individual developer's own workflow.

Generative AI-based tools provide an additional way for developers to respond to such blocking situations. Studies of tools such as GitHub Copilot show that generated suggestions can provide starting points and possible directions, while also introducing difficulties related to understanding, verification, modification, and debugging [30, 3]. These tools may support developers by explaining code, proposing implementation options, generating examples, or assisting with debugging. However, their usefulness during a blocking episode is not self-evident because they may lack project-specific or organisational context, provide misleading output, or require

substantial critical evaluation from the developer.

A further motivation for this thesis is the role of programming experience. Programmer's Block should not be understood only as a novice difficulty. Experienced developers may also become blocked, but the way they recognise the situation, explain the cause, seek help, evaluate AI output, and recover may differ from how novice developers respond. Since programming experience is difficult to capture through years alone [27], this thesis treats experience as a multidimensional construct and uses it as a lens for comparing how developers experience and handle development handoffs.

This thesis therefore investigates Programmer's Block at development handoffs with particular attention to developer experiences, coping strategies, generative AI-assisted problem solving, and differences between novice and experienced developers. The study is guided by the following research questions:

- **RQ1:** How do software developers experience Programmer's Block at handoffs between activities in the software development process, and how do they attempt to overcome it?
- **RQ2:** How do developers perceive and use AI-based tools when experiencing Programmer's Block at handoffs between development activities?
- **RQ3:** How do novice and experienced developers differ in their experiences of Programmer's Block at development handoffs and in their perceptions of AI-assisted support?

The thesis follows an exploratory empirical software engineering approach. A short pre-survey was used to collect descriptive background information, support interview recruitment, and identify potentially relevant participant profiles. The main empirical data source consists of nine semi-structured interviews with developers. The survey results are treated as contextual and descriptive, while the main contribution of the thesis is based on the qualitative interview analysis.

The contribution of this thesis is an exploratory empirical account of Programmer's Block at development handoffs that connects the phenomenon with context reconstruction, recovery strategies, developer experience, and AI-assisted problem solving. The study provides evidence that blocking situations can emerge when developers possess partial understanding of a task or system but cannot transform that understanding into a sufficiently justified next action. It further examines how developers reconstruct context and externalise uncertainty when recovering, how AI-based tools support and redistribute this reasoning work, and how experience appears to shape recovery strategies more clearly than it prevents blocking. In doing so, the thesis develops a socio-technical interpretation of Programmer's Block that considers the developer together with the surrounding technical and organisational context.

The remainder of this thesis is structured as follows. Chapter 2 defines the central concepts used in the study, including Programmer's Block, development handoffs,

developer experience, and generative AI-based coding assistance. Chapter 3 reviews related work and establishes the research gap. Chapter 4 describes the research design, data collection, ethical considerations, and analysis approach. Chapter 5 presents the descriptive survey results, and Chapter 6 presents the interview findings. Chapter 7 discusses the findings in relation to prior research and considers implications for practice and future research. Chapter 8 discusses threats to validity, and Chapter 9 concludes the thesis.

2. Definitions and Theoretical Background

This chapter defines the concepts that are used throughout the thesis. It first introduces Programmer's Block and explains why the phenomenon is not limited to a lack of skill. It then defines development handoffs, developer experience, and generative AI-based coding assistance as they are used in this study.

2.1. Programmer's Block

Programmer's Block describes a situation in which a developer is unable to write or change code despite being objectively able to do so. The concept is inspired by writer's block, where a capable and motivated writer may struggle to start or continue writing. In software development, this means that being blocked is not equivalent to simply lacking programming knowledge, being unable to access the required tools, or being unwilling to work.

Schantong et al. define Programmer's Block through four conditions. The programming task is known or given, the required programming skills are sufficiently available, the programming infrastructure is available and running, and the programmer is willing and motivated to work on the task [25]. A developer may therefore understand the programming language and have access to the required environment while still being unable to determine how to continue with the implementation.

Prior work also distinguishes between different possible origins of blocks. Conceptual blocks can occur when a developer's understanding of the task, system, or intended solution is incomplete, inconsistent, or based on assumptions that do not fit the current situation. Performative blocks are associated with factors such as pressure, fear of producing an inadequate result, or the expectation that an initial attempt must already meet the standard of a finished solution [25]. These concepts help characterise different mechanisms through which a capable developer may become unable to progress. In this thesis, they provide theoretical background for interpreting participants' accounts, while the empirical focus remains on where blocks emerge during development handoffs and how developers respond to them.

2.2. Development Handoffs

Software engineering is commonly described as involving activities such as requirements engineering, design, implementation, testing, integration, maintenance, and

evolution [28]. These activities are often represented as parts of a software development life cycle. Such representations are useful for describing different kinds of development work, but they should not be interpreted as implying that developers always progress through them in a fixed sequence. In practice, developers move repeatedly between activities, revisit earlier decisions, and may perform several activities in parallel. This thesis uses the term *development handoff* for movements in which work or understanding must be transferred or transformed from one development activity to another.

A development handoff may involve work being passed from one person to another, but it may also occur within one developer's own workflow. Examples include moving from understanding requirements to deciding what to build, from deciding what to build to designing a solution, from designing a solution to starting implementation, from writing code to testing or debugging, from debugging to deciding on a fix, and from code review to integration.

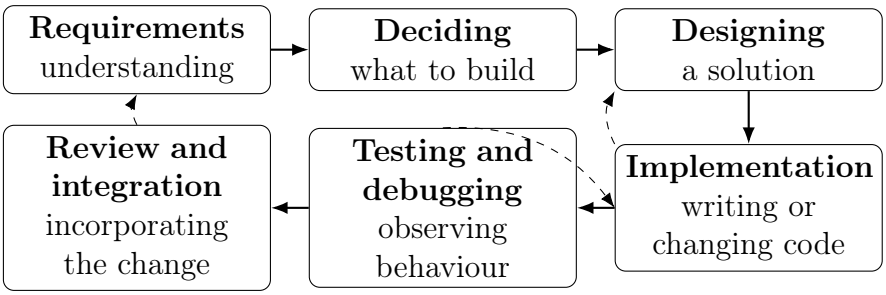

Figure 2.1.: Illustrative development handoffs between common software development activities. The solid arrows show example movements between activities, while the dashed arrows illustrate that development work may return to earlier activities. The figure is conceptual and does not represent a fixed or prescriptive software development life cycle.

Figure 2.1 illustrates the meaning of a development handoff used in this thesis. The arrows represent points at which information, decisions, assumptions, or technical understanding must be carried from one activity into another; they do not imply that software development follows a strictly sequential process.

The connection between development handoffs and Programmer's Block is grounded in the findings of Schantong et al. Their study found that, although a block becomes visible when a developer cannot write or modify code, its cause may originate in other development activities, including requirements clarification, design, testing, and debugging [25]. This observation motivates examining the boundaries between such activities more closely. For example, a requirement may need to become an implementation direction, observed test behaviour may need to become a diagnosis, or a design concept may need to become executable code.

Development handoffs are therefore relevant because they are points at which the developer must transform one form of partial understanding into another. When

the information required for this transformation is incomplete, contradictory, inaccessible, or difficult to evaluate, the developer may understand the general task and possess the necessary programming skills while still being unable to identify a sufficiently safe next action. Under the conditions defined above, such a failure to continue may constitute Programmer's Block rather than an ordinary lack of skill.

2.3. Developer Experience

Programming experience is relevant because Programmer's Block is defined as occurring despite the developer possessing sufficient ability to work on the task. The original study by Schantong et al. was conducted with experienced programmers and nevertheless found that nearly all participants could describe blocking experiences [25]. Experience therefore does not necessarily prevent Programmer's Block. It may instead influence the kinds of situations in which developers become blocked, how quickly they recognise an unproductive approach, which coping strategies they use, when they seek assistance, and how confidently they evaluate possible solutions.

Experience is also context-dependent. A developer may be highly experienced in programming generally but comparatively inexperienced with a particular codebase, framework, domain, team, or development environment. Prior research shows that programming experience cannot be represented adequately by years alone because duration, self-assessed expertise, education, and breadth of practical exposure capture different aspects of experience [27]. Research on software-development expertise similarly describes expertise as dependent on knowledge, practice, and the context in which a task is performed [2].

This distinction is especially relevant to AI-assisted support. Using an AI-generated suggestion requires more than obtaining an answer: the developer must determine whether it fits the requirement, project architecture, business context, and quality constraints. Experience may therefore shape both what developers ask AI tools to do and whether they can recognise incomplete, misleading, or unsuitable output. These considerations motivate the comparison in RQ3 between developers with different experience profiles.

This thesis consequently treats experience as a multidimensional construct. Years of programming experience are considered together with self-assessed expertise and breadth of exposure to projects, domains, technologies, and system contexts. The comparison is used as an interpretive lens rather than as an assumption that one group experiences fewer blocks than another.

2.4. Generative AI-Based Coding Assistance

This thesis uses *generative AI-based coding assistance* as an umbrella term for large language model-based tools that developers can use to generate, transform, explain, or reason about software-related information. Examples include conversational sys-

tems such as ChatGPT and IDE-integrated assistants such as GitHub Copilot. Depending on the tool and the interaction, developers may use such systems to generate code, explain unfamiliar logic, summarise documentation, interpret errors, propose alternative solutions, create tests, or support debugging.

Generative AI can support different forms of programming work. A developer who already knows what needs to be implemented may use AI to reduce repetitive effort, while a developer who is uncertain about the next step may use generated suggestions to explore possible directions. Barke et al. distinguish these patterns as acceleration and exploration modes [3]. At the same time, generated output does not remove the need for developer judgement. Human developers may still need to inspect, interpret, modify, reject, or later reconsider AI-generated suggestions [18]. AI assistance can therefore change the distribution of problem-solving work without necessarily removing the reasoning required to complete the task.

The present thesis does not evaluate generative AI as a code-generation benchmark or compare the technical performance of individual models. Instead, AI-assisted problem solving is examined as one possible coping resource during blocking episodes. The focus is on what developers ask AI tools to do when they are unable to progress, where they perceive the tools as helpful, which limitations prevent AI from resolving a block, and how developer experience influences the supervision and evaluation of AI-generated output. This definition provides the conceptual basis for RQ2 and for the AI-related comparison considered in RQ3.

3. Related Work

This chapter positions the thesis within research on Programmer's Block, difficult transitions in software development work, programming experience, and generative AI-based coding assistance. These areas have mostly been studied separately. Research on Programmer's Block establishes that capable developers can become unable to progress. Studies of developers' everyday work explain how incomplete context, unresolved information needs, interruptions, unfamiliar systems, and debugging uncertainty can make progress difficult. Research on expertise shows that experience is multidimensional and context-dependent. Finally, studies of AI coding assistants describe both their value as problem-solving support and the additional effort required to verify their output. Bringing these areas together provides the basis for investigating Programmer's Block specifically at development handoffs.

3.1. Programmer's Block and Related Experiences of Being Stuck

The closest prior work is the study by Schantong et al. on Programmer's Block [25]. Based on semi-structured interviews with experienced programmers, the authors describe Programmer's Block as an inability to write or change code despite the task being known, sufficient programming skills being available, the required infrastructure being operational, and the programmer being willing to work. This distinction is important because it separates a block from ordinary learning difficulties, missing basic skills, unavailable tools, or a temporary lack of motivation.

The study also shows that a block does not necessarily originate where it becomes visible. Although the immediate symptom may be an inability to write or modify code, its cause may lie in requirements clarification, design, testing, debugging, or another part of the development process. Schantong et al. therefore argue that Programmer's Block should be studied across the wider development process rather than only during code writing. This observation directly motivates the focus of the present thesis on development handoffs, where developers must transform one kind of understanding into another.

Other research has investigated being stuck without using the specific construct of Programmer's Block. Müller and Fritz examined developers' emotions and progress during programming tasks and found that being stuck was associated with negative affect, while flow was associated with more positive emotional states [19]. Their findings show that lack of progress is not only visible in reduced output; it is also

experienced emotionally through frustration and uncertainty. More recent mixed-methods research on developers' "bad days" similarly identifies friction from tools, processes, and team interactions as factors that negatively affect productivity and well-being [20]. This broader developer-experience literature supports the practical relevance of blocked work, but it does not determine which difficult episodes satisfy the more specific conditions of Programmer's Block.

The present thesis builds on Schantong et al. by retaining Programmer's Block as the central construct while narrowing the setting to development handoffs. It also extends the existing discussion by examining coping strategies, the role of AI-based tools, and differences associated with developer experience.

3.2. Context Reconstruction and Sense-Making at Development Handoffs

A development handoff requires a developer to move from one activity or representation of a problem to another. Examples include moving from requirements to design, design to implementation, observed failure to diagnosis, diagnosis to a fix, or review feedback to integration. Prior studies do not usually describe these movements as Programmer's Block, but they identify several mechanisms that can make such transitions difficult.

One mechanism is the need to maintain an accurate mental model of the system. LaToza et al. found that developers invest substantial effort in recovering implicit knowledge about source code, design decisions, dependencies, and system behaviour [14]. Much of this knowledge remained in individual memory rather than in documentation. Consequently, developers often had to explore code or interrupt colleagues to reconstruct the information required for their work. At a handoff, this creates a risk that a developer understands the general objective but lacks the local system knowledge needed to select a safe next action.

Ko et al. studied the information needs of professional developers and identified recurring questions about program behaviour, design rationale, task history, relevant artefacts, and the people who possess particular knowledge [13]. Some information searches were deferred because the required colleague or source was unavailable. These findings are closely related to development handoffs: moving forward may depend on knowing why existing code was written in a particular way, what a component is expected to do, or which assumption explains the current program state. When such information is unavailable, the transition from understanding a problem to acting on it may remain incomplete.

The same issue is visible when developers enter unfamiliar projects. Dagenais et al. describe onboarding as movement into a new project landscape that includes code, documentation, tools, processes, conventions, and people [8]. Their grounded theory study shows that newcomers must learn these elements together and validate whether they are making progress. Begel and Simon similarly found that newly

hired developers may become novices again when they encounter industrial codebases, tools, organisational practices, and coordination demands that differ from their previous experience [4]. These studies suggest that competence in programming does not automatically provide the contextual knowledge required in a new language, framework, team, or codebase.

A second mechanism is loss of continuity. Parnin and Rugaber analysed interrupted programming sessions and found that resuming work usually required navigation and context-seeking before editing restarted [21]. Returning to a task is therefore not an immediate continuation; it is a process of reconstructing goals, locations, and assumptions. Cruz et al. found that work fragmentation, in which activities and interruptions are interleaved, was correlated with lower observed productivity in developer interaction data [7]. Although a planned development handoff is not identical to an interruption, both require reorientation and may weaken access to the context that supported the previous activity.

A third mechanism concerns the learnability of technical artefacts. Robillard and DeLine found that developers learning unfamiliar APIs were hindered by fragmented documentation, missing examples, difficulty locating relevant material, and problems matching examples to the current scenario [23]. Kelleher and Brachman later described API learning as a sense-making process in which programmers search for information, form an interpretation, test it, and revise their understanding [12]. Together, these studies show that documentation availability alone is not sufficient. Developers must connect available information to the local task and determine whether their interpretation explains the observed behaviour.

Debugging provides a particularly clear example of a difficult handoff. A developer must transform symptoms, logs, test failures, or unexpected output into a plausible explanation and then into a safe fix. Weber et al. describe debugging as a sequence of goal-oriented episodes, including exploration, search, reasoning, and periods of aimlessness [31]. The developer may therefore remain active while still being unable to progress because the available evidence has not yet supported a sufficiently reliable diagnosis. This resembles the central pattern examined in the present thesis: partial understanding exists, but it cannot yet be converted into a confident next action.

Across these studies, development difficulty is repeatedly linked to incomplete mental models, unresolved information needs, lost context, unfamiliar project landscapes, fragmented work, and iterative sense-making. However, this literature generally studies information work, onboarding, API learning, interruption, or debugging as separate phenomena. It does not examine how developers themselves experience these moments as Programmer's Block across different development handoffs.

3.3. Programming Experience and Responses to Difficulties

RQ3 requires a careful treatment of programming experience. Years of experience provide useful background information, but they do not represent expertise on their own. Siegmund et al. compared different ways of measuring programming experience and showed that years, education, self-assessment, and practical exposure capture different aspects of experience [27]. This motivates the multidimensional operationalisation used in this thesis, which combines duration, self-rated expertise, and breadth of programming exposure.

Baltes and Diehl further argue that software-development expertise is context-dependent and shaped by knowledge, practice, experience, and the surrounding work environment [2]. An experienced developer may therefore be highly capable in one domain while becoming locally inexperienced in an unfamiliar framework, architecture, organisation, or codebase. This is consistent with the professional newcomers studied by Begel and Simon, who already possessed programming education but still faced novice-like difficulties in their industrial setting [4].

Self-assessment also needs to be interpreted cautiously. Daun et al. analysed four experiments involving model-comprehension tasks and found that students' overall self-rated experience did not reliably predict their performance, although confidence in a specific answer provided more useful information [9]. Their study concerns students and conceptual models rather than professional programming, so the result cannot be transferred directly to this thesis. It nevertheless supports the decision not to treat one self-rating as an objective measure of ability.

Behavioural comparisons indicate that experience can change the type of difficulty and the response to it. Chuang and Chang found that novice and competent programmers differed in error patterns, correction time, and solution correctness during automated programming tasks [6]. Competent programmers spent less effort on low-level coding errors, while novices devoted more attention to syntax-level correction. Such findings do not suggest that experienced developers stop becoming blocked. Instead, they suggest that blocks may occur at different levels. Less experienced developers may be more affected by syntax, tool use, confidence, or uncertainty about when to seek help, whereas experienced developers may encounter architectural, integration, domain, or system-history problems.

The literature therefore supports examining experience as a factor that shapes the recognition, interpretation, and management of blocks rather than assuming that experience prevents them. The present study uses this perspective to compare how developers explain their blocking episodes, which coping strategies they select, and how they evaluate AI-assisted support.

3.4. Generative AI-Based Coding Assistance

Research on generative AI in software engineering includes studies of code generation, testing, repair, and productivity. For this thesis, the most relevant work is human-centred research on how developers use coding assistants when they are uncertain, how they evaluate suggestions, and what additional work is created by AI-generated output.

Vaithilingam et al. evaluated the usability of GitHub Copilot in a within-subjects study [30]. Participants often preferred having the tool available and valued generated code as a possible starting point, but they also experienced difficulty understanding, editing, and debugging its suggestions. The study is relevant to Programmer's Block because it distinguishes perceived helpfulness from successful problem resolution. A suggestion may help a developer begin exploring without resolving the underlying uncertainty.

Barke et al. provide a particularly useful account through their grounded theory of programmer interaction with Copilot [3]. They distinguish between *acceleration mode*, in which the developer already knows what to do and uses AI to complete it faster, and *exploration mode*, in which the developer is uncertain and uses suggestions to investigate possible directions. Blocking episodes in this thesis are more closely related to exploration. In such situations, the contribution of AI may lie in producing alternatives, examples, explanations, or intermediate representations rather than directly generating a final implementation.

Large-scale survey research shows both the appeal and the limits of this support. Liang et al. found that developers valued AI programming assistants for reducing keystrokes, completing tasks faster, and recalling syntax [16]. Developers were less positive about using them for deeper solution development. Reported difficulties included suggestions that did not satisfy functional or non-functional requirements, limited control over the generated result, and the effort required to modify or debug it. This suggests that AI assistance is easier to use when uncertainty is bounded and the developer can specify and verify the local task. It is less reliable when the missing element is project rationale, business knowledge, or architectural direction.

Verification is therefore a central part of AI-assisted programming. Mozannar et al. developed a taxonomy of programmer activities around AI suggestions and showed that developers spend substantial effort inspecting, evaluating, accepting, rejecting, and later reconsidering generated code [18]. Accepting a suggestion does not necessarily mean that the developer fully trusts it or that the task has been completed. AI can shift the problem from producing a candidate solution to judging whether that solution is correct, maintainable, secure, and suitable for the surrounding system.

More directly related to blocking situations, Anderson et al. examined how developers became stuck and unstuck while making a non-trivial change to an unfamiliar codebase with and without ChatGPT [1]. Their exploratory laboratory study involved ten advanced student developers, five with access to ChatGPT and five without. Nine of the ten participants experienced at least one stuck moment. The

authors identified several forms of stuckness, including situations in which developers had insufficient or flawed understanding of the system. AI played different roles across these episodes: it sometimes helped developers regain progress, sometimes failed to resolve the difficulty, and in some cases reinforced an incorrect understanding or contributed to the developer becoming more stuck.

This study is particularly relevant to the present thesis because it examines AI at the level of problem-solving processes rather than only productivity or code quality. It also shows that successful AI assistance does not necessarily require the developer to perform all of the implementation reasoning independently. In some cases, participants delegated parts of the work to AI and continued making progress despite gaps in their own understanding. At the same time, the study warns that developers may outsource parts of their cognition to AI and therefore fail to construct an adequate understanding of the system themselves. These findings strengthen the need to examine AI not only as a source of generated code, but as a tool that can influence how developers enter, remain in, and recover from stuck states.

Recent research also raises questions about how generative AI redistributes cognitive effort. Lee et al. found that greater confidence in generative AI among knowledge workers was associated with lower self-reported critical-thinking effort, while the remaining cognitive work increasingly involved verification, integration, and responsibility for the final task [15]. In software development specifically, Tang et al. showed that validating and repairing LLM-generated code still required substantial inspection and search behaviour and could introduce additional cognitive workload [29]. Studies of AI-assisted programming therefore suggest that reducing the effort required to generate a possible solution does not necessarily remove the cognitive work of programming. Instead, some of that work may shift toward understanding, verification, supervision, and deciding whether generated changes can be trusted. A recent study of AI-assisted brownfield programming similarly found improved task effectiveness and efficiency while raising questions about whether successful use of generated solutions is accompanied by an equally strong understanding of why those solutions work [26].

Research in enterprise settings adds further context. Weisz et al. found that code understanding was an important use case for an internally deployed AI code assistant and that perceived benefits varied across users [32]. Their study also highlights questions of responsibility and ownership for generated code. These issues are relevant during blocked work because developers remain accountable for decisions even when AI contributes explanations or implementation suggestions. Organisational policies, confidentiality restrictions, and limited access to project-specific context may also determine whether AI can be used at all.

Evidence on productivity is similarly conditional. Peng et al. reported increased task-completion speed in a controlled study of GitHub Copilot [22]. However, faster completion under bounded experimental conditions does not establish that AI can resolve context-heavy blocks in real projects. The human-centred studies reviewed above show that usefulness depends on the developer's ability to provide context, interpret the response, and validate the result.

Overall, the literature presents AI as a support resource rather than an autonomous replacement for developer reasoning. AI may help developers recall syntax, understand unfamiliar code, generate alternatives, prepare tests, or externalise a tentative plan. At the same time, it may lack local system knowledge, organisational context, current documentation, or access to private artefacts. Its output may also introduce additional verification and repair work. These tensions make it important to study not only whether developers use AI when blocked, but how they use it, what they expect from it, and how experience shapes their trust and supervision.

3.5. Research Gap

The reviewed literature provides several complementary perspectives on difficult software-development work. Research on Programmer's Block establishes that capable developers can become unable to write or change code even when the task, skills, infrastructure, and motivation required for programming are present, and that the causes of such blocks may originate outside code writing itself [25]. Research on information needs, onboarding, task resumption, fragmentation, API learning, and debugging explains why developers may need to reconstruct context or revise their understanding before they can continue [13, 8, 21, 7, 23, 12, 31]. Research on programming experience further shows that expertise is multidimensional and context-dependent [27, 2]. Finally, studies of generative AI show that coding assistants can support exploration, comprehension, and implementation while also creating additional work related to interpretation, verification, and supervision [30, 3, 16, 18].

Recent work has begun to connect generative AI directly with developers' experiences of being stuck. Anderson et al., for example, examined how developers became stuck and unstuck while changing an unfamiliar codebase and showed that AI could help restore progress, fail to resolve the difficulty, or contribute to the stuckness itself [1]. This provides an important process-level account of AI-assisted problem solving. However, the study was conducted as a controlled laboratory task with ten advanced student developers working on the same unfamiliar codebase. It therefore leaves open how similar processes appear in naturally occurring blocking episodes across professional software-development work.

Several questions consequently remain insufficiently understood. First, existing Programmer's Block research does not examine development handoffs in depth or explain how blocks emerge when understanding from one development activity must be transformed into action in another. Second, existing studies of stuckness and AI largely examine bounded or experimentally defined programming tasks, while less is known about AI use during naturally occurring blocks involving project history, organisational knowledge, inaccessible environments, cross-role dependencies, and other socio-technical conditions. Third, although experience research shows that expertise is context-dependent, there is limited evidence about how experience shapes developers' emotional responses, help-seeking, recovery strategies, and evaluation of

AI support during blocking episodes.

This thesis addresses these gaps through an exploratory empirical study of Programmer's Block at development handoffs. It combines descriptive survey evidence with detailed accounts of naturally occurring blocking experiences and investigates how developers reconstruct context, attempt to recover, use or avoid AI-based support, and interpret the role of programming experience in these situations.

4. Methodology

This chapter describes the empirical design of the thesis. The study followed an exploratory design combining a short pre-survey with semi-structured interviews. The survey was conducted first and provided descriptive information about blocking experiences at development handoffs, difficult handoffs, AI tool use, and perceived AI helpfulness. It also supported the recruitment of participants for the follow-up interviews.

The interviews formed the main qualitative data source. They provided detailed accounts of how developers experienced Programmer's Block, how they interpreted its causes, which strategies they used to move forward, how generative AI tools contributed to these strategies, and how participants perceived programming experience to influence their responses to blocking situations. Figure 4.1 provides an overview of the research process.

To support transparency and traceability, the study materials and analytical artefacts are reproduced in the appendices. The survey instrument is provided in Appendix A, the semi-structured interview guide in Appendix B, and the consolidated master codebook in Appendix C.

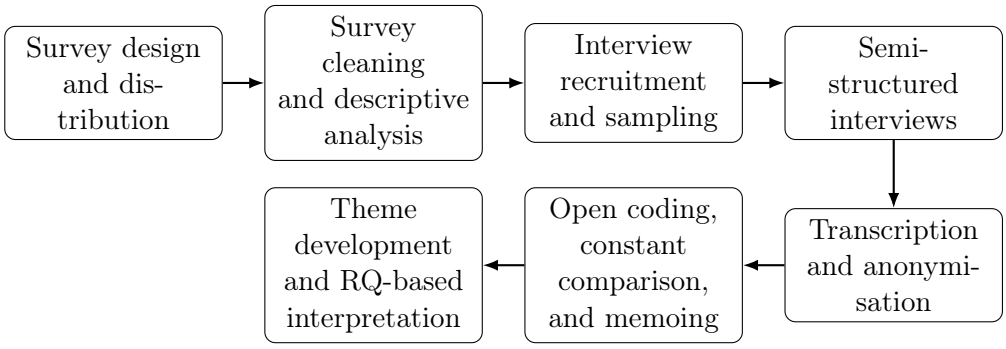

Figure 4.1.: Overview of the empirical research process.

4.1. Survey and Recruitment Instrument

The survey was designed as a short four- to five-minute questionnaire containing 11 questions. It introduced the topic of Programmer's Block at development handoffs and provided working definitions of the two central concepts. In the survey, being blocked was defined as feeling stuck, uncertain, or unable to make meaningful

progress on a programming task despite actively trying. A transition was defined as moving from one development activity to another, for example from understanding requirements to designing a solution or from writing code to debugging.

The survey definition of being blocked was intentionally broader than the four-condition definition of Programmer's Block introduced in Section 2.1. A short survey could not establish for every reported episode whether the participant possessed all required skills, whether the complete programming infrastructure was available, or whether all other conditions of Programmer's Block were satisfied. Survey responses are therefore treated as reports of *blocking experiences relevant to Programmer's Block* rather than as independently verified cases of Programmer's Block. The interviews provided greater contextual detail for examining how individual blocking episodes related to the construct.

The survey contained four parts: participant background, experiences of being blocked, AI tool use, and willingness to participate in a follow-up interview. Each part served a separate purpose within the study.

The background section collected years of programming experience, self-assessed programming expertise, and breadth of programming experience. These questions were needed because experience was treated as a multidimensional construct rather than as a variable based only on years. They provided the information required to describe the survey sample, classify interview participants, and support the qualitative comparison in RQ3.

The blocking section asked whether respondents had experienced a block at or due to a development handoff during the previous month, which handoff they found most difficult, and how long such blocks usually lasted. These questions provided descriptive context for RQ1 and helped identify respondents who could discuss a concrete and recent blocking episode during an interview.

The AI section asked how frequently respondents used AI-based tools when blocked and how helpful they perceived those tools to be. These questions provided descriptive context for RQ2. They also made it possible to consider variation between frequent, occasional, rare, and non-users of AI when constructing the interview sample.

The final section asked whether respondents were willing to participate in a follow-up interview. Respondents could select "Yes", "Maybe", or "No". Respondents selecting "Maybe" could optionally provide a short explanation for their answer. Respondents selecting "Yes" or "Maybe" could optionally provide an email address for follow-up contact. The contact information was collected only for interview recruitment and scheduling. Expressions of interview interest were counted independently of whether usable contact information was provided. Respondents who expressed interest and supplied usable contact details could be approached for interview recruitment. The interview-participation question and the email-address field were optional. Not answering these questions did not affect whether a response was eligible for the descriptive survey analysis.

The background, blocking, and AI-use responses were used to assess the relevance of potential participants to the research questions and to consider variation across

experience profiles, recent blocking experiences, difficult handoffs, and AI-use patterns. Recruitment was also influenced by respondents' continued willingness and availability. Nine respondents ultimately completed a follow-up interview.

Email addresses were used only for interview recruitment and scheduling and were stored separately from the analytical survey data. They were not included in the descriptive survey analysis.

The survey was not intended to establish the statistical prevalence of Programmer's Block. The sample was exploratory and was not designed to represent the wider population of software developers. The findings were therefore interpreted as descriptive patterns within the collected sample and as contextual support for the qualitative interview study. The complete survey instrument, including the participant information, data-protection text, response options, and interview-recruitment questions, is reproduced in Appendix A.

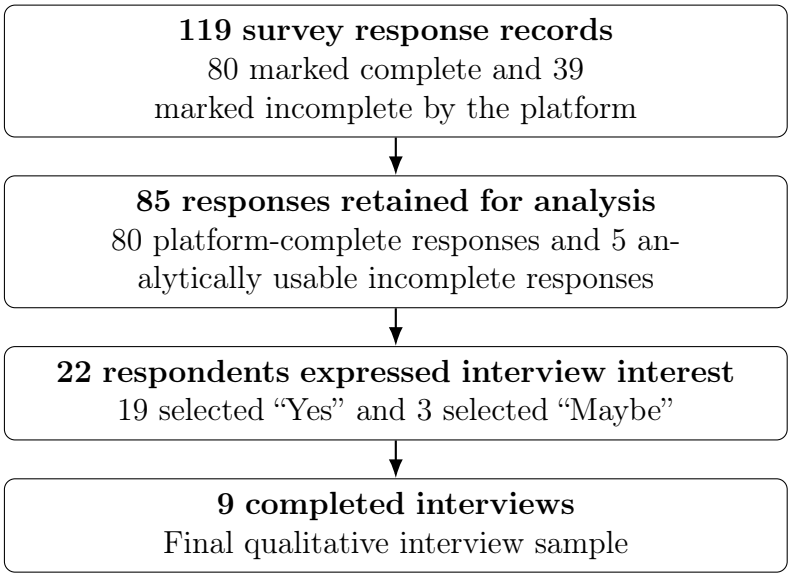

Figure 4.2.: Survey cleaning and interview recruitment outcomes. Of the 39 responses marked incomplete by the platform, five contained all analytical answers and were retained; the remaining 34 were excluded because they contained insufficient information for the planned analysis.

Recruitment outcome. Recruiting developers for both stages of the study required repeated distribution through online developer communities and the researcher's professional network. The survey platform recorded 119 response records. Eighty responses were marked as complete, while 39 were marked as incomplete. Five of the incomplete responses contained all variables required for the descriptive analysis but did not complete the optional interview-recruitment section. These responses were retained, resulting in a final analytical dataset of 85 responses. The remaining 34 records were excluded because they contained insufficient information for the planned analysis.

Twenty-two respondents selected "Yes" or "Maybe" when asked whether they would be willing to participate in a follow-up interview: 19 selected "Yes" and 3 selected "Maybe". However, only nine ultimately completed an interview. Continued participation depended on whether respondents provided usable contact information, replied to follow-up communication, remained willing to discuss their experiences, and were available for scheduling.

Access to a professional network did not automatically lead to participation. Within the researcher's own organisation, two employees volunteered for an interview, and no members of the researcher's immediate team participated. Because reasons for non-participation were not collected, no conclusions can be drawn about why other colleagues did not volunteer. Recruitment and sustained practitioner engagement are recognised challenges in empirical software engineering, particularly in industrial contexts [24].

4.2. Semi-Structured Interviews

Semi-structured interviews were selected because they provide access to detailed participant experiences while allowing the researcher to cover a consistent set of topics across interviews [11]. Their flexible structure also permits follow-up questions when a participant introduces an unexpected but relevant aspect of the phenomenon.

The interviews focused on concrete and preferably recent situations in which participants had felt stuck or unable to make progress while working on a programming task. Beginning with a specific episode rather than asking only for general opinions about Programmer's Block supported the collection of experience-based accounts and reduced the likelihood of receiving only abstract or speculative answers.

The interview guide was organised around five connected analytical areas. The first group collected participant context, including the type of software development work performed, years of programming experience, self-assessed expertise, and breadth of programming experience. These questions supported the experience classification described in Section 4.4 and the qualitative comparison in RQ3.

The second group asked participants to describe a recent blocking episode and identify the point in the development process at which they became stuck. These questions supported RQ1 by locating the episode at a possible development handoff. Participants were not required to describe their experience using the term Programmer's Block. Instead, the questions elicited the circumstances and experiences needed to determine how the episode related to the study's definition.

The third group explored the participant's interpretation of the cause of the block and the strategies used to move forward. These questions also supported RQ1 by examining both how the block emerged and how the developer responded to it. Follow-up questions were used where necessary to clarify what had been unclear, unavailable, contradictory, or missing at the time of the episode.

The fourth group focused on generative AI. Participants were asked whether they had used an AI-based tool during the episode, how they had used it, and in what

way it had or had not been helpful. Participants who had not used AI were asked why they had chosen not to use it. These questions supported RQ2 by capturing both AI use and relevant non-use cases.

The fifth group focused on reflections on experience. Participants were asked to compare the described episode with how they might have responded earlier in their career and how they expected greater experience to affect similar situations in the future. These retrospective and prospective comparisons supported RQ3 by eliciting participants' own interpretations of how experience influenced emotional responses, problem diagnosis, help-seeking, strategy selection, and AI use.

All nine interviews were conducted remotely and in English over a period of approximately four to six weeks. The interviews lasted approximately 20–25 minutes. The same core interview guide was used across all nine interviews, while follow-up questions were adapted to the situation described by each participant. The full interview guide is included in Appendix B.

4.3. Data Analysis

The survey data were analysed descriptively. Counts and percentages were used to summarise the respondent profile, recent blocking experiences, difficult development handoffs, block duration, AI tool use, and perceived AI helpfulness. Mean and median values were used where appropriate for the five-point AI-helpfulness scale. The survey results were used to contextualise the qualitative interview analysis rather than to support statistical generalisation.

The interview data were analysed using thematic analysis supported by STGT-informed procedures of open coding, constant comparison, and memoing [5, 11]. STGT was used at the qualitative data-analysis level. The study did not claim to apply the complete STGT method or to develop a formal grounded theory.

The analysis began with familiarisation. Each anonymised transcript was read in full before and during coding so that individual statements could be interpreted in relation to the wider account. Relevant extracts were then coded openly rather than being assigned to a predefined set of themes.

Open coding was used to assign concise labels to statements concerning blocking experiences, development handoffs, perceived causes, emotional responses, coping strategies, social and organisational conditions, AI-related practices, and reflections on experience. During the initial stage, codes remained close to the participant's account so that potentially relevant differences were not removed through premature abstraction.

Constant comparison was used throughout the analysis. Codes were compared with other codes from the same interview and with codes developed from earlier interviews. Similar codes were merged when they represented the same underlying idea. Codes were separated or renamed when one label covered meaningfully different experiences. Comparisons were also used to identify negative, contrasting, and limiting cases rather than retaining only evidence that supported an emerging

interpretation.

Memoing was used to record developing interpretations, relationships between codes, questions raised by the data, links to the research questions, and cases that challenged an emerging pattern. These memos supported the movement from descriptive codes towards broader analytical categories and themes.

The qualitative analysis proceeded in several stages. First, each transcript was coded individually. Second, codes from all nine interviews were compared and consolidated into a master codebook. The consolidated codebook included code definitions, inclusion and exclusion boundaries, example extracts, and links to the research questions.

The 62 master codes were organised into seven code families: participant and experience context; block location or development handoff; causes and conditions of the block; lived experience of being blocked; coping and recovery strategies; AI use and perception; and experience-related differences. These families were used to organise related analytical material and support cross-interview comparison. They were not treated as themes themselves.

Third, the coded extracts were organised into a cross-interview evidence matrix. The matrix was used to compare the participants' main blocking situations, development handoffs, perceived causes, coping strategies, AI use, AI limitations, and experience-related observations. Related codes were then grouped into candidate categories and developed into candidate themes.

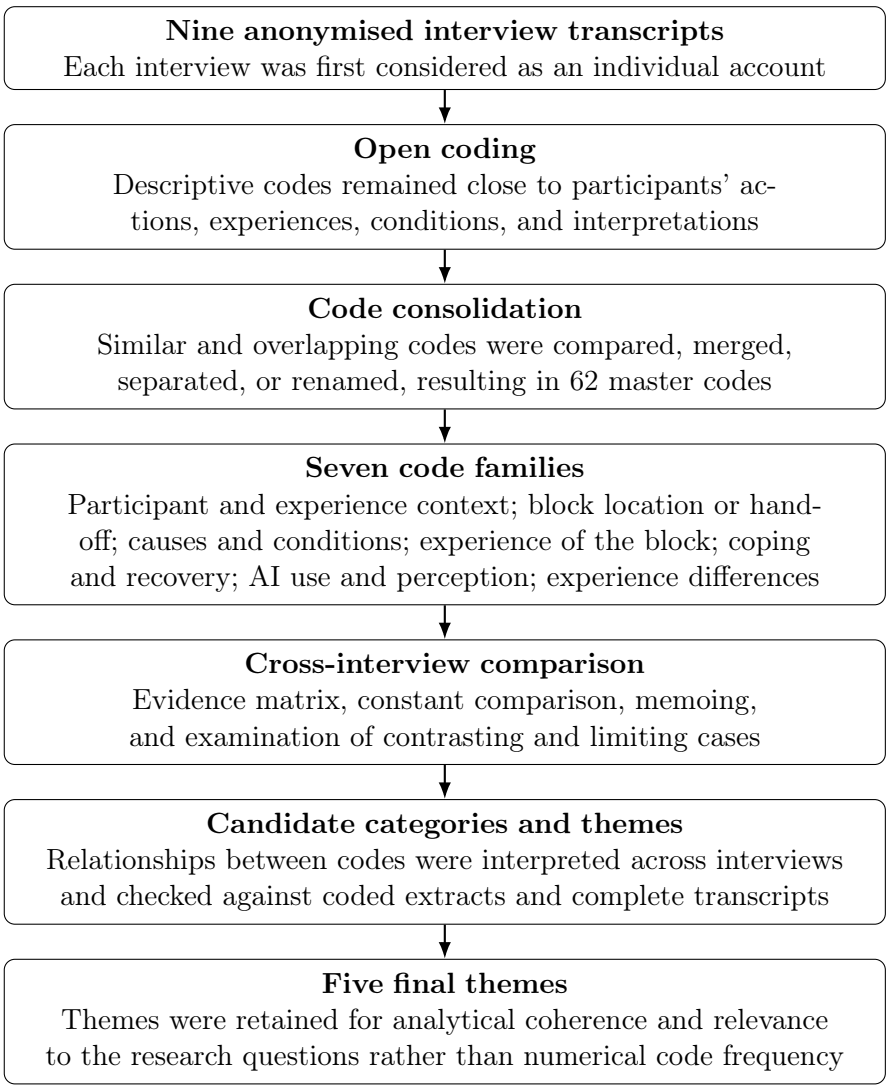

Figure 4.3.: Progression of the qualitative analysis from nine interview transcripts to five final themes. Open codes were consolidated into 62 master codes organised into seven code families and compared across interviews using constant comparison, memoing, and an evidence matrix. Themes were selected for analytical meaning rather than numerical frequency.

Figure 4.3 summarises this progression. The complete master codebook, including code definitions, coding rules, research question links, and source examples, is provided in Appendix C.

The candidate themes were reviewed repeatedly against the coded extracts and the complete transcripts. Theme boundaries and names were revised where proposed themes overlapped, lacked sufficient supporting evidence, or did not provide a coherent answer to the research questions. The final themes were selected because each captured a recurring and analytically meaningful pattern across the dataset rather than because it appeared with the highest numerical frequency.

Quotations selected for the findings chapter were checked against the transcripts. Filler words, repeated fragments, and clear transcription errors were removed only where this did not alter the participant's meaning. Clarifying additions are shown in square brackets. Paraphrased coded extracts were not presented as direct quotations.

For RQ1, the analysis examined where blocking episodes emerged, how participants interpreted their causes, and which strategies they used to move forward. For RQ2, it examined how participants used or avoided AI-based tools and how they evaluated their usefulness and limitations. For RQ3, patterns were compared across the intermediate and experienced participant profiles and were supplemented by participants' retrospective reflections on how similar blocking situations affected them earlier in their careers. Because no interview participant met the novice classification, the analysis does not provide a direct comparison with developers who are currently novices. This comparison was qualitative rather than statistical and focused on differences in confidence, emotional response, help-seeking, diagnosis, strategy selection, and evaluation of AI output.

4.4. Operationalising Developer Experience

Programming experience was treated as a multidimensional construct rather than as a variable based only on years of experience. This decision was informed by prior work on measuring programming experience, which shows that software engineering studies use indicators such as years of programming, education, self-estimation, questionnaires, and practical project exposure. Siegmund et al. recommend combining indicators from different categories and reporting clearly how experience groups are formed [27].

In this thesis, programming experience was operationalised using three indicators collected through the survey and confirmed during the interviews:

- years of programming experience, including study and personal projects;
- self-assessed expertise compared with developers who had similar backgrounds; and
- breadth of programming experience across projects, domains, technologies, and system contexts.

The specific thresholds used to form the three experience profiles were operational decisions defined for this study. They should not be interpreted as validated boundaries between novice, intermediate, and experienced programmers, nor as thresholds proposed by the cited prior work. Their purpose was to combine the three indicators consistently while avoiding classification based on any single measure alone.

Participants were categorised as novice when at least two of the three indicators reflected a lower experience level. The lower thresholds were two years or less of

programming experience, self-assessment as much less or slightly less experienced than comparable developers, and breadth described as very narrow or somewhat narrow.

Participants were categorised as experienced when at least two indicators reflected a higher experience level. The higher thresholds were six or more years of programming experience, self-assessment as slightly more or much more experienced than comparable developers, and breadth described as broad or very broad.

Participants who did not meet at least two lower or two higher thresholds were categorised as intermediate. The two-of-three rule reduced dependence on any single indicator. For example, a participant with several years of programming experience was not automatically treated as experienced when their self-assessment and breadth of exposure suggested a more intermediate profile.

This multidimensional classification supported the construction and description of the interview sample and the qualitative comparison for RQ3. In parts of the survey results, respondents were grouped descriptively using years of programming experience only because the small subgroup sizes did not support detailed comparisons across every combination of the three indicators. The multidimensional classification was used more directly in the interview analysis, where the complete participant profile could be considered.

The classification was not intended to provide a statistically validated measure of programming expertise or to imply that experience levels represented fixed personal attributes. Experience was treated as context-dependent: a generally experienced developer could still be comparatively inexperienced in a particular language, framework, codebase, domain, team, or organisational environment.

4.5. Ethical Considerations

Participation in both stages of the study was voluntary. For the survey, participants were informed about the purpose of the study, the type of data being collected, the intended scientific use of the data, and their right to stop participating by closing the survey. The participant information stated that the legal basis for processing the survey data was voluntary consent under Article 6(1)(a) of the EU General Data Protection Regulation (GDPR).

The analytical survey responses were collected without collecting directly identifying information about participants. Optional contact information could be provided separately by respondents interested in a follow-up interview. Email addresses were used only for interview recruitment and scheduling, were stored separately from the survey responses, and were not linked to the analytical survey dataset. The full participant information and data-protection text presented with the survey are reproduced in Appendix A.

Before each interview, participants were informed about the purpose of the study, the expected duration of the interview, the use of audio recording, the transcription process, anonymisation, and their right to withdraw. Participants were informed

that they could stop the interview at any time without negative consequences.

With each participant's permission, the interviews were audio-recorded. During the consent process, participants were initially informed that the recordings would be transcribed locally using Whisper. Whisper was used for transcription, but the resulting transcript quality was insufficient for reliable qualitative analysis. The transcription workflow was therefore changed to the f4 transcription software provided by dr. dresing & pehl GmbH (audiotranskription.de).

The provider describes f4's automatic transcription service as GDPR-compliant. According to the provider's documentation, automatic speech recognition is performed on servers located in Germany, data are transmitted using encrypted connections, and uploaded recordings are deleted after processing. The provider also states that interview data are not used to train language models [10]. Before analysis, names, affiliations, company information, project names, and other details that could reveal participant identities were removed or generalised. The qualitative analysis was conducted using the anonymised transcripts.

Consent for interview participation, audio recording, use of anonymised quotations, and optional publication of fully anonymised transcripts were treated as separate decisions. Participants could take part in the interview and permit the use of anonymised quotations without agreeing to the publication of their complete anonymised transcript. Participants were informed that permission to publish a fully anonymised transcript could be revoked until July 2026.

Survey contact information was separated from the analytical survey dataset and was used only for interview recruitment and scheduling. Survey findings are reported only in aggregate form. Interview findings are reported using the anonymised participant identifiers I1–I9 and short anonymised quotations where appropriate.

5. Survey Results

This chapter reports the descriptive survey results. The survey results are not treated as the main empirical contribution of the thesis. Instead, they provide context for the interview study, show which handoffs appear relevant within the collected sample, and help motivate the later qualitative analysis. Although the survey used the term "transition", this thesis uses the term "development handoff" to refer to the same movement from one software development activity to another.

5.1. Data Preparation

The raw survey export contained 119 response records, including substantive survey responses, administrative fields, page metadata, and interview-participation information. The survey platform marked 80 responses as complete and 39 as incomplete. Platform completion status was not used as the only inclusion criterion because the interview-recruitment section was optional and was not required for the descriptive survey analysis.

Five responses marked as incomplete contained answers to all analytical questions but did not complete the optional interview-recruitment section. These responses were therefore retained. The remaining 34 incomplete records were excluded because they contained insufficient information for the planned analysis. The final analytical dataset consequently contained 85 responses.

The retained analytical variables were years of programming experience, self-rated expertise, breadth of programming experience, recent experience of being blocked, most difficult development handoff, duration of blocks, AI tool use, and perceived AI helpfulness. Five valid responses did not contain an answer to the duration question. These were different responses from the five platform-incomplete records retained above. The missing duration values were not imputed and are reported explicitly in Table 5.7. Missing answers to the optional interview-participation question did not affect eligibility for the descriptive analysis.

5.2. Respondent Profile

The survey sample included respondents with different levels of programming experience. Table 5.1 shows the distribution by years of programming experience. The sample was relatively experienced overall: 35 respondents had more than 10 years of programming experience, and 19 respondents had 6–10 years of experience.

This means that 54 respondents, or 63.5% of the sample, had at least six years of programming experience.

Table 5.1.: Programming experience of survey respondents

Programming experience	**Count**	**Percentage**
Less than 1 year	1	1.2%
1–2 years	8	9.4%
3–5 years	22	25.9%
6–10 years	19	22.4%
10+ years	35	41.2%

In addition to years of programming experience, the survey measured two further dimensions of experience: respondents' self-assessed programming expertise relative to developers with similar backgrounds and the breadth of their experience across projects, domains, and technologies. Tables 5.2 and 5.3 report the distributions of these responses. These measures are not treated individually as objective indicators of expertise. Instead, they provide complementary information for describing the sample and applying the multidimensional experience classification introduced in Section 4.4.

Table 5.2.: Self-assessed programming expertise of survey respondents

Self-assessed expertise	**Count**	**Percentage**
Much less experienced	4	4.7%
Slightly less experienced	13	15.3%
About average	35	41.2%
Slightly more experienced	21	24.7%
Much more experienced	12	14.1%

Most respondents assessed their expertise as comparable to or higher than that of developers with similar backgrounds. In total, 68 respondents, or 80.0% of the sample, described themselves as about average, slightly more experienced, or much more experienced. Seventeen respondents, or 20.0%, assessed themselves as slightly or much less experienced.

Table 5.3.: Breadth of programming experience of survey respondents

Breadth of experience	Count	Percentage
Very narrow, mostly one type of project or technology	4	4.7%
Somewhat narrow	19	22.4%
Moderately broad	31	36.5%
Broad	19	22.4%
Very broad	12	14.1%

The largest group comprised 31 respondents, or 36.5% of the sample, who described their experience as moderately broad. In total, 62 respondents, or 72.9%, reported moderately broad, broad, or very broad experience. The remaining 23 respondents, or 27.1%, described their experience as somewhat narrow or very narrow.

Taken together, the three indicators provide a broader description of the sample than years of programming experience alone. The respondent profile suggests that the sample was not dominated by respondents with little programming experience or limited breadth of technical exposure. However, the self-assessed expertise and breadth measures are interpreted descriptively and are not treated as objective standalone measures of programming ability.

5.3. Year-Based Experience Group Comparison

For the descriptive survey analysis, experience groups were approximated using years of programming experience. Table 5.4 shows recent blocking experience across these year-based groups. The multidimensional comparison used in the interview analysis also considers self-assessed expertise and breadth of experience.

Table 5.4.: Recent blocking experience by year-based experience group

Experience group	Total	Yes	No	Not sure
Early experience, $\leq$2 years	9	5	0	4
Intermediate, 3–5 years	22	14	6	2
Experienced, 6+ years	54	33	14	7

Reported blocking experiences appeared across all three year-based groups. Among respondents with six or more years of programming experience, 33 out of 54 reported experiencing a transition-related block in the previous month. This indicates that the reported blocking experiences within the sample were not limited to respondents with little programming experience.

At the same time, the early-experience group was small, containing only nine respondents, four of whom selected “Not sure”. The survey therefore does not provide

a sufficiently balanced basis for making strong comparisons between early-experience and experienced developers. Experience-related differences are examined primarily through the qualitative interview analysis.

5.4. Recent Experience of Being Blocked

Respondents were asked whether they had felt blocked or unable to make progress in the last month at or due to a transition between development activities. Table 5.5 shows the distribution of responses.

Table 5.5.: Recent experience of being blocked

Response	Count	Percentage
Yes	52	61.2%
No	20	23.5%
Not sure	13	15.3%

A majority of respondents reported experiencing a transition-related block in the last month. Specifically, 52 out of 85 respondents answered "Yes", while 20 answered "No" and 13 were "Not sure". This suggests that transition-related blocking was a common experience within the collected sample. However, because the survey was exploratory and not based on representative sampling, this result should not be interpreted as a general prevalence estimate for all software developers.

5.5. Most Difficult Development Handoffs

Respondents were asked which transition they found most difficult. The results are shown in Table 5.6. The most frequently selected transitions were related to understanding, deciding, designing, and starting implementation.

Table 5.6.: Most difficult development handoffs reported by respondents

Most difficult transition	Count	Percentage
Understanding requirements → deciding what to build	19	22.4%
Designing a solution → starting to code	19	22.4%
Deciding what to build → designing a solution	14	16.5%
Debugging → deciding on a fix	13	15.3%
Writing code → testing/debugging	8	9.4%
Other	7	8.2%
Code review/integration	5	5.9%

The two most frequently selected transitions were understanding requirements → deciding what to build and designing a solution → starting to code. Each was selected by 19 respondents, representing 22.4% of the sample. Together, the first three conceptual or pre-code transitions accounted for 52 out of 85 responses, or 61.2% of the sample. Among respondents who reported being blocked in the last month, these same conceptual or pre-code transitions accounted for 35 out of 52 responses, or 67.3%.

The most frequently reported difficult handoffs were therefore located before or around the start of code writing. Within the collected sample, difficulties were concentrated at points where respondents had to transform requirements or design understanding into a concrete implementation direction. Because the survey was exploratory, this pattern is reported descriptively and should not be interpreted as evidence of a general distribution across software developers.

5.6. Duration of Blocking Episodes

The duration question was shown to all respondents. Therefore, the results in Table 5.7 describe how respondents generally reported the duration of such blocks. The interpretation is most meaningful for respondents who also reported a recent blocking experience.

Table 5.7.: Reported duration of blocking episodes

Duration	**Count**	**Percentage**
It depends on several factors	23	27.1%
Around a day	16	18.8%
1–4 hours	15	17.6%
Several days or more	14	16.5%
15–60 minutes	10	11.8%
Less than 15 minutes	1	1.2%
Cannot say	1	1.2%
Missing	5	5.9%

The duration of blocking episodes varied across respondents. The most common response was “It depends on several factors”, indicating that many participants did not perceive blocks as having a fixed duration. Among respondents who reported a recent block and provided a concrete duration, 30 of 33 described blocks lasting at least one hour. This indicates that, for many of these respondents, the reported blocking experience represented more than a brief moment of hesitation. Duration is nevertheless treated only as a descriptive indicator rather than as a measure of the severity of Programmer’s Block.

5.7. AI Tool Use and Perceived Helpfulness

Respondents were asked whether they used AI-based tools, such as ChatGPT or GitHub Copilot, when they felt blocked at development handoffs. Table 5.8 shows the reported frequency of AI tool use.

Table 5.8.: AI tool use when blocked

AI tool use	Count	Percentage
Yes, often	37	43.5%
Yes, sometimes	25	29.4%
Rarely	11	12.9%
Never	12	14.1%

AI tool use was common in the sample. Overall, 62 out of 85 respondents, or 72.9%, reported using AI tools often or sometimes when blocked at development handoffs. Among respondents who reported a recent block, this proportion was slightly higher: 40 out of 52 respondents, or 76.9%, used AI tools often or sometimes when blocked. This indicates that AI tools were a commonly reported source of support during blocking situations within the collected sample.

Respondents also rated the helpfulness of AI tools on a scale from 1 to 5, where 1 meant "not helpful at all" and 5 meant "very helpful". The overall mean helpfulness rating was 3.18 out of 5, and the median was 3 out of 5. Table 5.9 shows mean helpfulness by AI use frequency.

Table 5.9.: Mean AI helpfulness by AI use frequency

AI use frequency	Mean helpfulness
Yes, often	3.97
Yes, sometimes	3.36
Rarely	2.18
Never	1.25

Respondents who used AI tools more frequently tended to rate them as more helpful. Those who used AI often had an average helpfulness rating of 3.97, while those who reported never using AI during blocking situations had an average rating of 1.25. This relationship should be interpreted cautiously. The survey does not establish whether AI directly reduces blocking experiences, and the helpfulness ratings of respondents who reported never using AI in such situations may reflect general perceptions rather than experience with AI during a specific block. It may also be the case that developers who already perceive AI as useful are more likely to use it frequently.

5.8. Summary and Limitations

Overall, the survey results show that blocking experiences at development handoffs were frequently reported within the collected sample. A majority of respondents reported experiencing a recent block, and the most frequently selected difficult transitions were concentrated around conceptual handoffs before or around code writing. In particular, transitions involving requirements understanding, deciding what to build, designing a solution, and starting implementation were prominent.

The results also indicate that AI tools are commonly used when developers feel blocked, although their perceived helpfulness is moderate rather than uniformly high. These survey findings provide useful context for the interview study, where the reasons behind these blocking experiences are explored in greater depth.

The survey has several limitations. First, the sample was exploratory and should not be treated as representative of all software developers. Second, most survey questions were closed-ended, which limited the depth of explanation available from individual responses. Third, the working definition of being blocked used in the survey was intentionally broader than the four-condition definition of Programmer's Block adopted in this thesis. The survey could not establish for each reported episode whether all required skills and infrastructure were available or whether the other conditions of Programmer's Block were satisfied. The survey results therefore represent self-reported blocking experiences relevant to Programmer's Block rather than independently verified cases of the construct. Finally, asking respondents directly about being blocked may itself have influenced how they classified their experiences.

For these reasons, the survey findings are interpreted as descriptive context rather than as prevalence estimates or standalone evidence of Programmer's Block. The interview data provide the main basis for examining how blocking episodes emerged, how developers responded to them, and how the reported experiences related to the phenomenon investigated in this thesis.

6. Interview Findings

This chapter presents the findings from the nine semi-structured interviews. The interviews provide detailed accounts of where developers became blocked, how they experienced the block, which conditions contributed to it, how they attempted to move forward, and how they perceived AI-based support.

The themes presented in this chapter were not defined before the interviews. They were developed progressively from the coded data. Each interview was first coded individually using open codes that remained close to the participant's account. At this stage, the purpose was to capture potentially relevant actions, conditions, experiences, and interpretations without forcing them into a predefined explanation of Programmer's Block. The resulting codes described, for example, missing documentation, uncertainty about where to make a change, repeated code-reading loops, delayed help-seeking, structured debugging, AI-supported summarisation, and concerns about trusting generated output.

The codes were then compared across the nine interviews and consolidated into a master codebook. Similar codes were merged when they represented the same underlying idea, while broad codes were separated when they covered meaningfully different experiences. This process resulted in 62 master codes organised into seven code families: participant and experience context, block location or development handoff, causes and conditions of the block, the lived experience of being blocked, coping and recovery strategies, AI use and perception, and experience-related differences. These families helped organise the dataset, but they were not treated as themes themselves. A code family such as "coping strategies", for example, groups related extracts by topic, whereas a theme needs to make a broader analytical claim about how or why recovery occurs.

The next analytical step was to examine relationships between codes across interviews. A cross-interview evidence matrix was used to compare where blocks occurred, which conditions contributed to them, what developers tried, how AI was involved, and how experience influenced the response. This comparison revealed that codes which initially appeared separate often formed part of the same underlying pattern. Missing documentation, unavailable original authors, inaccessible environments, and excessive documentation all pointed toward a broader problem of fragmented technical context. Similarly, logging payloads, creating temporary tables, writing notes, constructing test harnesses, and explaining the problem to another person all involved making previously hidden behaviour or reasoning observable.

Candidate themes were developed from these relationships and repeatedly checked against the coded extracts and complete transcripts. A theme was retained only

when it captured a coherent pattern across several interviews and contributed meaningfully to one or more research questions. Contrasting and limiting cases were also considered. For example, documentation was absent in some blocking cases, whereas in another case substantial documentation existed but remained too complex to form a usable mental model. Likewise, AI was central to some participants' workflows, selectively used by others, and unavailable in one case because of organisational privacy restrictions. These differences were used to refine the themes rather than being treated as exceptions to remove.

The final analysis resulted in five main themes. The first explains how blocks emerged when partial understanding could not be converted into a sufficiently safe next action. The second examines how lost or fragmented technical context turned development handoffs into blocking situations. The third describes how developers recovered by making system behaviour and their own reasoning visible. The fourth explains why AI was most useful as supervised sense-making support rather than as an autonomous problem solver. The fifth examines how experience changed developers' responses to blocks more than it eliminated the blocks themselves. A socio-technical interpretation runs across all five themes and is discussed separately as a cross-cutting pattern rather than as an additional independent theme.

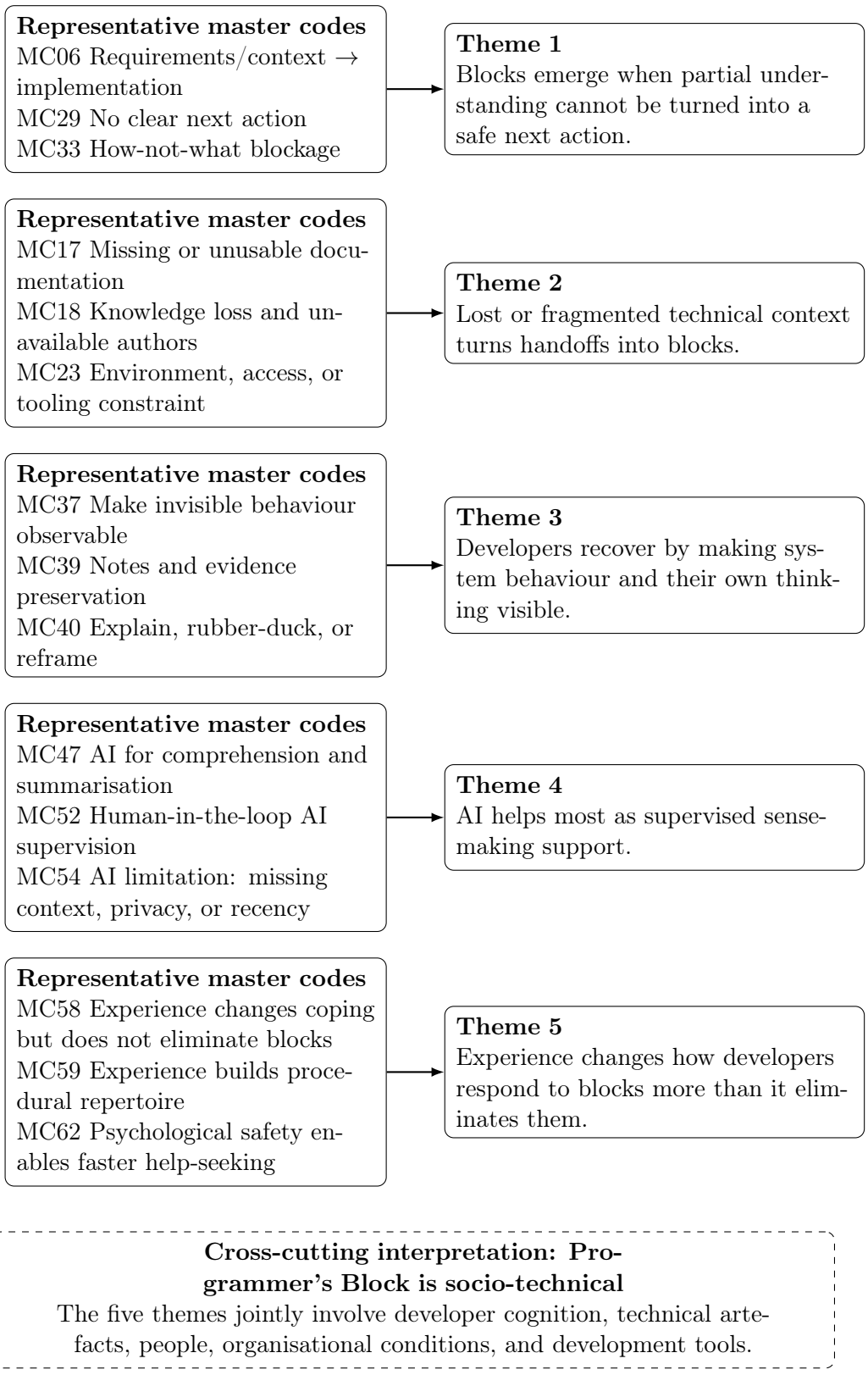

Figure 6.1.: Illustrative traceability from representative master codes to the five final themes. The examples are selective rather than exhaustive; the complete master codebook is provided in Appendix C. Individual codes could contribute to more than one analytical pattern.

The themes are not ranked by frequency. Each represents an analytically meaningful pattern supported by multiple interviews, while some interviews provide stronger evidence for particular themes than others. Quotations were lightly cleaned by removing filler words, repeated fragments, and clear transcription errors where this did not alter the participant's meaning. Clarifying additions are shown in square brackets, and paraphrased coded extracts are not presented as direct quotations.

Table 6.1 provides a short overview of the participants and their main blocking cases. The participant identifiers I1–I9 are used throughout the chapter.

Table 6.1.: Overview of interview participants and main blocking cases

ID	Participant context	Main blocking case
I1	Experienced enterprise maintenance developer	Legacy integration between older application components and Office 365/Graph under undocumented compatibility changes and restricted client access.
I2	Highly experienced full-stack and solo developer	Ground-up platform rebuild in which large components could not be translated into a clear development order or next action.
I3	Intermediate full-stack developer with broad tool exposure	Adoption of poorly documented packages and SDKs, including a PDF-processing case that required a Python sidecar instead of the intended Go solution.
I4	Experienced equipment-integration engineer	Inherited framework components that did not work as advertised, lacked documentation, and could not be clarified with the original authors.
I5	Data engineer with approximately five years of experience	Debugging a dynamic SQL stored procedure that produced duplicate aggregated records for one country-specific payroll feed.
I6	Backend Node.js developer with more than three years of experience	Missing requirements and resources, cross-role dependencies, and a Python-to-Node.js migration involving package, deployment, and environment differences.
I7	Career-transition and research-software developer	React Native/Expo and Spring Boot integration failures, followed by module-integration and refactoring blocks in an autonomous-shuttle monitoring system.
I8	Mid-level backend engineer moving from Go to Kotlin	Difficulty contributing in a new team and fast-changing codebase while understanding the architecture but lacking confidence in Kotlin syntax.
I9	Experienced backend and event-driven developer	Changing complex behaviour in an inherited migration service after several original contributors had left.

6.1. Blocks Emerge When Partial Understanding Cannot Be Turned into a Safe Next Action

Across the interviews, participants were rarely blocked because they had no idea what the task was. More often, they understood the goal but could not decide on a concrete and safe next action. For example, I2 knew that an ageing platform needed

to be rebuilt but could not decide which component to start with, while I5 knew that a stored procedure produced incorrect output but could not locate the responsible logic. In both cases, the block appeared between understanding the problem and deciding what to do next. Similar blocks occurred before implementation, during debugging, at integration boundaries, and while changing inherited behaviour.

The clearest planning example came from I2. The participant understood that an ageing platform required a ground-up rebuild, but struggled to break the work into smaller components, decide what should be developed first, and determine where to start. The amount of possible work created a state of prioritisation paralysis:

> *"It just leaves me basically staring at my screen going, what the hell am I supposed to be doing?"* (I2)

The episode was not readily explained by a lack of general programming experience: I2 had more than twenty years of professional development experience. The block occurred when the participant had to turn the overall rebuild goal into decisions about architecture, task order, and implementation. Competing billable work and repeated project switching added another difficulty because the participant first had to reconstruct where the work had stopped.

I4 described the same gap between knowing the goal and knowing how to reach it. The required behaviour was clear, but the inherited framework was unfamiliar and undocumented:

> *"I was really stuck ... not what I wanted to do, but how to implement them."* (I4)

The gap between understanding and action also appeared during debugging. I5 knew that a stored procedure produced incorrect duplicate aggregations, but the procedure contained many conditional branches and the participant could not locate the responsible logic. Repeatedly reading the same code did not create new understanding:

> *"I realised that I [was] reading the same logic again and again and again and not really reaching the root cause."* (I5)

A similar block occurred during integration and migration. In I7, the frontend worked with mock data and the backend worked in Postman, but the complete system failed because assumptions about JSON serialisation, CORS, Spring Security, and device networking did not align. The participant understood the separate components but did not yet know which assumption at the boundary needed to change. In I6, the participant knew Node.js but could not directly reproduce the behaviour of a Python implementation because packages and libraries did not map cleanly across the two ecosystems. I6 therefore described the problem as a technology-stack compatibility issue rather than a lack of basic programming skill. In both cases, partial understanding was insufficient for choosing the next safe change.

Not all reported blocking episodes aligned equally closely with the strict definition of Programmer's Block. I8 provides an important boundary case. The participant was learning Kotlin and explicitly linked part of the difficulty to low confidence in the language syntax. At the same time, the participant described a rapidly changing codebase, changing requirements, and insufficient time to build a stable understanding of the new system. This episode is therefore treated primarily as evidence about transition-related blocking and AI-assisted recovery rather than as an unambiguous instance of Programmer's Block.

I9 provides a different case. The participant understood the purpose of an inherited migration service but could not safely change its behaviour until the merging and migration rules could be held together in one mental model. Here, the difficulty concerned converting substantial but fragmented understanding into a sufficiently confident implementation decision.

Together, these accounts show that blocking often centred on selecting the next action. Participants could describe the goal, the system, or the incorrect result, but could not determine what to change next without creating unacceptable risk. This theme directly contributes to RQ1 by locating Programmer's Block at handoffs where partial understanding must be turned into an implementation, debugging, or integration decision.

6.2. Lost or Fragmented Technical Context Turns Handoffs into Blocks

A second pattern concerned the quality and availability of context. Several participants inherited code, frameworks, services, or requirements that could not be understood from the available information. In these cases, the handoff was not only between development activities. It was also a handoff across people, teams, versions, tools, or time.

I1 encountered this problem during a legacy integration in which selected email components had to be upgraded without upgrading the complete application. The participant encountered API changes, binary compatibility problems, undocumented modifications, and a client environment in which the usual debugging approach was not possible. In particular, the participant could not attach a debugger. Previous contributors were no longer available, and small undocumented modifications made in earlier versions repeatedly created new uncertainty:

> *"Then we just hit block after block, [because of] all sorts of little changes made by people who no longer work for the company."* (I1)

The environment made the context difficult not only to understand but also to observe. The participant had to work through a VPN, a jump server, and an Azure Virtual Desktop. The block was therefore produced by the interaction between

legacy code, missing knowledge about earlier modifications, and restricted access to debugging tools.

I4 reported a closely related case. The framework did not work out of the box as advertised, the participant had only recently started using it, documentation was absent, and the original contacts were unavailable or in another time zone. Importantly, the participant stated that the requirements were clear. The missing information concerned the capabilities and behaviour of the artefact that had been handed over. This shows that unclear implementation context can create a block even when the business goal is understood.

Documentation problems also appeared in newer technical environments. I3 became blocked when packages, libraries, or SDKs did not provide enough documentation. The participant's PDF-processing case required an architectural workaround because the intended Go-based route was not workable. I7 similarly described a rapidly changing framework in which AI suggestions were outdated and even the official documentation did not fully reflect package restructuring. These cases show that context loss is not limited to old legacy systems. It can also occur when technologies change faster than documentation and community knowledge.

I6 extended this pattern beyond technical documentation. The participant described missing requirements, incomplete resources, dependencies on testers or other developers, and the need to request endpoints, URLs, and technology-stack information. AI could suggest technical alternatives, but it could not provide business information that had never been supplied. In this case, the block existed because the required context was located in the organisation rather than in the code.

I9 provides an important limiting case. The inherited migration service had substantial documentation, but the amount and complexity of the material did not enable the participant to form a clear mental model. The participant explained that there was "so much documentation", yet reviewers still lacked enough context to judge the expected behaviour. This means that documentation quantity alone does not prevent blocks. Context must also be usable, connected to the current task, and understandable within the time available.

Across these interviews, the common problem was fragmented ownership of knowledge. Information was distributed across former contributors, business roles, documents, production environments, and rapidly changing tools. The developer became blocked when this distributed context could not be reconstructed well enough to continue safely. This finding supports the view that Programmer's Block at handoffs is socio-technical: the technical task cannot be separated from how knowledge is created, transferred, documented, and made accessible.

6.3. Developers Recover by Making System Behaviour and Their Own Thinking Visible

The strongest recovery pattern across the interviews was externalisation. Participants moved forward by taking information that was hidden in a large system or held only in their own working memory and turning it into something visible, smaller, and testable. This included test harnesses, temporary tables, logs, notes, diagrams, explanations, pseudocode, and conversations.

I1 recovered by reconstructing the inaccessible client context. The participant pulled the source code for the client's exact version, used the same dynamic-link libraries, and created a local test harness with shims. This changed the problem from an inaccessible production failure into a behaviour that could be reproduced and inspected. The same participant also stressed the value of preserving evidence, writing notes, discussing the problem at a whiteboard, and explaining it to another person:

> *"How can you solve a problem if you can't explain it to somebody?"* (I1)

I4 followed a similar path when attempts to contact the original authors failed. The participant went through the framework line by line and documented it. Missing documentation had created the block, and producing documentation became part of the recovery. I4 also described discussing a technical problem with their young children after first translating it into a simple scenario they could understand. The children's basic questions exposed assumptions that the developer had not questioned. This illustrates that externalisation is not only a way to record information; it can also change the perspective from which the problem is understood.

I5 made intermediate system states visible during SQL debugging. Instead of reading the complete stored procedure repeatedly, the participant broke it into sections, stored intermediate results in temporary tables, compared working and failing country-specific paths, and wrote down expected and actual outputs. This created observable checkpoints and turned a circular investigation into a controlled sequence of comparisons.

I7 used the same general strategy across a multi-layer integration problem. The participant logged exact payloads on the frontend and backend, validated data-transfer-object mappings, tested the physical-device network path, and checked CORS and security configuration. In a later research-software case, the participant isolated the smallest reproducible unit, printed types, checked environment variables, and verified each system layer. The participant summarised the broader lesson as follows:

> *"I have learned that Programmer's Block is not a lack of ability. It's a mismatch between [the] current mental model and the new system demands."* (I7)

Stepping away was another way of changing perspective. I1 described walks, tea breaks, and even bike rides as legitimate thinking time. I5 also recommended taking a step back and starting again with a fresh perspective when repeated code reading produced mental fatigue. These breaks were not presented as abandoning the problem. They allowed participants to interrupt an unproductive loop and return with a different mental representation.

Writing also supported thinking for I8 and I9. I8 sometimes wrote by hand or entered unstructured notes when designing something complex, even when the notes were not used later. The value was in the act of externalising thought. I9 used an AI-generated rule table and concise written summaries to hold complex migration logic outside working memory. The participant also described explanation during code review as a diagnostic activity: while explaining the change, they sometimes realised that their own understanding was incomplete.

The recovery strategies therefore followed two related directions. First, developers made invisible system behaviour visible through logging, controlled experiments, intermediate outputs, and reproducible environments. Second, they made invisible reasoning visible through notes, diagrams, explanation, pseudocode, and discussion. Both directions reduced the amount of information that had to be held mentally at once and created concrete points at which assumptions could be checked.

6.4. AI Helps Most as Supervised Sense-Making Support

AI use differed substantially across participants, but a consistent pattern appeared in the situations where participants considered it useful. AI was most helpful when it supported comprehension, generated alternatives, handled repetitive work, or translated an existing high-level intention into an output that the developer could inspect. Participants were less willing to rely on AI when the problem required missing business context, project-specific production knowledge, current framework information, or independent responsibility for a high-level decision.

Several participants used AI to reduce the effort required to understand or structure a problem. I5 used Copilot to identify relevant SQL logic, understand business rules, and reason about columns that should or should not be part of an aggregation. I6 asked Copilot for several possible solutions and selected an approach using expected outputs and tests. I7 used AI across idea generation, design, implementation, and debugging, while still checking other sources when framework information was outdated. I9 asked AI to simplify migration rules into a table and to produce concise summaries of long code sections that could be read repeatedly.

I8 represented the strongest AI-integrated case. The participant understood the architecture but lacked confidence in the syntax of the new language. They guided agents by identifying relevant files, describing the intended feature, and pointing to earlier pull requests. The generated code was then reviewed, criticised, and revised:

> *"Essentially, I use AI for almost all the code that I'm writing today."* (I8)

Although AI produced much of the code, the participant did not describe a fully autonomous process. Human judgement remained central in deciding which files mattered, whether the implementation was overengineered, and which revisions were needed. I9 expressed this supervised relationship directly:

> *"I use it more as a pair programming partner where I know what I want to do."* (I9)

Other participants used AI more selectively. I1 considered AI useful for unit tests, code exploration, and dependency graphs, but not for an installation-specific legacy problem. I2 used AI for boilerplate or work that might otherwise be delegated to a junior developer, but did not trust it to determine architecture or planning. The participant argued that an AI-generated plan would still have to be checked for missing items, incorrect flow, hallucinations, and outdated knowledge. In such a case, verification could become as demanding as producing the plan directly.

The interviews also identified clear limits. I4 did not use AI in the main blocking case because organisational privacy and data rules prevented proprietary or third-party material from being shared with an AI tool. I6 stated that AI could not solve a block caused by missing organisational knowledge:

> *"If we are talking about the business blockage, then definitely [AI] won't help me."* (I6)

I7 reported that rapidly changing frameworks could make both AI responses and documentation outdated. I3 warned that fast AI-assisted development could hide missing requirements, security problems, or bugs that developers did not yet know they had created. I8 raised two further concerns: the possible loss of hard-earned coding skills and the cognitive limit of supervising several agents at the same time:

> *"I worry that I'm losing some of the skills that I worked hard to get by not actually writing the code myself."* (I8)

These accounts suggest that AI does not simply remove Programmer's Block. It can reduce syntax burden, working-memory load, repetitive work, and the effort of generating alternatives. At the same time, it can introduce a new verification and supervision task. AI support was strongest when the developer could provide sufficient context, recognise a useful answer, and remain responsible for the final decision. This theme answers RQ2 by showing that AI was used mainly as supervised cognitive and implementation support rather than as a reliable independent problem solver.

6.5. Experience Changes How Developers Respond to Blocks

The interview sample contained intermediate and experienced participants but no participants who met the study's novice classification. The experience-related pattern therefore emerged from comparisons across the available participant profiles together with participants' retrospective reflections on earlier stages of their careers. These accounts suggest that experience influenced how developers reacted to blocks and attempted to recover more clearly than it influenced whether blocking occurred at all. Earlier-career reflections placed greater emphasis on panic, self-doubt, delayed help-seeking, and trying several approaches without fully understanding the problem. Current accounts with greater experience more often described structured investigation, earlier escalation, and the deliberate use of knowledge gained from previous problems.

I9 provided the clearest comparison between earlier and current experience. Earlier in the participant's career, being stuck created panic and reluctance to ask questions. The participant wanted to appear competent and therefore continued trying different approaches without understanding why they failed. With more experience, the participant changed this behaviour:

> *"If I'm stuck for half an hour, I go and ask someone. I don't care."* (I9)

This change was not explained by experience alone. I9 also worked in a team where asking repeated or apparently simple questions was accepted. Psychological safety therefore made it easier to use help-seeking as a recovery strategy. Experience helped the participant recognise when individual effort was no longer productive, while the team environment made escalation socially acceptable.

Other interviews also showed that substantial experience did not remove blocking situations. I2 had more than twenty years of professional development experience but still became blocked while deciding how to decompose and prioritise a large platform rebuild. The participant understood the overall goal but could not identify the next development step:

> *"It just leaves me basically staring at my screen going, what the hell am I supposed to be doing?"* (I2)

This case shows that experienced developers can still become blocked when the task involves architecture, prioritisation, solo responsibility, or competing business demands. Experience did not provide an immediate answer to an unfamiliar and uncertain situation.

Experience appeared to provide a broader set of recovery strategies. I1 preserved evidence, recorded attempted solutions, discussed the problem with colleagues, and reconstructed a test environment instead of continuing with unstructured trial and error. I4 used previous cases, decomposition, and self-produced documentation to

understand an inherited framework. I5 explained that the same SQL debugging problem could still occur in the future, but that experience would make it easier to divide the procedure into sections and identify the root cause. I6 similarly expected migration and setup problems to become easier once the complete procedure and the necessary project information were more familiar.

The interviews also suggest that experience may change the kinds of uncertainty developers encounter. Earlier-career descriptions included unfamiliar frameworks, development environments, setup procedures, and uncertainty about when to ask for help. Experienced participants more often described blocks involving architecture, inherited systems, integration, documentation, organisational knowledge, and responsibility for changes. However, this should not be interpreted as a strict division between novice and experienced developers. The interviews mainly provide retrospective comparisons, and similar forms of block appeared across different experience levels.

The relationship between experience and AI use was less clear. Some intermediate developers, such as I3 and I8, used AI extensively, while some highly experienced developers, such as I1 and I2, used it more selectively. I9 adopted AI cautiously despite substantial experience. AI adoption therefore appeared to depend on the task, organisational rules, trust, and personal working style rather than years of experience alone. This pattern is discussed primarily in the previous theme on AI support.

For RQ3, the interviews therefore suggest that participants perceived experience as changing the emotional response to being blocked, the timing of help-seeking, and the range of available recovery strategies more than it changed whether blocking could occur. Across the retrospective accounts, participants described becoming better able over time to recognise unproductive approaches, change strategy, preserve useful information, and seek support. Because no current novice developer was interviewed, these findings should be interpreted as experience-related patterns within the available participant profiles and retrospective accounts rather than as a direct comparison between current novice and experienced developers.

6.6. Cross-Cutting Pattern: Programmer's Block Is Socio-Technical

Across the five themes, Programmer's Block did not appear as a purely individual cognitive failure or as a purely technical defect. The blocking episodes were produced by interactions between technical systems, people, organisational processes, and the developer's cognitive or emotional state.

Technical factors included legacy compatibility, complex conditional logic, integration contracts, package differences, unfamiliar syntax, and fast-changing frameworks. Social and organisational factors included unavailable authors, missing requirements, stakeholder uncertainty, solo responsibility, privacy restrictions, and limited reviewer

context. Cognitive and affective factors included mental-model overload, fatigue, low task enjoyment, self-doubt, anxiety about appearing incompetent, and difficulty restoring focus after switching tasks.

The same interaction was visible in recovery. Developers used technical actions such as logging, temporary tables, test harnesses, and controlled experiments together with social and cognitive actions such as asking colleagues, consulting business analysts, writing notes, explaining the problem, reframing it, or taking a break. AI tools became another part of this socio-technical system: they could support explanation, memory, and implementation, but their value depended on data access, organisational policy, current knowledge, and human evaluation.

This cross-cutting pattern helps explain why the same developer may move quickly through one task but become blocked in another. Ability alone is not sufficient when the required context is unavailable, the system cannot be observed, the organisational handoff is incomplete, or the developer cannot yet form a stable model of what a safe change requires.

6.7. Summary of Interview Findings

The interview findings provide the following answers to the research questions.

For RQ1, Programmer's Block was experienced most clearly when developers could not convert partial understanding into a safe next action. The blocks appeared during conceptual planning, inherited-code comprehension, debugging, integration, migration, deployment, and task re-entry. Common causes included fragmented context, hidden system behaviour, complex interacting rules, dependencies on other roles, and cognitive or affective pressure. Developers attempted to recover by decomposing the problem, making intermediate states visible, reconstructing context, documenting and explaining their reasoning, taking breaks, and seeking social support.

For RQ2, AI tools were used for code comprehension, summaries, rule tables, alternative solutions, scaffolding, repetitive implementation, debugging ideas, and working-memory support. AI was most useful when developers could provide the relevant context and evaluate the result. Its usefulness was limited by missing business information, privacy restrictions, installation-specific problems, outdated knowledge, hallucination risk, hidden bugs, and the effort required to supervise or verify generated output.

For RQ3, blocking episodes occurred among both intermediate and experienced participants. The interview accounts suggest that experience influenced the form of the difficulty and, more clearly, the way participants responded to it. Current accounts with greater experience emphasised system-level and organisational uncertainty, broader coping repertoires, and earlier changes of strategy or help-seeking. Retrospective earlier-career reflections placed greater emphasis on confidence loss, panic, setup uncertainty, and delayed help-seeking. These findings do not provide a direct novice–experienced comparison because no interview participant met the

novice classification. AI adoption also did not follow a simple experience gradient; cautious and highly AI-integrated practices appeared across different participant profiles.

7. Discussion

This chapter discusses the survey and interview findings in relation to the research questions and the literature reviewed in Chapter 3. The purpose is not to repeat the themes presented in Chapter 6, but to consider what they mean when viewed together. In particular, the discussion focuses on why development handoffs appear to be relevant locations for Programmer's Block, what role context and sense-making play in blocking and recovery, how AI changes the work involved in moving forward, and how experience influences the response to being blocked.

7.1. Interpretation of the Findings

Taken together, the findings suggest that Programmer's Block at development handoffs cannot be explained only by difficulty with writing code. Across the interviews, participants were often able to describe the overall task, the intended behaviour, or at least part of the problem. What became difficult was turning that understanding into an action they considered sufficiently justified and safe. This interpretation is consistent with the observation by Schantong et al. that the causes of Programmer's Block may originate outside the immediate act of code writing [25]. The findings of this thesis extend this observation by showing why the movement between development activities may be particularly important.

Development handoffs as transformation points. A development handoff requires more than simply finishing one activity and starting another. It often requires the developer to transform one representation of a problem into another. Requirements need to become design decisions, a design needs to become executable code, an observed failure needs to become a diagnosis, and a diagnosis needs to become a change that can be implemented safely. The difficulty therefore lies partly in the transformation itself.

This interpretation helps explain why the survey results were concentrated around conceptual handoffs. The three transitions involving requirements understanding, deciding what to build, designing a solution, and starting implementation accounted for 52 of the 85 responses, or 61.2% of the sample. Among respondents who reported a recent block, 35 of 52 selected one of these three transitions. These results do not establish that conceptual handoffs are generally the most common source of Programmer's Block, because the survey was exploratory. However, they support the interview finding that a block can emerge even before the difficulty becomes visible in code.

This also provides a more specific interpretation of what it means to be unable to progress despite possessing the necessary programming ability. A developer may know the goal and have the technical ability to implement a solution, but still be unable to determine which available action is appropriate. In this sense, the block is not necessarily a complete absence of understanding. It can instead arise when the developer has *partial understanding without sufficient confidence to act on it*. Development handoffs expose this problem because they are moments at which understanding has to become a concrete decision.

Context needs to be usable, not simply available. The findings also refine the role of context in blocking situations. Prior research shows that developers invest substantial effort in recovering knowledge about program behaviour, dependencies, design decisions, task history, and the people who possess relevant information [14, 13]. Similar difficulties were visible in the interviews, but the cases suggest that the problem is broader than simply "missing documentation".

Context took several forms in the reported blocking episodes. Information could be absent because previous contributors had left, distributed across different people or systems, inaccessible because of environment restrictions, outdated because a technology had changed, or difficult to interpret because of the size and complexity of the available material. Importantly, one participant described having substantial documentation while still being unable to form a usable understanding of the system. This indicates that the amount of available information alone is not a good indicator of whether a developer has sufficient context.

A more useful distinction may therefore be between *available information* and *usable context*. Information becomes useful for overcoming a block only when the developer can connect it to the immediate decision that needs to be made. A large amount of documentation may still leave questions unanswered about which behaviour is important, why a previous decision was made, which assumption remains valid, or where a change can safely be introduced. Conversely, a short conversation with a knowledgeable colleague may provide exactly the missing relationship between the available information and the current task.

This interpretation is also consistent with research on onboarding and context-dependent expertise. Newcomers must learn not only source code but also tools, processes, conventions, people, and the wider project landscape [8, 4]. The interviews in this study suggest that a similar local loss of expertise can occur when an otherwise experienced developer moves into an unfamiliar framework, codebase, service, or organisational context. The relevant question is therefore not only whether the developer has information, but whether they can use that information to reconstruct an adequate working model of the current problem.

Recovery can be understood as reducing uncertainty. The coping strategies described in Chapter 6 initially appear diverse. Participants constructed test harnesses, created temporary tables, logged payloads, compared expected and actual

results, reduced problems to smaller reproducible cases, wrote notes and pseudocode, explained problems to colleagues, and sometimes temporarily stepped away from the task. Considered together, however, these strategies performed a similar function: they reduced uncertainty by turning something that was difficult to reason about into something that could be inspected.

Some strategies externalised the behaviour of the technical system. Logging, intermediate outputs, test harnesses, and controlled experiments allowed participants to observe what the software was actually doing rather than relying only on assumptions about its behaviour. Other strategies externalised the developer's reasoning. Notes, diagrams, explanations, pseudocode, and conversations made assumptions visible and easier to challenge. In both cases, the developer created evidence against which the current mental model could be checked.

This provides a possible explanation for why activities that do not immediately produce production code can still contribute directly to recovery. Writing down an assumption, explaining the problem to another person, or taking time to create a small reproduction may appear to delay implementation. In the reported cases, however, these activities helped participants leave repeated and unproductive reasoning loops. Progress was restored not necessarily by working faster, but by changing the form in which the problem could be examined.

This interpretation also helps distinguish a productive pause from a blocking episode. Stepping away, discussing alternatives, or delaying a decision is not necessarily evidence of Programmer's Block. Such activities may themselves be part of productive problem solving when they generate new information or a different representation of the problem. The relevant issue is whether the developer remains unable to obtain or construct a meaningful direction despite actively attempting to continue.

AI redistributes reasoning work rather than removing it. The combined findings provide a similarly nuanced interpretation of AI-assisted support. In the survey, 62 of 85 respondents, or 72.9%, reported using AI tools often or sometimes when blocked. Among respondents who reported a recent blocking episode, this proportion was 40 of 52, or 76.9%. At the same time, the overall mean helpfulness rating was 3.18 out of 5. AI was therefore commonly used within the sample, but it was not perceived as uniformly capable of resolving blocking situations.

The interviews help explain this difference. AI was particularly useful when the developer already possessed relevant context but needed help processing, restructuring, or acting on it. Participants used AI to summarise code, explain existing logic, produce alternative approaches, generate repetitive implementation, create tests, or transform complicated rules into simpler representations. This resembles the exploration mode described by Barke et al., in which developers use AI suggestions while they are still deciding what direction to take [3].

However, the findings also suggest that AI may move rather than eliminate the difficult part of the task. Without AI, a developer may be blocked while producing

a candidate solution. With AI, several candidate solutions can be generated quickly, but the developer must then determine whether one of them is correct, appropriate, maintainable, secure, and consistent with the surrounding system. This is consistent with prior findings that developers spend considerable effort inspecting, interpreting, accepting, rejecting, and revisiting generated suggestions [18]. The difficult reasoning can therefore move from *solution generation* toward *solution evaluation*.

A second boundary concerns the source of the missing context. AI can help process information that can be supplied to it, but it cannot reliably reconstruct information that is genuinely absent or unavailable. In the interviews, this included project-specific production knowledge, undocumented organisational history, missing business decisions, and information that could not be provided because of privacy restrictions. This distinction may help explain why AI was highly useful in some blocking episodes and of limited value in others. AI appears most useful when the problem is "I have context, but I cannot make sense of it efficiently", and less useful when the problem is "the context required to make the decision is not available".

Could more capable AI change the form of Programmer's Block? The increasing capability of AI-based development tools raises a further question: if more of the cognitive work of programming is delegated to AI, will developers continue to experience Programmer's Block in the same way? One possibility is that some forms of Programmer's Block become less visible. If an AI system can translate a high-level description into an implementation, a developer who is uncertain about how to express a solution in code may no longer need to overcome that difficulty directly. However, existing research suggests that generative AI does not simply remove cognitive work. Instead, it can change where that work occurs. Lee et al. found that the use of generative AI in knowledge work shifted critical-thinking activities toward verifying information, integrating generated responses, and maintaining responsibility for the overall task [15]. Similarly, Tang et al. showed that developers validating and repairing LLM-generated code still performed substantial inspection and search activities, with AI-related validation in some conditions producing higher cognitive workload [29].

From the perspective of Programmer's Block, this suggests that increasingly capable AI may *shift* the block rather than necessarily eliminate it. A developer may become less likely to be blocked by the question "How do I write this code?" but more likely to face questions such as "What exactly should I ask the AI to do?", "Does this generated solution actually satisfy the system constraints?", or "Can I safely accept this change?" The cognitive bottleneck could therefore move upstream toward specifying intent and supplying context, or downstream toward verification, integration, and responsibility for generated changes. This interpretation is consistent with the findings of this thesis, where AI was most useful when relevant context could be supplied and its output could be evaluated by the developer.

A more substantial change could occur if future AI systems perform not only code generation but also task decomposition, implementation, testing, debugging,

and revision with limited human involvement. Under such conditions, Programmer's Block as originally defined may become less visible at the code-writing level because the developer would no longer be responsible for performing much of the implementation directly. This would not necessarily mean that blocking disappears. Instead, difficulty could move toward formulating the problem, communicating the required context, supervising generated work, or deciding whether the resulting change can be trusted. Programmer's Block could therefore remain relevant even if the activity in which the block appears changes as the developer's role becomes increasingly supervisory.

A related concern is whether AI could make developers less likely to experience Programmer's Block in the short term while making them more vulnerable when AI support is unavailable or insufficient. If developers routinely delegate tasks such as decomposition, implementation, debugging, or explanation to AI, they may need to perform less of the reasoning that would otherwise contribute to building and maintaining their own understanding of the system. In the short term, this could reduce friction and help developers move past situations that might otherwise become blocking episodes. However, the same reliance could become problematic when the AI cannot access the required context, produces an incorrect solution, is restricted by organisational policy, or is simply unavailable.

This possibility is particularly relevant because the interview accounts suggested that AI was most useful when participants could supervise and evaluate its output. Participants needed enough knowledge to recognise inappropriate assumptions, missing requirements, outdated information, or unnecessarily complicated solutions. One participant who used AI for almost all current code writing also expressed concern about losing skills that had previously been developed through writing the code directly. This tension is consistent with research suggesting that greater confidence in generative AI can be associated with reduced self-reported critical-thinking effort [15]. Research on AI-assisted programming similarly raises questions about whether improvements in task performance necessarily correspond to an equally strong understanding of why a generated solution works [26]. These findings do not establish that AI use causes developers to lose programming ability. Rather, they motivate a longitudinal question that cannot be answered by the present study: whether repeated AI-supported problem solving changes the mental models and recovery strategies developers can draw upon when AI is unable to provide useful support.

This leads to an important future direction for Programmer's Block research. Rather than asking only whether AI helps developers overcome a block, future studies should examine whether increasingly autonomous AI *eliminates*, *shifts*, or *postpones* the cognitive difficulty involved. They should also investigate whether repeated AI-supported recovery changes developers' ability to recover independently when AI assistance is unavailable or unable to resolve the problem. As the developer's role moves from directly producing code toward directing, evaluating, and supervising AI-generated changes, both the location and the meaning of Programmer's Block may change with it.

Experience appears to expand the recovery repertoire. The findings also challenge a simple assumption that Programmer's Block should primarily be associated with inexperienced programmers. In the survey, 33 of the 54 respondents with six or more years of programming experience reported a recent transition-related block. The interview participants likewise included developers with substantial professional experience who still described situations in which they were unable to make meaningful progress. This is consistent with the original Programmer's Block study, which identified blocks among experienced programmers [25], and with the broader view that software-development expertise is context-dependent [2].

The interview accounts suggest that experience may matter less for whether a block can occur and more for what happens after the developer recognises it. Participants described learning to recognise repeated unproductive attempts, decomposing problems more deliberately, preserving evidence, drawing on analogous situations, switching strategy, and asking another person for help earlier. Experience can therefore be interpreted as providing a larger *recovery repertoire*: more ways of deciding what information is missing and more strategies for obtaining or testing it.

This interpretation also avoids treating years of experience as a simple protection against difficulty. An experienced developer can become locally inexperienced when moving into a new language, framework, system, team, or business domain. Prior research similarly describes expertise as dependent on knowledge, practice, and context [2]. Programmer's Block may therefore occur at different levels of abstraction across a career. A less experienced developer may struggle with unfamiliar tools or implementation choices, while an experienced developer may become blocked by architecture, integration, historical design decisions, or organisational dependencies.

The comparison must nevertheless remain cautious. No interview participant met the study's novice classification. Evidence about earlier-career reactions therefore came mainly from experienced participants reflecting retrospectively on their own development. These accounts suggest differences in confidence, persistence, help-seeking, and emotional response, but they cannot replace direct observations of developers who are currently at the beginning of their careers. RQ3 should therefore be interpreted primarily as showing how participants perceived the influence of experience on their response to blocks rather than as a controlled comparison between current novice and experienced developers.

Survey categories simplify episodes that are interconnected in practice. Considering the survey and interview findings together also reveals a methodological and conceptual difference between identifying a difficult handoff and understanding a complete blocking episode. The survey required respondents to select the transition they considered most difficult. This produced a useful overview and showed a concentration around conceptual and pre-code handoffs. The interviews, however, showed that a real blocking episode did not necessarily remain within one transition.

For example, an inherited-code problem could begin with trying to understand existing behaviour, continue into deciding where a change should be made, move

into implementation, and return to debugging when the behaviour did not match expectations. Similarly, a migration or integration problem could require repeated movement between design, coding, testing, environment configuration, and discussion with other people. The interviews therefore suggest that handoffs are useful analytical points at which a block may emerge, but the block itself can extend across several interconnected activities.

This distinction is important because software development is not a strictly linear process. The term "handoff" in this thesis should therefore not be interpreted as implying a fixed sequence of development stages. Instead, it identifies moments where the developer must transform or transfer understanding in order to continue. A failure at one of these transformations can produce repeated movement between activities while the developer attempts to reconstruct enough context to regain a direction.

Taken together, these interpretations suggest that Programmer's Block at development handoffs is best understood neither as a purely cognitive failure nor as a purely technical obstacle. Blocking emerged from the relationship between the developer's current understanding, the information available in the surrounding technical and organisational environment, and the decision demanded by the next development activity. Recovery similarly involved combinations of technical investigation, cognitive externalisation, social support, and, in some cases, AI assistance. Development handoffs provide a useful analytical lens because they expose the point at which available understanding must become action.

Figure 7.1 summarises this interpretation. It is intended as a synthesis of the findings rather than as a formal theory or a causal model.

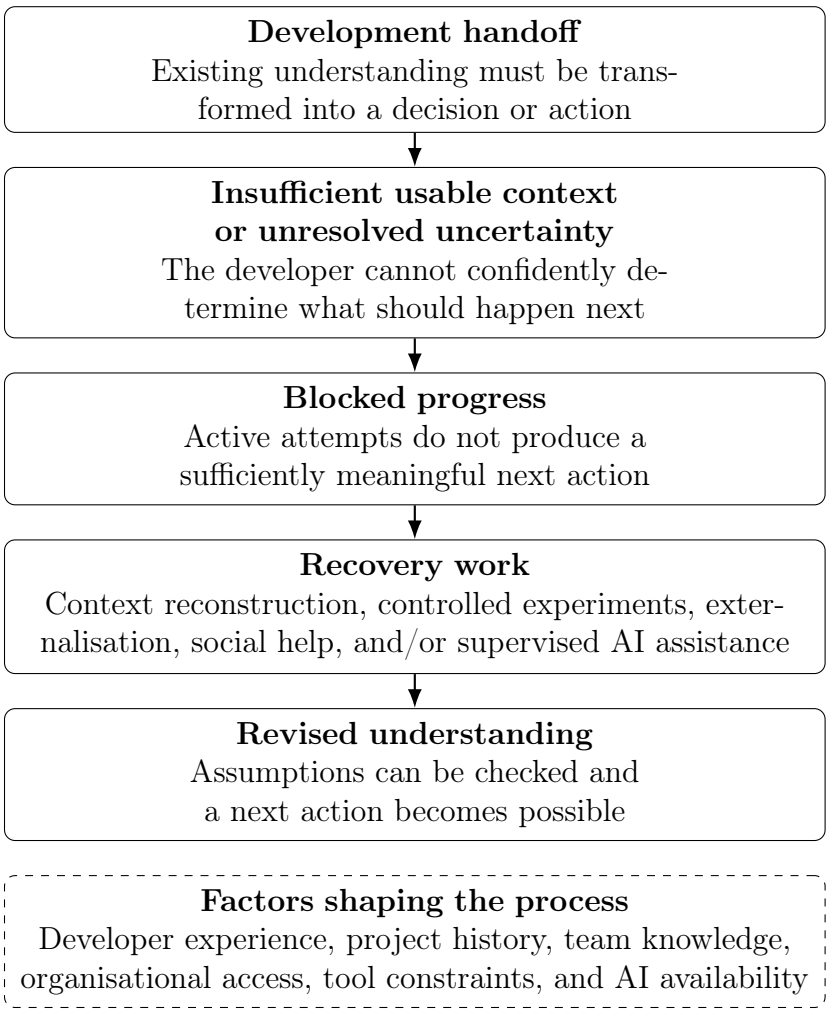

Figure 7.1.: Interpretive synthesis of Programmer's Block at development handoffs. The figure summarises relationships identified across the survey and interview findings and is not presented as a formal theory or causal model.

Construct boundaries revealed during recruitment. Informal reactions received while distributing the survey also illustrated that Programmer's Block is not interpreted uniformly by developers. These reactions were not collected or analysed as part of the formal dataset and are therefore treated only as reflections on the study's framing rather than as empirical findings.

One reaction questioned whether the survey placed too much emphasis on the individual developer, since software development usually takes place within teams. The response instead highlighted interruptions to flow, analysis paralysis, and repeated urges to redesign a solution. Another reaction questioned whether time spent discussing options, thinking about an architectural decision, or stepping away from the computer should be described as being blocked. A further response used the term

more broadly for situations in which code, tools, dependencies, and development environments repeatedly failed.

These reactions illustrate why the boundary of Programmer's Block matters. Deliberate reflection, discussion, incubation, or temporarily stepping away does not necessarily represent Programmer's Block when it contributes to progress. Similarly, a clear technical failure is not automatically a block when its cause and next corrective action are known. The concept used in this thesis is narrower: the relevant situation is one in which a capable and motivated developer remains unable to convert the available understanding into meaningful progress despite actively attempting to continue.

The reactions also challenged an initially individual-centred reading of the survey. Although the survey described movements between development activities, the interviews showed that these movements were often shaped by team members, business knowledge, documentation, organisational access, inherited systems, and technical dependencies. This reinforces the socio-technical interpretation developed above. Programmer's Block may be experienced by an individual developer, while the information required to resolve it may be distributed across the wider technical and organisational environment.

7.2. Implications for Practice

The findings suggest several practical ways in which developers, teams, and organisations may reduce the likelihood that a difficult development handoff develops into a longer blocking episode. Since this was an exploratory study, these implications should be understood as recommendations grounded in the participants' experiences rather than as interventions whose effectiveness has already been experimentally demonstrated.

First, teams should avoid treating every blocking situation as evidence that a developer lacks ability or effort. In many of the cases examined in this study, the difficulty was not readily explained by a lack of general programming ability. Participants often understood the general task but could not determine how to proceed with sufficient confidence. The missing element was frequently project-specific context, such as design rationale, expected system behaviour, historical changes, business rules, or knowledge held by an unavailable colleague. This supports earlier research showing that developers invest considerable effort in reconstructing implicit system knowledge and repeatedly seek information about behaviour, dependencies, design decisions, and task history [14, 13]. When a developer becomes blocked, a useful first response is therefore to examine whether the required context is accessible before assuming that the problem is individual performance.

Second, development handoffs should transfer reasoning as well as tasks. A ticket, review comment, or technical document may state what needs to be changed without explaining why the current system behaves as it does, which assumptions must remain valid, or which dependencies could be affected. Teams can reduce this loss of

context by recording the purpose of a change, important design decisions, expected and observed behaviour, known limitations, relevant components, and the people who can clarify unresolved questions. This does not require documenting every detail. The goal is to preserve the information another developer would need to construct a usable understanding of the change. This is particularly important when services are transferred between teams, original contributors leave, or a developer begins working with an unfamiliar part of the system.

Third, organisations should recognise investigation and externalisation as part of software development rather than as time spent before the "real" work begins. Participants moved forward by constructing test harnesses, creating temporary tables, logging payloads, comparing expected and actual results, isolating small reproducible cases, writing notes, and explaining the problem to another person. These activities created evidence that could be checked and reduced the amount of uncertainty the developer needed to hold mentally. Teams can support this process by providing suitable debugging access, allowing time for controlled experiments, encouraging developers to record unsuccessful attempts, and making it acceptable to involve another person before a block becomes prolonged.

Help-seeking is particularly relevant in this respect. Some participants described learning through experience that continuing alone for too long did not necessarily produce better solutions. Teams can make help-seeking easier by creating psychological safety around questions, pairing, and incomplete understanding. Rather than defining a strict universal time limit, teams may benefit from agreeing that repeated attempts without new evidence are a signal to change strategy or involve another person. This makes escalation part of normal problem solving rather than an admission of inability.

Finally, AI-based tools should be treated as supervised support rather than as an autonomous solution to Programmer's Block. The participants found AI useful for summarising code, explaining unfamiliar logic, generating alternatives, producing scaffolding, and supporting tests or documentation. This resembles the exploration mode described by Barke et al., in which a developer who is uncertain about the next step uses AI suggestions to investigate possible directions [3]. However, AI could not reliably supply missing business knowledge, organisational history, private project context, or accurate information about every current framework. Its output also created additional work because the developer still had to inspect, evaluate, modify, or reject the proposed solution [18].

For organisations adopting such tools, this means that access to AI alone should not be treated as a solution to developer blocking. AI assistance should be combined with human review, testing, appropriate privacy rules, and clear responsibility for the final change. It is also important to preserve access to human sources of project and business knowledge. An AI system may help a developer understand available information more efficiently, but it cannot reliably replace context that was never recorded or cannot be shared with it.

Overall, the practical implication is that Programmer's Block should be managed as a socio-technical problem. Supporting the individual developer is important, but

it is not sufficient when the difficulty is produced by unavailable context, inaccessible systems, incomplete handoffs, organisational restrictions, or uncertainty across technical boundaries. Teams may be better able to support recovery by preserving relevant reasoning during handoffs, making system behaviour observable, normalising timely help-seeking, and treating AI as one supervised source of support among several.

7.3. Implications for Future Research

The exploratory nature of this study leaves several opportunities for further research. A first priority concerns developer experience. The survey included respondents with different levels of programming experience, but the interview sample did not include participants who met the study's novice classification. Consequently, the experience-related interview findings rely partly on participants' retrospective descriptions of how they reacted earlier in their careers. Future studies should deliberately recruit developers who are currently at the beginning of their professional development and compare their blocking episodes with those of more experienced developers. Such work could examine whether differences actually occur in recognising a block, choosing recovery strategies, asking for help, evaluating uncertainty, and using AI-based support.

A second direction concerns the timing of data collection. Interviews provide rich descriptions but depend on participants remembering and reconstructing events after they occurred. Future work could collect evidence closer to the blocking episode through diary studies, experience-sampling methods, workplace observation, or recorded problem-solving sessions. These methods could help distinguish more precisely between a productive pause, ordinary debugging difficulty, and a sustained Programmer's Block. They could also reveal how a block develops over time and when developers decide to change strategy, seek assistance, or use an AI tool.

Future research could also examine the role of handoff quality directly. The present findings suggest that important difficulties arise when technical or organisational context cannot be converted into information that is usable for the next task. This could be tested through interventions that preserve design rationale, document expected behaviour, identify knowledgeable contacts, or make historical decisions easier to retrieve. Studies could investigate whether such practices reduce time spent reconstructing context or shorten blocking episodes when responsibility moves between developers, teams, or systems.

The role of AI also requires more focused investigation. The survey measured perceived helpfulness rather than objective effectiveness, and the interviews included different AI tools, models, and styles of use. Future studies could compare specific forms of AI support during blocking situations, such as code explanation, context summarisation, alternative generation, debugging assistance, or agentic implementation. In addition to perceived usefulness, such studies could measure time to regain progress, correctness of the eventual solution, amount of verification required,

cognitive effort, and whether the developer develops a better understanding of the underlying system. This would help determine whether AI actually resolves a block or simply changes the work required to move through it. Longitudinal research could additionally examine whether frequent AI-assisted recovery changes developers' independent problem-solving and recovery strategies, particularly in situations where AI support is unavailable or lacks the context needed to provide a useful solution.

Another important direction is the relationship between AI and missing context. The findings of this study suggest a possible distinction between information that exists but is difficult for the developer to process and information that is genuinely unavailable. AI may be particularly effective in the first situation, while providing limited benefit in the second. Future studies could examine this distinction explicitly by varying how much project, business, and historical context is available to the developer and to the AI system. Such research would also be relevant to organisations where privacy or security policies restrict the context that can be provided to external tools.

Finally, the construct of Programmer's Block itself would benefit from further empirical refinement. The informal reactions during recruitment and the interview accounts show that developers can use the term "blocked" to describe very different situations, including technical failures, productive deliberation, missing information, emotional frustration, and genuine inability to determine a next action. Future research could investigate these boundaries more directly and develop clearer criteria for distinguishing Programmer's Block from related forms of development difficulty. In particular, the role of socio-technical causes deserves further attention. A block may be experienced cognitively by one developer while its cause lies in missing organisational knowledge, unavailable people, inaccessible environments, or fragmented ownership.

The present study should therefore be viewed as a step toward a more detailed understanding of Programmer's Block rather than a final model of the phenomenon. Its main contribution is to connect blocking experiences with development handoffs, context reconstruction, recovery strategies, AI-assisted problem solving, and developer experience within the same empirical setting. Future work can test, refine, or challenge these relationships using larger, more targeted, and more immediate forms of data collection.

8. Threats to Validity

Because the main empirical contribution of this study is based on qualitative interview analysis, the trustworthiness of the findings is discussed using the framework proposed by Lincoln and Guba. The framework consists of credibility, transferability, dependability, and confirmability [17]. These criteria are used to discuss how well the interpretations are supported by the collected data and how transparently the research process was conducted.

8.1. Credibility

Credibility refers to whether the findings provide a trustworthy representation of the participants' experiences. Several steps were taken to improve credibility throughout this study.

First, data were collected using two complementary methods. The online survey provided descriptive information about blocking experiences relevant to Programmer's Block at development handoffs, while the semi-structured interviews allowed participants to describe individual episodes in substantially greater depth. Using the survey and interviews together provided both broad descriptive context and detailed accounts of how blocking situations developed.

Second, the interview questions were open-ended and encouraged participants to describe real situations from their own professional experience instead of answering predefined assumptions. Follow-up questions were used whenever additional clarification was needed.

A further credibility limitation concerns the alignment between reported blocking experiences and the specific construct of Programmer's Block. As described in Section 4.1, the working definition used in the survey was intentionally broader than the four-condition definition adopted in this thesis. The survey could not establish for every reported episode whether the required programming skills and infrastructure were available or whether all other conditions of Programmer's Block were satisfied. The survey findings should therefore be interpreted as reports of blocking experiences relevant to Programmer's Block rather than as independently verified instances of the construct.

The interviews provided substantially more contextual information, but even here not every reported episode aligned equally closely with the strict definition. For example, the case discussed for I8 also involved learning a new programming language and was therefore treated as a construct-boundary case rather than as an unambiguous instance of Programmer's Block. These distinctions reduce the risk of forcing

all experiences of being stuck into the same construct, but they also limit the extent to which every reported episode can be labelled definitively as Programmer's Block.

The interviews nevertheless relied on participants' own descriptions of past experiences. This is particularly relevant to RQ3 because participants were asked to reflect on how similar situations would have affected them earlier in their careers. These accounts provide useful insight into how participants understand the influence of experience, but they may be affected by recall and hindsight. Participants may no longer remember the exact order, duration, or emotional intensity of an earlier blocking episode. Their current knowledge may also make their earlier behaviour appear more deliberate or systematic than it was at the time.

Professional self-presentation may have influenced the accounts as well. Admitting panic, confusion, repeated unsuccessful attempts, or reluctance to ask for help can be uncomfortable for a professional developer. Participants may therefore have described some experiences in a less intense or more organised way. The retrospective statements are consequently treated as the participants' current interpretations of their earlier experiences rather than as direct observations of developers who were at an earlier stage of their programming careers.

Third, the interview data were analysed using thematic analysis supported by STGT-informed procedures of open coding, constant comparison, and memoing [5, 11]. Coding began with open codes that remained close to participants' accounts. Similar and contrasting codes were then compared across interviews and consolidated into a master codebook before candidate themes were developed. Candidate themes were repeatedly checked against coded extracts and complete transcripts, and contrasting and limiting cases were considered during theme refinement.

Despite these measures, the findings remain interpretive. Qualitative theme development involves analytical judgement, and another researcher could organise the same interview material into somewhat different categories or themes. The themes presented in this thesis should therefore be understood as a systematically developed and evidence-supported interpretation of the participants' accounts rather than as the only possible interpretation of the dataset.

8.2. Transferability

Transferability refers to the extent to which readers may judge the findings to be relevant in other contexts.

The participants represented variation in programming experience, technical backgrounds, development roles, technologies, and work contexts. They also described different forms of development work, including requirements-related decisions, implementation, debugging, maintenance, system integration, migration, and work with inherited systems. This variation allowed similar and contrasting patterns to be examined across different situations, but it does not establish that the findings will apply to all software-development settings.

However, the study does not aim to produce statistically generalisable results.

The survey participants and interviewees were recruited through voluntary participation and convenience sampling, which means that they may not represent all software developers. People who volunteered for an interview may also have been more interested in the topic or more comfortable discussing difficult programming experiences.

A particularly important limitation is that none of the nine interview participants met the study's definition of a novice developer. Although the recruitment process was open to different experience levels, the resulting interview sample consisted of intermediate and experienced developers. Consequently, RQ3 cannot be answered through a direct comparison between developers who are currently novices and developers who are currently experienced.

Information about earlier-career experiences was instead obtained by asking participants to reflect on how similar situations would have affected them when they had less experience. These reflections suggest possible differences in confidence, help-seeking, problem decomposition, and emotional responses. However, they cannot replace accounts from developers who are currently at the beginning of their careers. The findings may therefore underrepresent blocks related to onboarding, development-environment setup, unfamiliar tools, basic debugging strategies, uncertainty about team expectations, and deciding when to ask for help.

To allow readers to judge whether the findings are applicable to their own context, this thesis provides detailed descriptions of the research design, participant characteristics, interview procedure, and thematic analysis process.

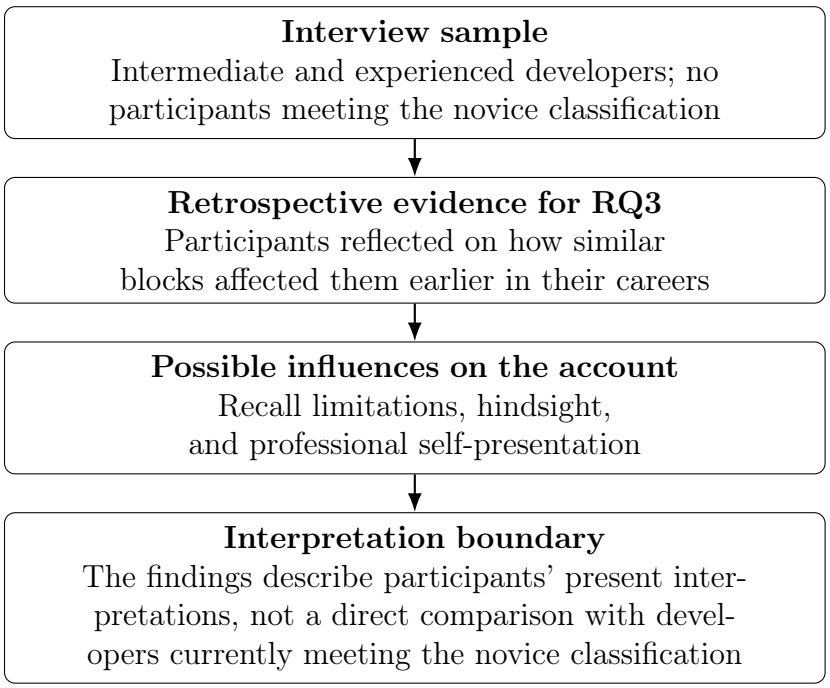

Figure 8.1.: Evidence boundary for interpreting experience-related differences. The figure summarises the sampling and retrospective self-report limitations and does not represent an additional empirical finding.

8.3. Dependability

Dependability concerns the consistency and transparency of the research process. To improve dependability, the study followed a documented sequence from survey data collection and interview recruitment through qualitative analysis. The survey was conducted before the interviews to provide descriptive context about blocking experiences at development handoffs and to support participant recruitment. Semi-structured interviews were subsequently conducted using the same core interview guide, while follow-up questions were adapted to the concrete situations described by individual participants.

During analysis, each interview was coded individually before codes were compared across participants. Similar codes were consolidated into a master codebook and organised into seven code families. Constant comparison, memoing, and a cross-interview evidence matrix were used to examine relationships and differences across interviews. Candidate themes were then repeatedly compared with coded extracts and complete transcripts and refined until they represented coherent analytical patterns across the dataset.

One procedural deviation occurred during transcription. Participants had initially been informed that Whisper would be used locally, but its transcription quality was not sufficient for the qualitative analysis. The workflow was therefore changed to

f4, as documented in Section 4.5. Although this represents a change from the originally planned procedure, documenting the change makes the final data-processing workflow transparent.

The coding process, analytical decisions, code consolidation, and theme development were documented throughout the analysis. These records provide an audit trail from individual interview material through the master codebook and cross-interview comparison to the final themes. Another researcher might make different analytical judgements, but the documented procedure makes the development of the reported findings traceable.

8.4. Confirmability

Confirmability refers to the extent to which the findings are grounded in the participants' accounts rather than the researcher's personal assumptions.

Several measures were taken to improve confirmability. The analysis remained closely connected to the interview data throughout the coding and theme development process. Candidate themes were retained only when they were supported by evidence from multiple interviews and contributed to answering one or more research questions. Cases that differed from the majority of participants were also considered during theme refinement instead of being ignored.

Representative quotations are included throughout the findings chapter to illustrate how the themes emerged from the participants' own experiences. The use of direct quotations allows readers to evaluate whether the interpretations presented in this thesis are supported by the underlying data.

Nevertheless, complete researcher neutrality is neither assumed nor achievable in qualitative analysis. Theme development requires interpretation when deciding which extracts are related, how codes should be consolidated, and which broader patterns are analytically meaningful. The purpose was therefore not to remove researcher interpretation, but to make it systematic and traceable. Open coding, constant comparison, memoing, consideration of contrasting cases, the cross-interview evidence matrix, and the use of participant quotations were used to keep the interpretation closely connected to the empirical material.

The qualitative analysis was conducted by a single researcher, and no independent second-coder comparison or analyst triangulation was performed. This means that the code consolidation and theme development reflect one researcher's analytical decisions. The use of an explicit codebook, evidence matrix, contrasting cases, and supporting quotations was intended to make these decisions visible and open to inspection rather than to claim complete independence from researcher judgement.

8.5. Summary

This study used several strategies to strengthen the trustworthiness of the findings, including complementary survey and interview data, a documented thematic analysis process supported by STGT-informed procedures, systematic coding, constant comparison, a master codebook, cross-interview comparison, consideration of contrasting cases, and participant quotations supporting the reported themes.

The findings nevertheless need to be interpreted within the boundaries of the study design. The survey used a broader working definition of being blocked than the strict definition of Programmer's Block and therefore provides descriptive evidence of relevant blocking experiences rather than independently verified cases of the construct. The interview sample was small and self-selected, and the interview accounts depended on participants' retrospective descriptions of events.

The interpretation of experience-related differences requires particular caution. No interview participant met the study's novice classification. Evidence concerning earlier-career behaviour therefore came mainly from intermediate and experienced participants reflecting retrospectively on how they had responded when they had less experience. These accounts may be influenced by recall, hindsight, and professional self-presentation. RQ3 consequently identifies perceived experience-related patterns within the available participant accounts rather than providing a direct comparison between developers who currently meet the novice and experienced classifications.

Rather than claiming universal conclusions, this thesis provides an exploratory, evidence-supported interpretation of how developers experience blocking situations at development handoffs, how they attempt to recover, and how AI-based support and programming experience interact with these situations.

9. Conclusion

This thesis investigated Programmer's Block at development handoffs, with particular attention to how developers experience and overcome blocking situations, how AI-based tools are used during these situations, and how developer experience influences the way blocks are perceived and handled. The study combined an exploratory survey of 85 respondents with nine semi-structured interviews. The survey provided descriptive context, while the interviews formed the main basis for understanding how blocking episodes developed and how participants attempted to recover from them.

The findings suggest that Programmer's Block at development handoffs cannot be understood simply as a situation in which a developer does not know how to write code. Across the interviews, participants often understood the overall task, the intended behaviour, or substantial parts of the system. The difficulty arose when this partial understanding could not be transformed into a sufficiently safe and meaningful next action. Development handoffs were particularly relevant because they required developers to transform one form of understanding into another: requirements into decisions, designs into implementations, observed failures into diagnoses, or diagnoses into changes. The survey provided descriptive support for this interpretation. Within the collected sample, the most frequently selected difficult handoffs were concentrated before or around the beginning of code writing, where requirements and design understanding had to be converted into implementation direction.

With respect to RQ1, the blocking episodes reported in the interviews occurred across conceptual planning, inherited-code comprehension, debugging, integration, migration, deployment, and task re-entry. Their causes were rarely isolated to one technical problem. Blocks were associated with fragmented or unavailable context, hidden system behaviour, complex interacting rules, dependencies on other people, unfamiliar technologies, organisational restrictions, and cognitive or emotional pressure. Recovery similarly involved more than writing additional code. Developers decomposed problems, exposed intermediate system states through logs and tests, reconstructed missing context, documented and explained their own reasoning, experimented with smaller cases, sought help from other people, and sometimes temporarily stepped away from the task. Although these strategies differed in form, many reduced uncertainty by making either the behaviour of the system or the developer's own reasoning easier to inspect.

For RQ2, AI-based tools were a common part of problem solving for many participants. The survey showed frequent use of AI during blocking situations, although perceived helpfulness was moderate rather than uniformly high. The interviews pro-

vided a more detailed explanation of this pattern. Participants used AI to explain unfamiliar code, summarise complex information, generate implementation alternatives, produce scaffolding, support debugging and testing, and reduce the amount of information that had to be maintained mentally. However, AI was most useful when relevant context could be supplied and when the developer had enough understanding to evaluate the generated result. Missing business knowledge, organisational history, privacy restrictions, installation-specific conditions, outdated information, and incorrect or unnecessarily complex generated solutions remained important limitations.

AI should therefore not be understood as simply removing Programmer's Block. Within the cases examined in this thesis, AI assistance appeared instead to redistribute some of the reasoning required to regain progress. AI could reduce the effort involved in producing a candidate solution, while increasing the importance of determining whether that solution was correct, appropriate, maintainable, and safe. The developer therefore remained responsible for judgement even when part of the implementation work was delegated to an AI system.

For RQ3, the findings suggest that programming experience does not make developers immune to Programmer's Block. Blocking situations occurred among participants with substantial professional experience, and the survey similarly showed recent blocking experiences among respondents with six or more years of programming experience. What appeared to change with experience was the response to the block. More experienced participants described a broader repertoire of recovery strategies, earlier recognition of unproductive approaches, more deliberate problem decomposition, greater use of previous experience, and greater willingness to seek help when individual investigation stopped producing new information. Earlier-career reflections placed greater emphasis on self-doubt, panic, setup uncertainty, and delayed help-seeking.

This experience-related finding must nevertheless be interpreted cautiously. No interview participant met the study's novice classification. Comparisons involving earlier-career behaviour therefore relied partly on experienced participants' retrospective reflections rather than on a direct comparison between developers currently meeting the novice and experienced classifications. The results consequently support the interpretation that experience changes how developers perceive and recover from blocks, but they do not establish general differences between developers currently meeting those two classifications.

Taken together, the findings suggest that Programmer's Block at development handoffs can be understood as a socio-technical phenomenon. The block is experienced by an individual developer, but the conditions that create or resolve it may be distributed across source code, tools, documentation, technical environments, colleagues, organisational processes, business knowledge, and the developer's own cognitive state. A capable developer may therefore become blocked not because the ability to program has disappeared, but because the available understanding is insufficiently connected to the decision that must be made next. This perspective broadens Programmer's Block beyond the immediate act of code writing and places

greater emphasis on the context in which programming work is performed.

The practical implication is that reducing Programmer's Block cannot be treated solely as the responsibility of the individual developer. Teams can support recovery by preserving design rationale and relevant context during handoffs, making system behaviour observable, allowing time for structured investigation, and normalising timely help-seeking. AI can form part of this support, but it should be treated as a supervised source of explanation and possible solutions rather than as a replacement for developer understanding or responsibility.

Finally, increasingly capable AI-based development tools may change where Programmer's Block becomes visible. If AI systems perform more implementation, testing, debugging, and revision work, some code-writing blocks may become less prominent while difficulties shift toward specifying intent, supplying sufficient context, evaluating generated changes, integrating them into existing systems, and deciding when AI-generated work can be trusted. Future research should therefore examine not only whether AI helps developers overcome Programmer's Block, but whether increasingly autonomous assistance eliminates, shifts, or postpones the cognitive and socio-technical difficulties through which blocking emerges.

Overall, this thesis contributes an exploratory empirical account of Programmer's Block at development handoffs by connecting the phenomenon with context reconstruction, recovery strategies, developer experience, and AI-assisted problem solving. Its central conclusion is that progress becomes difficult when developers cannot turn the understanding available to them into a sufficiently justified next action. Understanding Programmer's Block in this way shifts attention from the question of why a capable developer is "not coding" toward the more useful question of what information, reasoning, technical visibility, or social support is missing for the developer to move forward.

Bibliography

[1] Anderson, N., Alakmeh, T., Jackson, V., Vaz Pereira, G., Akirmak, U., Estey, A., Prikladnicki, R., Fritz, T., van der Hoek, A., Storey, M.A.: ChatGPT: Friend or foe when comprehending and changing unfamiliar code. arXiv preprint arXiv:2605.10702 (2026)

[2] Baltes, S., Diehl, S.: Towards a theory of software development expertise. In: Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 187–200. ACM (2018)

[3] Barke, S., James, M.B., Polikarpova, N.: Grounded Copilot: How programmers interact with code-generating models. Proceedings of the ACM on Programming Languages 7(OOPSLA1), 85–111 (2023)

[4] Begel, A., Simon, B.: Novice software developers, all over again. In: Proceedings of the Fourth International Workshop on Computing Education Research. pp. 3–14. ACM (2008)

[5] Braun, V., Clarke, V.: Thematic Analysis: A Practical Guide. SAGE (2022)

[6] Chuang, Y.T., Chang, H.Y.: Analyzing novice and competent programmers' problem-solving behaviors using an automated evaluation system. Science of Computer Programming 237, 103138 (2024)

[7] Cruz, L.C., Sanchez, H., González, V.M., Robbes, R.: Work fragmentation in developer interaction data. Journal of Software: Evolution and Process 29(3), e1839 (2017)

[8] Dagenais, B., Ossher, H., Bellamy, R.K.E., Robillard, M.P., de Vries, J.P.: Moving into a new software project landscape. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering. vol. 1, pp. 275–284. ACM (2010)

[9] Daun, M., Brings, J., Obe, P.A., Stenkova, V.: Reliability of self-rated experience and confidence as predictors for students' performance in software engineering. Empirical Software Engineering 26, 80 (2021)

[10] dr. dresing & pehl GmbH: How f4 ensures GDPR-compliant automatic transcription. `https://www.audiotranskription.de/en/f4-features/f4-gdpr/` (2024), status: 10 September 2024; accessed: 11 August 2026

[11] Hoda, R.: Qualitative Research with Socio-Technical Grounded Theory: A Practical Guide to Qualitative Data Analysis and Theory Development in the Digital World. Springer (2024)

[12] Kelleher, C., Brachman, M.: A sensemaking analysis of API learning using React. Journal of Computer Languages 74, 101189 (2023)

[13] Ko, A.J., DeLine, R., Venolia, G.: Information needs in collocated software development teams. In: Proceedings of the 29th International Conference on Software Engineering. pp. 344–353. IEEE Computer Society (2007)

[14] LaToza, T.D., Venolia, G., DeLine, R.: Maintaining mental models: A study of developer work habits. In: Proceedings of the 28th International Conference on Software Engineering. pp. 492–501. ACM (2006)

[15] Lee, H.P., Sarkar, A., Tankelevitch, L., Drosos, I., Rintel, S., Banks, R., Wilson, N.: The impact of generative AI on critical thinking: Self-reported reductions in cognitive effort and confidence effects from a survey of knowledge workers. In: Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems. pp. 1–22. ACM (2025)

[16] Liang, J.T., Yang, C., Myers, B.A.: A large-scale survey on the usability of AI programming assistants: Successes and challenges. In: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. pp. 52:1–52:13. ACM (2024)

[17] Lincoln, Y.S., Guba, E.G.: Naturalistic Inquiry. Sage Publications, Beverly Hills, CA (1985)

[18] Mozannar, H., Bansal, G., Fourney, A., Horvitz, E.: Reading between the lines: Modeling user behavior and costs in AI-assisted programming. In: Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems. pp. 142:1–142:16. ACM (2024)

[19] Müller, S.C., Fritz, T.: Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress. In: Proceedings of the 37th IEEE/ACM International Conference on Software Engineering. vol. 1, pp. 688–699. IEEE (2015)

[20] Obi, I., Butler, J.L., Haniyur, S., Hassan, B., Storey, M.A., Murphy, B.: Identifying factors contributing to "bad days" for software developers: A mixed-methods study. In: Proceedings of the 47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice. pp. 1–11. IEEE (2025)

[21] Parnin, C., Rugaber, S.: Resumption strategies for interrupted programming tasks. Software Quality Journal 19(1), 5–34 (2011)

[22] Peng, S., Kalliamvakou, E., Cihon, P., Demirer, M.: The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv preprint arXiv:2302.06590 (2023)

[23] Robillard, M.P., DeLine, R.: A field study of API learning obstacles. Empirical Software Engineering 16(6), 703–732 (2011)

[24] Salleh, N., Hoda, R., Su, M.T., Kanij, T., Grundy, J.: Recruitment, engagement and feedback in empirical software engineering studies in industrial contexts. Information and Software Technology 98, 161–172 (2018)

[25] Schantong, B., Siegmund, N., Siegmund, J.: Toward a theory on programmer's block inspired by writer's block. Empirical Software Engineering 30(1), 15 (2025)

[26] Shihab, M.I.H., Hundhausen, C.D., Tariq, A., Haque, S., Qiao, Y., Mulanda, B.W.: The effects of GitHub Copilot on computing students' programming effectiveness, efficiency, and processes in brownfield coding tasks. In: Proceedings of the 2025 ACM Conference on International Computing Education Research. pp. 407–420. ACM (2025)

[27] Siegmund, J., Kästner, C., Liebig, J., Apel, S., Hanenberg, S.: Measuring and modeling programming experience. Empirical Software Engineering 19(5), 1299–1334 (2014)

[28] Sommerville, I.: Software Engineering. Pearson, 10 edn. (2016)

[29] Tang, N., Chen, M., Ning, Z., Bansal, A., Huang, Y., McMillan, C., Li, T.J.J.: Developer behaviors in validating and repairing LLM-generated code using IDE and eye tracking. In: 2024 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). pp. 40–46. IEEE (2024)

[30] Vaithilingam, P., Zhang, T., Glassman, E.L.: Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In: Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems. ACM (2022)

[31] Weber, M., Mailach, A., Apel, S., Siegmund, J., Dachselt, R., Siegmund, N.: Understanding debugging as episodes: A case study on performance bugs in configurable software systems. Proceedings of the ACM on Software Engineering 2(FSE), 1409–1431 (2025)

[32] Weisz, J.D., Kumar, S.V., Muller, M.J., Browne, K.E., Goldberg, A., Heintze, K.E., Bajpai, S.: Examining the use and impact of an AI code assistant on developer productivity and experience in the enterprise. In: Extended Abstracts of the 2025 CHI Conference on Human Factors in Computing Systems. pp. 673:1–673:13. ACM (2025)

A. Survey Instrument

The following reproduces the survey content presented to participants, including the participant information, response options, data-protection information, and interview-recruitment questions. Administrative elements of the LimeSurvey print export, such as timestamps, page URLs, and internal branching expressions, are omitted. Conditional question logic is stated in prose.

Programmer's Block at Development Transitions

A.1. Survey Introduction

Programmer's Block During Development Transitions

This short 4–5 minute survey supports a master's thesis at the Technical University of Chemnitz.

The goal of this study is to understand situations in which software developers feel "blocked" when transitioning between different development activities.

Understanding these situations can help researchers better understand developer productivity challenges and improve development processes.

A.2. Key Definitions

To ensure a shared understanding, the following terms are used in this survey:

Being blocked

Feeling stuck, uncertain, or unable to make meaningful progress on a programming task—even though you are actively trying.

Transition (handoff)

Moving from one development activity to another. For example:

- Understanding requirements → Designing a solution
- Designing → Writing code
- Writing code → Debugging or testing

A.3. Who Can Participate

Anyone with programming or software development experience is welcome to participate.

This includes:

- Professional developers
- Students with development experience
- Engineers working with software systems

A.4. Participation

Participation in this survey is voluntary.

- The survey takes approximately 4–5 minutes to complete.
- You may stop the survey at any time by closing your browser.
- There are 11 questions in total.

A.5. Data Protection Information

This survey is part of a research project on "Programmer's Block during Development Activity Transitions" conducted at the Technical University of Chemnitz, Professorship of Software Engineering.

The legal basis for processing data is your voluntary consent according to Article 6(1)(a) of the EU General Data Protection Regulation (GDPR). No personally identifiable information (such as name, address) will be collected. [1]

Data Collected

This survey collects only:

- Responses about your programming experience
- Responses about situations where you feel blocked during development tasks

No personally identifiable information (such as name, address) will be collected.

[1] This statement referred to the analytical survey responses. Respondents who volunteered for a follow-up interview could optionally provide an email address for recruitment and scheduling. Contact information was stored separately from the analytical survey data and was not included in the descriptive analysis.

A.6. Use of Data

Your responses will be used exclusively for scientific research purposes.
The collected data will:

- be analyzed as part of an academic master's thesis
- potentially contribute to scientific publications

A.7. Anonymity

This survey is anonymous.

- The record of your responses does not contain identifying information, unless you explicitly provide such information in a response.
- If you accessed the survey using an access code, the code is stored separately from survey responses and cannot be linked to your answers.

A.8. Data Storage

The collected data will be:

- securely stored
- accessed only by the researchers involved in this project
- deleted in accordance with academic research guidelines

A.9. Contact

If you have questions about the study or data protection, please contact:

Kunal Rokde
Master's Student
Technical University of Chemnitz
Email: `kunal.rokde@s2023.tu-chemnitz.de`

Belinda Schantong, M.Sc.
Technical University of Chemnitz – Professorship of Software Engineering
Email: `belinda.schantong@informatik.tu-chemnitz.de`

Prof. Dr.-Ing. Janet Siegmund
Technical University of Chemnitz – Professorship of Software Engineering
Email: `janet.siegmund@informatik.tu-chemnitz.de`

A.10. Section 1 — Background

Question 1

How many years of programming experience do you have (including study and personal projects)?

- Less than 1 year
- 1–2 years
- 3–5 years
- 6–10 years
- 10+ years

Question 2

Compared to other developers with similar backgrounds, how would you rate your programming expertise?

- Much less experienced
- Slightly less experienced
- About average
- Slightly more experienced
- Much more experienced

Question 3

How broad is your programming experience (in terms of projects, domains, or technologies)?

- Very narrow (mostly one type of project/technology)
- Somewhat narrow
- Moderately broad
- Broad
- Very broad

Broad experience refers to the variety of contexts in which you have programmed. This may include working across different types of projects (e.g., web, mobile, back-end, embedded), domains (e.g., finance, e-commerce, research), programming languages, frameworks, or system sizes.

For example, "very narrow" might mean most of your experience is within one type of project or technology stack. "Very broad" might mean you have worked across multiple domains, technologies, or project types.

A.11. Section 2 — Experiences of Being Blocked

Question 4

In the last month, have you felt blocked or unable to make progress at or due to a transition between development activities?

- Yes
- No
- Not sure

Question 5

Which transition is most difficult for you?
Transition (handoff)
Moving from one development activity to another.

- Understanding requirements → deciding what to build
- Deciding what to build → designing a solution
- Designing a solution → starting to code
- Writing code → testing/debugging
- Debugging → deciding on a fix
- Code review/integration (transitions around code review/merge conflicts)
- Other: _____

Question 6

When you feel blocked, how long does the block usually last?

- Less than 15 minutes
- 15–60 minutes
- 1–4 hours
- Around a day
- Several days or more
- Cannot say
- It depends on several factors

A.12. Section 3 — AI Tool Use

Question 7

When you feel blocked at such transitions, do you use AI-based tools (e.g., ChatGPT, Copilot) to help you move forward?

- Yes, often
- Yes, sometimes
- Rarely
- Never

Question 8

Overall, how helpful do you find AI tools for getting past such blocks?

- 1
- 2
- 3
- 4
- 5

1 = Not helpful at all, 5 = Very helpful.

A.13. Section 4 — Interview Participation

Participation in the follow-up interview is completely voluntary and not required to complete this survey.

The Interview will be based on your experience of being blocked.

If you choose to participate in a follow-up interview, you may be asked to provide your email address so that the researcher can contact you to schedule the interview.

Handling of Contact Information

If you provide your email address:

- It will be used only for the purpose of scheduling the follow-up interview.
- Your email address will be stored separately from your survey responses.
- The email address will not be linked to your survey answers.
- Only the researcher conducting the study will have access to this contact information.

Data Protection

After the interview scheduling process is completed and the interview has taken place (or if you withdraw), the contact information will be deleted.

Withdrawal

You may withdraw your interest in participating in the follow-up interview at any time by informing the researcher via email (`kunal.rokde@s2023.tu-chemnitz.de`). This will not affect your participation in the survey.

Question 9

Would you be willing to participate in a 20–25 minute interview about this topic?

- Yes
- Maybe
- No

The Interview will be based on your experience of being blocked.

Question 10

Is there any specific reason behind your answer of "Maybe"? (optional)
Displayed only when "Maybe" was selected in Question 9.

Please write your answer here:

Question 11

If yes or maybe, please leave your email address so we can contact you for interview scheduling:
If you want, you can also write us an email to schedule an interview at `kunal.rokde@s2023.tu-chemnitz.de`.
Displayed only when "Yes" or "Maybe" was selected in Question 9.

Please write your answer here:

Your email address will be deleted after data processing is complete.

Survey Completion Message

Thank you for completing this survey! Your responses are valuable for our research.

B. Interview Guide

B.1. Preamble

Thank you for participating in this research and for taking the time to speak with me.

My name is Kunal Rokde, and I am currently working on my master's thesis at TU Chemnitz. The topic of my thesis is Programmer's Block at Development Handoffs. Before we begin, I would like to provide some information about data protection, confidentiality, and your rights as a participant.

The interview will take approximately 20–25 minutes. Your participation is voluntary, and you are free to stop the interview at any point without any negative consequences. All responses will be treated confidentially and used only for academic research purposes.

With your permission, the interview will be audio recorded. The recording will be transcribed locally using Whisper in accordance with GDPR requirements.[1] Any mentions of names, affiliations, or other information that could reveal your identity will be removed from the transcript before analysis.

With your explicit permission, fully anonymized transcripts may be published as part of the master's thesis or future scientific publications. This permission is optional and can be revoked until July 2026.

If you are comfortable with continuing, I will start the audio recording and ask for your consent again so that it is documented on the recording.

B.2. Interview Questions

Q1. Warm-up: Participant Context

Q1a. Can you briefly describe the kind of software development work you usually do?

Q1b. How many years of programming experience do you have, including study and personal projects?

Q1c. Compared to other developers with similar backgrounds, how would you describe your programming expertise?

[1] This text reproduces the information provided to participants before the interviews. During data processing, the transcription workflow was subsequently changed from Whisper to f4 because the initial Whisper transcripts were not of sufficient quality. The change is documented in Section 4.5.

- Much less experienced
- Slightly less experienced
- About average
- Slightly more experienced
- Much more experienced

Q1d. How broad is your programming experience in terms of projects, domains, or technologies?

- Very narrow, mostly one type of project or technology
- Somewhat narrow
- Moderately broad
- Broad
- Very broad

Q2. Entry into the Experience

Can you describe a recent situation where you felt stuck or unable to make progress while working on a programming task?

Optional prompt, only if needed: Ideally, this should be something from the last few weeks.

Q3. Locating the Moment

At what point in your work did you feel stuck?

Optional follow-up, only if needed: Did this happen while moving from one development activity to another?

Q4. Understanding the Cause

Why do you think you got stuck at that point?

Optional follow-ups, only if needed:

- Was anything unclear to you at that moment?
- Did you feel that something was missing before you could continue?

Q5. Coping Strategies

What did you try in order to move forward?

Q6. AI Tools

Did you use any AI-based tools in that situation?

If yes: How did you use them, and in what way did they help or not help?

If no: Was there any reason you chose not to use AI tools in that situation?

Q7. Reflection on Experience Level

Now think back to earlier in your career, for example when you were still onboarding or had less experience. How do you think a situation like this would have affected you then?

How do you think situations like this will affect you in the future, once you have gained even more experience?

Q8. Wrap-up

Is there anything else about getting stuck during software development, or about how you deal with it, that you think is important but we have not discussed?

Do you have any questions?

Would you like to receive a short summary of the study results once the thesis is completed?

C. Master Codebook

This appendix reproduces the consolidated master codebook used during the qualitative analysis. It contains 62 master codes organised into seven code families. The code families were used to organise and compare coded material and should not be interpreted as the final themes presented in Chapter 6. The codebook retains the working term "development transition" in some labels; throughout the main thesis, the corresponding concept is referred to as a "development handoff".

Code families

Family	Codes	Main research use
Participant and experience context	5	Classifying participant role, breadth, and experience for RQ3.
Block location / development transition	11	Locating where the block occurs in the development process for RQ1.
Cause / condition of block	12	Explaining why the handoff becomes blocking for RQ1 and sometimes RQ2/RQ3.
Experience of block	6	Capturing how being blocked feels/appears in practice for RQ1/RQ3.
Coping / recovery strategy	12	Capturing how developers move forward for RQ1/RQ3.
AI use / perception	10	Capturing AI support, limitations, and risks for RQ2.
Experience difference	6	Comparing experience-related patterns and retrospective earlier-career accounts for RQ3.

Master codebook

ID	Master code	Definition / coding rule	RQ / source examples
Participant and experience context			
MC01	Experience level and career trajectory	**Definition:** Captures the participant's career stage, years of experience, and non-linear background where relevant. **Coding rule:** Use for self-reported years, early/mid/senior status, second-career path, academic-to-industry path, or long professional history. Do not use for a specific blocking cause unless the extract links experience to the block.	**RQ:** RQ3 **Examples:** I1: professional since 2010; I2: 21 years; I4: second-career ten-year developer; I7: teaching-to-development transition; I8: mid-level; I9: 10-15 years

ID	Master code	Definition / coding rule	RQ / source examples
MC02	Role/domain specialisation	**Definition:** Captures the participant's current development role and domain/stack focus. **Coding rule:** Use for backend, data engineering, equipment integration, enterprise maintenance, full-stack, research software, API/event-driven work. Do not use for general experience comparisons unless tied to RQ3.	**RQ:** RQ3 **Examples:** I1 enterprise maintenance; I5 data engineering; I6 backend Node.js; I8 backend Kotlin/Go; I9 backend APIs/event-driven
MC03	Breadth versus specialisation of technical exposure	**Definition:** Captures how broad or narrow the participant's project, technology, framework, or domain exposure is. **Coding rule:** Use when participants compare their breadth across languages, frameworks, domains, cloud/DevOps, or ecosystems. Exclude local task-specific unfamiliarity unless it is part of a block.	**RQ:** RQ3 **Examples:** I3 broad full-stack/infrastructure exposure; I4 commercial .NET plus private broadening; I5 Microsoft concentration; I9 broad non-specialist stack history
MC04	Developer identity and self-assessment	**Definition:** Captures how participants describe their own strengths, limits, and developer identity. **Coding rule:** Use for statements such as stronger at maintenance than greenfield, full-picture developer, close to senior, average in a senior team, or communication strength. Do not over-interpret as objective competence.	**RQ:** RQ3 **Examples:** I1 existing-codebase expertise; I2 experienced solo developer; I5 average within senior team; I8 communication/team facilitation strength; I9 full-picture developer identity
MC05	Role scope and lifecycle responsibility	**Definition:** Captures whether the participant works across many development phases or in a specialised role. **Coding rule:** Use when role scope shapes transition exposure, e.g., solo end-to-end work, role specialisation, or reduced transition salience. Exclude generic job descriptions that do not affect handoffs.	**RQ:** RQ1/RQ3 **Examples:** I1 role specialisation reduces transition burden; I2 solo end-to-end lifecycle responsibility; I4 integration engineer; I6 cross-role dependencies
Block location / development transition			
MC06	Requirements or context to implementation handoff	**Definition:** Block occurs when moving from a stated goal, requirement, or available context toward implementation action. **Coding rule:** Use when the goal may be known but the implementation path, missing business information, or required resources prevent progress. Exclude pure syntax learning unless it blocks this handoff.	**RQ:** RQ1 **Examples:** I2 requirements information gap; I4 requirement-to-implementation handoff; I6 missing requirements; I9 understanding rules to changing behaviour
MC07	Architecture/design to next actionable step	**Definition:** Block occurs when a broad design, rebuild goal, or architectural understanding cannot be translated into a concrete next task. **Coding rule:** Use for large-to-small decomposition difficulty, prioritisation paralysis, deciding what to build first, or uncertainty about architectural direction. Exclude debugging after code already exists.	**RQ:** RQ1/RQ3 **Examples:** I2 platform rebuild decomposition; I8 architecture-to-code/syntax bridge; I3 AI plan/architecture memory

ID	Master code	Definition / coding rule	RQ / source examples
MC08	Code comprehension to behaviour change	**Definition:** Block occurs when developers must understand existing code/rules before safely changing behaviour. **Coding rule:** Use for inherited services, migration rules, framework capabilities, legacy code, or large existing procedures. Exclude cases where the participant already understands the code and is only writing new code.	**RQ:** RQ1 **Examples:** I4 understanding framework capabilities before coding; I9 code comprehension to implementation; I1 legacy integration transition
MC09	Inherited artefact to modification	**Definition:** Block occurs when a developer receives legacy/inherited code, framework parts, or services from others and must modify or extend them. **Coding rule:** Use when original authors, history, or design rationale are unavailable. Do not use for completely self-authored code unless it has become unfamiliar over time.	**RQ:** RQ1/RQ3 **Examples:** I1 undocumented legacy changes; I4 inherited framework handoff; I9 inherited migration service responsibility
MC10	Cross-component or cross-system integration handoff	**Definition:** Block occurs when separately working components, layers, modules, or services must work together. **Coding rule:** Use for frontend-backend, module-to-module, API contracts, service integration, cross-layer payloads, or contract negotiation. Exclude isolated unit-level bugs.	**RQ:** RQ1 **Examples:** I7 frontend-backend integration and module integration; I1 legacy email/Graph integration; I6 third-party service/deployment integration
MC11	Failing output to root-cause diagnosis	**Definition:** Block occurs when moving from an error, incorrect output, or failing behaviour to identifying the root cause. **Coding rule:** Use for debugging, root-cause invisibility, duplicate outputs, hidden aggregation logic, or repeated investigation without diagnosis. Exclude simple known fixes.	**RQ:** RQ1 **Examples:** I5 SQL debugging transition; I7 structured debugging; I1 debugging blocked by environment
MC12	Migration or cross-ecosystem transition	**Definition:** Block occurs when moving functionality, data, or behaviour between products, languages, frameworks, versions, or ecosystems. **Coding rule:** Use for Python-to-Node, old product to new product, old APIs to newer APIs, Go-to-Python sidecar, package equivalence, and upgrade compatibility. Exclude routine refactoring inside one stable stack.	**RQ:** RQ1/RQ3 **Examples:** I1 Office 365/Graph compatibility; I3 Go to Python sidecar; I6 Python-to-Node migration; I9 old-to-new product migration
MC13	Development/testing to deployment or environment reproduction	**Definition:** Block occurs when implementation must be tested, deployed, or reproduced in an environment that differs from local development. **Coding rule:** Use for AWS deployment, Docker/Kubernetes setup, remote/local mismatch, locked-down client environment, local DB gaps, physical-device network paths. Exclude generic cloud experience statements.	**RQ:** RQ1 **Examples:** I1 restricted client environment/test harness; I6 AWS, Docker/Kubernetes, local DB; I7 physical-device LAN IP
MC14	Task switching and re-entry transition	**Definition:** Block occurs when moving between tasks, projects, meetings, or contexts disrupts cognitive continuity. **Coding rule:** Use for returning to a complex task after revenue work, meetings, interruptions, unreadable notes, or losing action items. Exclude planned breaks used for incubation.	**RQ:** RQ1 **Examples:** I2 project switching and re-entry burden; I9 meeting-switching cognitive residue; I1 instant messaging interruption risk

ID	Master code	Definition / coding rule	RQ / source examples
MC15	New tool, interface, language, or team onboarding handoff	**Definition:** Block occurs while entering an unfamiliar tool, interface, language, codebase, team, or technical workflow. **Coding rule:** Use for new-language onboarding, new-team contribution, unfamiliar query/logging tools, AI-tool adoption friction, framework unfamiliarity. Exclude broad experience profile unless tied to block.	**RQ:** RQ1/RQ3 **Examples:** I6 earlier framework/setup gaps; I7 fast-changing LangChain; I8 new-team/new-language/tool-interface handoffs
MC16	Review/explanation to validation handoff	**Definition:** Block or recovery occurs when explaining, reviewing, or validating a change exposes uncertainty or misunderstanding. **Coding rule:** Use when code review, explaining to another person, or reviewer context becomes relevant to correctness. Exclude general social help-seeking unless validation is central.	**RQ:** RQ1 **Examples:** I9 insufficient reviewer context/explaining reveals misunderstanding; I3 human-reviewed AI code; I1 rubber-duck explanation
Cause / condition of block			
MC17	Missing, insufficient, or unusable documentation	**Definition:** The block is caused or intensified by absent, outdated, incomplete, too-large, or unusable documentation. **Coding rule:** Use for missing package/framework docs, undocumented legacy changes, outdated official docs, or documentation overload. Distinguish from missing requirements when the missing item is technical documentation.	**RQ:** RQ1/RQ2 **Examples:** I1 undocumented legacy changes; I3 undocumented packages; I4 missing framework documentation; I7 incomplete/outdated docs; I9 documentation volume overload
MC18	Knowledge loss and unavailable authors	**Definition:** The block is caused by lost historical knowledge, departed contributors, unavailable original authors, or limited access to people who understand the system. **Coding rule:** Use when the participant cannot ask the person who created or knows the artefact. Exclude ordinary waiting for roles unless the issue is knowledge loss.	**RQ:** RQ1 **Examples:** I4 unavailable original authors; I9 former contributors unavailable; I1 previous staff changes not documented
MC19	Mental-model mismatch or overload	**Definition:** The participant cannot maintain or align a coherent understanding of system behaviour, rules, interactions, or assumptions. **Coding rule:** Use for mismatch between mental model and system demands, overload from many rules, integration assumptions, or losing earlier rules while reading later ones. Exclude lack of basic skill unless participant frames it as understanding the system.	**RQ:** RQ1/RQ3 **Examples:** I7 mental-model mismatch; I9 mental model overload; I5 complex procedure reasoning; I1 reconstructing context
MC20	Missing business or requirement information	**Definition:** The block is caused by absent, incomplete, unstable, or ambiguous business/functional requirements. **Coding rule:** Use for missing requirements, unclear client needs, insufficient initial information gathering, or business blockers. Exclude technical documentation gaps.	**RQ:** RQ1/RQ2 **Examples:** I2 requirements information gap/client uncertainty; I6 missing requirements/business blockage; I5 business-rule understanding
MC21	Cross-role or stakeholder dependency	**Definition:** Progress depends on another role, stakeholder, tester, lead, BA, product owner, client, reviewer, or team member. **Coding rule:** Use when waiting for people or layered stakeholder mediation blocks progress. Exclude social help-seeking as coping unless dependency causes the block.	**RQ:** RQ1 **Examples:** I2 stakeholder mediation; I6 waiting for other roles; I9 reviewer context limits; I1 BA collaboration as prevention

ID	Master code	Definition / coding rule	RQ / source examples
MC22	Library, package, or ecosystem incompatibility	**Definition:** The block is caused by packages, APIs, frameworks, SDKs, or libraries not mapping cleanly across ecosystems or changing unexpectedly. **Coding rule:** Use for deprecated/missing packages, dependency knowledge gaps, package restructuring, API/binary compatibility, JSON/CORS/security contract issues. Exclude general unfamiliarity unless the ecosystem itself creates the mismatch.	**RQ:** RQ1/RQ2 **Examples:** I1 API/binary compatibility; I3 dependency knowledge gap; I6 package equivalence uncertainty; I7 framework/package changes
MC23	Environment, access, or tooling constraint	**Definition:** Normal investigation or implementation is blocked by tooling, permissions, environment, setup, or infrastructure constraints. **Coding rule:** Use for restricted client environments, no debugger access, VPN/jump-server layers, local setup, cloud deployment, Docker/Kubernetes, physical-device network mismatch. Exclude deliberate tool choice.	**RQ:** RQ1 **Examples:** I1 restricted environment; I6 remote-to-local environment mismatch; I7 physical device path; I8 new tool/interface barriers
MC24	Speed of change and instability	**Definition:** The block is intensified by rapidly changing requirements, codebase, frameworks, tools, tickets, or documentation. **Coding rule:** Use when change outpaces the developer's ability to analyse or stabilise understanding. Exclude ordinary iteration unless it causes blocking.	**RQ:** RQ1/RQ2/RQ3 **Examples:** I8 requirements/codebase changing too quickly; I7 fast-changing frameworks; I3 AI speed can hide missed work
MC25	Complex business rules or conditional logic	**Definition:** The block is caused by dense rules, conditional branches, aggregation logic, migration merge rules, or domain-specific behaviour. **Coding rule:** Use for SQL stored procedures, payroll rules, migration rules, grouping/aggregation ambiguity, or complex data transformations. Exclude generic complex code without rule/domain context.	**RQ:** RQ1 **Examples:** I5 conditional branches and aggregation; I9 complex merging/migration rules; I6 expected-output migration
MC26	Responsibility pressure or solo ownership	**Definition:** The block is intensified by being the only/main responsible developer or by lack of technical peer support. **Coding rule:** Use for solo developer burden, one-person service ownership, insufficient reviewer context, high responsibility, or lack of internal peers. Exclude normal individual task ownership.	**RQ:** RQ1/RQ3 **Examples:** I2 solo developer context; I9 responsibility pressure; I4 inherited framework responsibility
MC27	Cognitive energy, fatigue, and attention limits	**Definition:** The block is shaped by tiredness, mental fatigue, limited focus windows, interruptions, or cognitive residue. **Coding rule:** Use for mental fatigue during debugging, few strong focus hours, afternoon frustration, meeting-switching residue, or interruptions. Do not reduce all blocks to fatigue if a technical/social cause is present.	**RQ:** RQ1/RQ3 **Examples:** I5 mental fatigue; I8 cognitive-energy limitation; I9 meeting residue; I1 instant messaging interruption risk
MC28	Affective resistance, self-doubt, or motivation drop	**Definition:** The block includes emotional resistance, self-doubt, frustration, panic, shame, or low motivation that affects progress. **Coding rule:** Use when participants explicitly describe affective impact. Keep distinct from causes such as missing context unless both are present. Avoid pathologising participants.	**RQ:** RQ1/RQ3 **Examples:** I1 tedious task motivation drop; I4 confidence threat; I8 frustration/burnout; I9 self-doubt, shame, early-career panic

ID	Master code	Definition / coding rule	RQ / source examples
Experience of block			
MC29	No clear next action	**Definition:** The lived experience of being capable and trying, but unable to select or start the next meaningful action. **Coding rule:** Use for screen-staring, 'what do I do next?', prioritisation paralysis, or inability to proceed despite effort. Exclude ordinary planning without distress or delay.	**RQ:** RQ1 **Examples:** I2 screen-staring uncertainty; I4 stuck on how not what; I9 overthinking change location
MC30	Circular or unproductive investigation loop	**Definition:** The participant repeats the same action or tries many things without meaningful progress. **Coding rule:** Use for repeatedly reading code, trying everything without understanding, or unproductive loops. Exclude systematic iteration with evidence tracking.	**RQ:** RQ1/RQ3 **Examples:** I5 repeated code-reading loop; I9 early-career trying everything without understanding; I1 younger programmers trying without notes
MC31	Uncertainty about change location or correctness	**Definition:** The participant is blocked by not knowing where to change code, whether the change is correct, or whether something obvious is missing. **Coding rule:** Use for change-location overthinking, root-cause invisibility, reviewer uncertainty, or correctness anxiety. Exclude documented design decisions.	**RQ:** RQ1 **Examples:** I5 root-cause invisibility; I9 overthinking change location/insufficient reviewer context; I2 architecture direction uncertainty
MC32	Confidence threat and shame in being stuck	**Definition:** The participant describes being stuck as damaging confidence, causing shame, panic, or fear of appearing unqualified. **Coding rule:** Use for early-career panic, qualification self-questioning, reluctance to ask, or self-doubt. Exclude normal uncertainty without confidence impact.	**RQ:** RQ3 **Examples:** I4 qualification self-questioning; I9 early-career panic/shame; I8 earlier-career burnout
MC33	How-not-what blockage	**Definition:** The task goal is known, but the route, method, or implementation path is unclear. **Coding rule:** Use when participant explicitly distinguishes knowing what to achieve from not knowing how to achieve it. Exclude missing requirements where the goal is not known.	**RQ:** RQ1 **Examples:** I4 requirements clear but implementation unclear; I6 business blockage distinct from technical; I9 understanding rules before changing behaviour
MC34	Delay, lost momentum, or stale work	**Definition:** The block leads to noticeable delay, stale tickets, postponed tasks, or reduced contribution momentum. **Coding rule:** Use for hours/days stuck, ticket staleness, putting tasks on hold, or re-entry delays. Exclude short planned pauses.	**RQ:** RQ1/RQ3 **Examples:** I5 hours spent stuck; I6 task put on hold; I8 ticket staleness; I2 rebuild progress interrupted
Coping / recovery strategy			
MC35	Step away, incubate, or reset perspective	**Definition:** The participant deliberately pauses, walks away, starts afresh, or returns with a fresh perspective. **Coding rule:** Use for walks, tea, bike rides, putting aside, starting afresh, taking a step back, or reviewing later when sharper. Exclude avoidance without return to the problem.	**RQ:** RQ1 **Examples:** I1 incubation/walks; I5 start afresh/fresh perspective; I8 deferred review when sharper

ID	Master code	Definition / coding rule	RQ / source examples
MC36	Decompose into smaller units or tasks	**Definition:** The participant breaks a large, opaque, or overwhelming problem into smaller pieces. **Coding rule:** Use for bite-size tasks, section-by-section debugging, stored-procedure sections, smallest reproducible unit, or task listing. Exclude random trial-and-error.	**RQ:** RQ1/RQ3 **Examples:** I2 task listing/decomposition; I4 bite-size pieces; I5 smaller sections; I7 smallest reproducible unit
MC37	Make invisible behaviour observable	**Definition:** Recovery occurs by creating observability into system behaviour, intermediate states, payloads, outputs, or rules. **Coding rule:** Use for logging, temporary tables, expected-versus-actual notes, rule tables, test harnesses, payload comparison, environment reproduction. Exclude general reading.	**RQ:** RQ1/RQ2 **Examples:** I1 local test harness; I5 temporary tables/expected-actual; I7 log payloads; I9 AI-generated rule table
MC38	Reconstruct missing context	**Definition:** The participant rebuilds, documents, simulates, or locally reproduces context needed to reason about the block. **Coding rule:** Use when recovery depends on reproducing production, self-documenting inherited code, using exact DLLs/shims, or building local setup. Exclude documentation as general prevention unless it reconstructs context.	**RQ:** RQ1 **Examples:** I1 production-like test harness; I4 self-documenting framework; I6 local environment setup; I9 written summaries
MC39	Notes and evidence preservation	**Definition:** The participant writes notes, preserves attempts, records expected/actual outputs, or externalises thinking in text. **Coding rule:** Use for notes as thinking tool, evidence preservation, expected-versus-actual lists, unreadable notes as negative case. Exclude formal documentation unless it captures analysis evidence.	**RQ:** RQ1/RQ3 **Examples:** I1 note-taking/evidence; I5 expected-actual notes; I8 notes as thinking tool; I9 inadequate notes under pressure
MC40	Explain, rubber-duck, or reframe the problem	**Definition:** The participant explains the problem to a person/object, simplifies it for another audience, or reframes it to expose assumptions. **Coding rule:** Use for rubber-ducking, explaining to reviewers, asking children/simple-perspective questions, or translating problem into English/pseudocode. Exclude ordinary status updates.	**RQ:** RQ1/RQ3 **Examples:** I1 rubber-duck/pseudocode; I4 children/simple questions; I9 explaining reveals misunderstanding
MC41	Social help-seeking and collaborative sense-making	**Definition:** The participant asks colleagues, seniors, team leads, peers, or external developers to reason through the block. **Coding rule:** Use for whiteboard collaboration, coworker idea-seeking, asking seniors, pair discussion, external developer discussion. Exclude stakeholder dependency unless the actor provides missing business information.	**RQ:** RQ1/RQ3 **Examples:** I1 whiteboard collaboration; I2 external developer discussion/body doubling; I4 coworker idea-seeking; I9 ask after short timebox
MC42	Escalate with explicit blockage and missing-information request	**Definition:** The participant communicates why the task is blocked and what specific information is needed from another role. **Coding rule:** Use for contacting leads/BAs/owners, requesting endpoints/URLs/tech-stack details, or explaining blocked status. Exclude vague asking for help.	**RQ:** RQ1 **Examples:** I6 escalate to lead/BA/owner; I6 explain why blocked; I6 demand knowledge transfer

ID	Master code	Definition / coding rule	RQ / source examples
MC43	Documentation as recovery and prevention	**Definition:** Documentation is created, improved, requested, or used to prevent/recover from future blocks. **Coding rule:** Use for self-documenting inherited code, preserving design knowledge, knowledge transfer expectations, documentation emphasis. Distinguish from MC17 where documentation absence is the cause.	**RQ:** RQ1/RQ3 **Examples:** I4 documentation as prevention/recovery; I6 knowledge transfer; I1 evidence/documentation; I9 written AI summary supports repetition
MC44	Structured debugging and controlled experiments	**Definition:** The participant applies systematic debugging, targeted tests, validation steps, or small experiments to narrow the issue. **Coding rule:** Use for line-by-line comparison, DTO validation, quick experiments, test-driven expected-output strategy, checking environment variables/types. Exclude random fixes.	**RQ:** RQ1/RQ2/RQ3 **Examples:** I5 line-by-line/temporary-table debugging; I6 test-driven expected-output; I7 structured debugging; I1 test harness
MC45	Prioritisation, process control, and change management	**Definition:** The participant manages blocks through prioritising, formal process, scope control, time-blocking, or change-request rules. **Coding rule:** Use for task priority setting, change-request process, time-blocking, stakeholder pushback, or MVP strategy. Exclude code-level decomposition unless process control is central.	**RQ:** RQ1/RQ3 **Examples:** I1 time-blocking/MVP; I2 priority setting/change requests; I2 stakeholder pushback; I6 putting task on hold
MC46	Pragmatic reuse and risk reduction	**Definition:** Experience-based strategy to reduce unnecessary complexity, avoid risky rewrites, buy/reuse reliable libraries, or narrow low-value edge cases. **Coding rule:** Use for buy rather than build, maintenance caution against rewrite, business-pragmatic edge-case decisions. Exclude AI-generated reuse unless human pragmatic decision is central.	**RQ:** RQ1/RQ3 **Examples:** I1 buy rather than build; I1 maintenance caution; I1 business-pragmatic scope decisions
AI use / perception			
MC47	AI for comprehension, summarisation, and rule externalisation	**Definition:** AI helps make code, rules, or system behaviour easier to understand by summarising or restructuring information. **Coding rule:** Use for code summaries, rule tables, line-by-line explanations, business-rule explanations, architecture memory. Exclude AI writing new code unless comprehension is the main function.	**RQ:** RQ2 **Examples:** I1 code exploration; I5 business-rule understanding; I8 architecture memory; I9 summarises code/rule table
MC48	AI for boilerplate, scaffolding, and repetitive implementation	**Definition:** AI helps create routine code, tests, schemas, types, documentation, or repetitive changes. **Coding rule:** Use for boilerplate, junior-level tasks, unit tests, types/structs, repetitive test updates, scaffolding. Exclude strategic planning unless explicitly AI-supported.	**RQ:** RQ2 **Examples:** I1 unit tests; I2 boilerplate/junior tasks; I3 scaffolding/types; I9 repetitive tests/documentation

ID	Master code	Definition / coding rule	RQ / source examples
MC49	AI for debugging, alternatives, and diagnostic planning	**Definition:** AI supports problem solving by suggesting approaches, identifying relevant code regions, or proposing debugging paths. **Coding rule:** Use for Copilot solution options, aggregation suggestions, debug planning, edge-case checking, test-driven prompts. Exclude direct implementation-only AI use.	**RQ:** RQ2 **Examples:** I3 edge-case checking; I5 aggregation changes; I6 alternative solutions/test-driven strategy; I7 guiding structure
MC50	AI for planning, brainstorming, and architecture support	**Definition:** AI is used or considered for higher-level planning, feature design, architecture memory, or brainstorming. **Coding rule:** Use for plan mode, feature design, architecture memory, or explicit distrust of AI for planning. Keep positive and negative perceptions under this code if planning is the object.	**RQ:** RQ2 **Examples:** I2 AI not trusted for planning/design; I3 AI plan mode; I7 AI across lifecycle; I8 architectural guidance to AI
MC51	AI as pre-escalation filter	**Definition:** AI is used before asking people, as an initial support layer to clarify or test possible solutions. **Coding rule:** Use when participant explicitly tries AI before team escalation. Exclude AI used after social help or as routine coding unless sequence matters.	**RQ:** RQ2/RQ1 **Examples:** I6 uses AI before team escalation; I3 reference checking with AI; I8 agent iteration before review
MC52	Human-in-the-loop AI supervision	**Definition:** The developer remains responsible for prompting, reviewing, critiquing, verifying, and deciding whether AI output is acceptable. **Coding rule:** Use for explicit prompting, review loops, human judgement, checking maintainability/readability, selecting best approach, not blindly trusting AI. Exclude unsupervised AI generation.	**RQ:** RQ2/RQ3 **Examples:** I3 AI output verification; I6 select best approach; I8 iterative AI review loop; I9 AI as pair partner
MC53	AI reduces cognitive load or working-memory burden	**Definition:** AI helps by reducing low-level memory demands, externalising rules, allowing deferred review, or supporting focus. **Coding rule:** Use when participant says AI reduces variables in head, summarises for repeated reading, bridges architecture-syntax gap, or handles low-value repetition. Exclude general productivity claims.	**RQ:** RQ2/RQ3 **Examples:** I8 reduces mental variable tracking/deferred review; I9 written AI summary supports repetition; I5 identifies relevant SQL logic
MC54	AI limitation: missing context, privacy, business information, or recency	**Definition:** AI is limited because it lacks project-specific context, cannot access private code, cannot provide missing business knowledge, or is outdated. **Coding rule:** Use for privacy/data governance, business blockers, model recency, fast-changing frameworks, hallucination/context concerns. Exclude AI risks that occur after adoption unless about limitation of support.	**RQ:** RQ2 **Examples:** I1 context-specific production problem; I2 recency/hallucination; I4 privacy rules; I6 business blockage; I7 AI not up to date
MC55	AI risk: hidden bugs, security, over-trust, skill erosion, or supervision load	**Definition:** AI may create new risks or downstream blocks even while helping in the moment. **Coding rule:** Use for hidden missed work, security injection, bugs, hallucination/sycophancy, skill erosion, agent orchestration limits, undervaluing development. Exclude simple tool limitation unless risk is explicit.	**RQ:** RQ2/RQ3 **Examples:** I2 sycophancy/hallucination; I3 security/hidden bugs; I8 skill erosion/agent limit; I9 privacy transcription risk

ID	Master code	Definition / coding rule	RQ / source examples
MC56	Variation in AI adoption and trust	**Definition:** Captures differences in how centrally or cautiously participants use AI depending on experience, workplace rules, or technical context. **Coding rule:** Use to compare AI-heavy, cautious, sceptical, constrained, or slow-adoption cases. Exclude specific AI function if another AI code is more precise.	**RQ:** RQ2/RQ3 **Examples:** I1 selective AI; I2 sceptical planning stance; I3 AI-heavy intermediate; I4 constrained by privacy; I8 AI for almost all current code; I9 slow adoption
Experience difference			
MC57	Early-career vulnerability to confidence loss	**Definition:** Less experienced or earlier-career versions of participants are described as more affected emotionally by blocks. **Coding rule:** Use for panic, shame, confidence damage, reluctance to ask, questioning qualification, burnout from forcing progress. Exclude current self-doubt unless explicitly linked to experience.	**RQ:** RQ3 **Examples:** I4 confidence threat for less experienced self; I8 earlier-career burnout; I9 early-career panic/shame
MC58	Experience changes coping but does not eliminate blocks	**Definition:** Blocks still occur with experience, but experience changes duration, framing, coping, escalation, or diagnosis speed. **Coding rule:** Use when participants say blocks remain but are handled differently/faster. Exclude statements implying experience prevents all blocks.	**RQ:** RQ3 **Examples:** I1 experienced still blocked; I2 experience does not remove blocks; I5 future experience reduces time; I9 current help-seeking
MC59	Experience builds procedural repertoire and scenario recognition	**Definition:** Experience provides strategies, pattern memory, debugging procedure, previous-case comparison, or broader scenario repertoire. **Coding rule:** Use for knowing how to decompose, where to look, what to request, which prior cases apply, or how to debug systematically. Exclude general seniority without process knowledge.	**RQ:** RQ3/RQ1 **Examples:** I4 past-case comparison; I5 decomposition skill; I6 migration procedure memory; I7 resilience/system thinking
MC60	Experienced blocks become higher-level socio-technical blocks	**Definition:** Experienced participants are less blocked by syntax/basic coding and more by architecture, inherited context, stakeholders, integration, responsibility, or organisational constraints. **Coding rule:** Use for cross-interview comparison where experienced developers face system-level, process-level, or context-level block. Exclude individual novice tool-learning cases.	**RQ:** RQ3/RQ1 **Examples:** I1 legacy/tooling context; I2 architecture/stakeholder uncertainty; I4 inherited framework; I9 responsibility/knowledge loss
MC61	Intermediate or AI-native workflow as contrast case	**Definition:** Intermediate/mid-level participants may use AI more centrally to bridge gaps or gain independence. **Coding rule:** Use for comparing AI-heavy participants with more cautious senior participants. Do not use to claim all intermediates are AI-native.	**RQ:** RQ3/RQ2 **Examples:** I3 AI-heavy intermediate; I8 mid-level AI-native contrast; I6 regular Copilot before escalation
MC62	Psychological safety enables faster help-seeking	**Definition:** A team culture that accepts questions reduces shame and allows earlier escalation. **Coding rule:** Use when the social environment makes it easier to ask, admit uncertainty, or recover from blocks. Exclude generic teamwork without safety dimension.	**RQ:** RQ3/RQ1 **Examples:** I9 team psychological safety/current shameless help-seeking; I1 collaborative whiteboards; I2 body doubling/social support

Source coverage

Interview	Main block / case used in codebook	Key contribution
I1	Legacy integration/upgrade in restricted enterprise environment	Experienced maintenance context, test harness/context reconstruction, AI limits for situated problems.
I2	Ground-up platform rebuild and decomposition/prioritisation block	Experienced solo developer, high-level handoff block, AI mistrust for planning.
I3	Undocumented package/SDK use and workaround architecture	Intermediate AI-heavy workflow, AI validation, risks of hidden bugs/security issues.
I4	Inherited undocumented framework did not work as advertised	Documentation/inherited-context block, privacy-constrained AI non-use, confidence impact for juniors.
I5	Dynamic SQL stored procedure debugging in payroll transformation	Root-cause diagnosis, decomposition/temporary tables, Copilot for business rules/aggregation.
I6	Missing requirements/resources and Python-to-Node migration/deployment	Business versus technical blockers, AI as pre-escalation filter, ecosystem compatibility not skill issue.
I7	Frontend-backend and module integration/refactoring blocks	Mental-model mismatch, API contract negotiation, AI limitations in fast-changing frameworks.
I8	New team/new Kotlin stack and tool-interface onboarding	Mid-level AI-native contrast, AI bridges architecture-syntax gap, skill erosion concerns.
I9	Inherited migration service with complex merge rules	Mental-model overload, AI rule externalisation, psychological safety and faster help-seeking.

Declaration of generative AI usage

AI tool	Purpose of use	Description of use	Thesis sections
ChatGPT (OpenAI)	**Language and academic writing support**	Used to improve grammar, readability, academic phrasing, conciseness, and consistency of text drafted for the thesis. Suggested revisions were reviewed and adapted before inclusion.	Throughout the thesis report
ChatGPT (OpenAI)	**Structure and argumentation feedback**	Used as a feedback tool to review chapter organisation, logical flow, repetition, alignment between research questions and chapters, and clarity of arguments. Final structure and content decisions were made by the author.	Chapters 1–9
ChatGPT (OpenAI)	**Literature-support and reference checking**	Used to help identify search terms, summarise selected literature, compare papers, and check whether statements were adequately supported by cited sources. Relevant publications and claims were checked against the original sources before inclusion.	Chapters 2 and 3; references throughout the thesis
ChatGPT (OpenAI)	**Research-instrument refinement**	Used to review and refine the wording, ordering, and clarity of survey and semi-structured interview questions.	Chapter 4; Appendices A and B
ChatGPT (OpenAI)	**Descriptive-analysis and consistency checking**	Used to assist with checking calculations, percentages, table presentation, and consistency of reported survey values. Final values were verified against the study data before inclusion.	Chapter 5

ChatGPT (OpenAI)	**LaTeX and document-formatting support**	Used to assist with LaTeX syntax, tables, figure captions, appendix formatting, cross-references, and troubleshooting document-layout issues.	Throughout the thesis and Appendices
ChatGPT (OpenAI)	**Final quality assurance**	Used to identify inconsistencies in terminology, numbers, quotations, references, spelling, and cross-chapter claims. Suggestions were manually reviewed and corrections were applied where appropriate.	Final thesis as a whole