

**An evaluative case study of the program debugging behaviour of the
paired Software Development Technician Apprentice in a
geographically distributed environment.**

Olajide Olayemi Jolugbo, BSc (Hons), MSc

March 2025

This thesis is submitted in partial fulfilment of the requirements for
the degree of Doctor of Philosophy

Department of Educational Research

Lancaster University

UK

Declaration

I declare that this thesis is my own original work and has not been submitted for a degree at this or any other university. The thesis does not exceed the 50,000-word limit, including footnotes and text in diagrams, tables, or figures, but excluding the title page, contents, abstract, appendices, and references. The word count is 58,608.

Signed: _____

Acknowledgements

I extend my deepest gratitude to my supervisor, Prof. Don Passey, for his unparalleled support and patience, without which I could not have completed this thesis. His expertise, exceptional guidance, and empathy offered stability throughout the difficult period. I also owe special thanks to Mrs. Alice Jesmont, TEL Programme Administrator, for her consistent support and timely reminders, which were essential to the success of the program.

Also, I deeply appreciate my family for their support and understanding. My daughters have been my source of joy and motivation for demonstrating exceptional understanding and being my pillar of support during this programme. My partner has been an unwavering source of strength and encouragement. Her support and faith in me never wavered, providing me with consistent motivation. I am grateful to the organisations and volunteers who contributed to my study. The research significantly benefited from the essential contributions of these individuals.

Lastly, I dedicate this work to my loving parents, who have been my moral and emotional anchor. Losing my father during this PhD was one of the most challenging experiences of my life. This thesis is in his honour; he continually supported my efforts to finish this program. His memory and spirit remain my constant inspiration. For this demanding and enriching journey, I am most grateful to God for His help. His support was crucial to my resilience and success. In addition, I am deeply grateful to everyone who contributed to this journey. Your backing has led me to a major accomplishment and fostered both my personal and career development.

An evaluative case study of the program debugging behaviour of the paired Software Development Technician Apprentice in a geographically distributed environment.

Olajide Olayemi Jolugbo, BSc (Hons), MSc

Abstract

This empirical study investigates the collective efforts of paired novice programmers working on rectifying Python code using technology-mediated tools. It aims to uncover: 1) the types of errors they encountered; 2) the debugging strategies and tactics employed by these apprentice pairs to locate and fix bugs within the Python code; 3) insights into how the pairs share the cognitive load; 4) the influence and efficacy of technological tools in the debugging process; and 5) the challenges faced by the pairs while working remotely to identify and resolve bugs, along with the underlying reasons for these challenges. It is methodically qualitative in nature and adopts a multi-case approach to closely examine each instance in its real-life context, utilising various data collection methods such as in-depth interviews, participant observations, code analysis, and focus groups.

Furthermore, this study examines 15 dyads as they work collaboratively to debug Python code, showing the challenges they confront as well as their diverse debugging strategies and tactics. It also demonstrates the importance of integrating debugging tools, as well as how dyads strategically distribute cognitive tasks. By focusing on the relatively unexplored area of distributed pair debugging, this study offers a fresh perspective on collaborative problem-solving among novice programmers working in remote settings. It notably presents a conceptual framework for understanding dyad's debugging in disparate settings, contributing significantly to computing education and integrating technology into educational practices.

However, despite its contributions, the study acknowledges its limitations and suggests directions for further research to enhance the generalisability and applicability of its conclusions. Ultimately, this thesis advances our understanding of the debugging processes of paired novice programmers in remote settings, offering empirical insights and technical recommendations to improve computing education and practice.

Keywords: Distributed Pair Debugging, Debugging Strategies, Remote Collaboration, Cognitive Load Management, Computer Science Education Education.

Table of Contents

Chapter 1: Introduction	1
1.0 Introduction.....	1
1.1 Motivation for the Study	5
1.2 Research Background	9
1.2.1 SDT Apprenticeship	10
1.2.2 Novice Programmers.....	12
1.3 The Rationale for this Study	16
1.4 Aims and Objectives	18
1.5 Research Questions.....	19
1.6 Structure of the Thesis	21
Chapter 2: Literature Review.....	24
2.0 Introduction.....	24
2.1 Choice of a Review Type.....	29
2.2 Methods	29
2.2.1 Review question	30
2.2.2 Sampling	31
2.2.3 Critical appraisal (CA) of sample (data collection)	37
2.2.4 Data analysis (Data extraction and synthesis, and thematic analysis)	38
2.2.5 Presentation	42
2.3 Discussion	57
2.4 Summary.....	64
Chapter 3: Conceptual Framework	66
3.0 Introduction.....	66
3.1 Information Foraging Theory (IFT)	66
3.2 Distributed Cognition	67
3.3 Integration of IFT and Distributed Cognition	69
3.4. Critical Analysis of Distributed Pair Debugging Conceptual Framework	70
3.4.1 Layer 1: Debugging Environment Layer	73
3.4.2 Layer 2: Information Foraging Layer	74
3.4.3 Layer 3: Distributed Cognition	75
3.4.4 Layer 4: Innermost Circle: Cognitive Processes	76
3.4.5 Centre: The Debuggers.....	77

3.5 Deployment for data collection and data analysis.....	78
3.6 Summary.....	83
Chapter 4: Methodology.....	84
4.0 Introduction.....	84
4.1 Research Question.....	84
4.2 Context and Study Site	87
4.3 Philosophical Perspectives of this Study	89
4.3.1 Paradigm	90
4.3.2 Ontology.....	91
4.3.3 Epistemology.....	92
4.4 Methodological Framework.....	93
4.4.1 Case study design and rationale	94
4.4.2 Sampling.....	96
4.4.3 Participants.....	98
4.4.4 Data Analysis	104
4.5 Empirical Research Process	114
4.5.1 Step 1: Debugging sessions.....	117
4.5.2 Step 2: Analysis of recorded debugging session	121
4.5.3 Step 3: Interview sessions	122
4.5.4 Step 4: Analysis of the dyadic interview session.....	124
4.5.5 Step 5: Focus group session	125
4.5.6 Step 6: Analysis of recorded focus group session.....	126
4.5.7 Limitation of the Chosen Methodology	127
4.6 Reliability and Validity	128
4.7 Ethical Issues and Concerns.....	131
4.8 Summary.....	133
Chapter 5: Findings	135
5.0 Introduction.....	135
5.1 Dyads Debugging Session Findings.....	135
5.1.1 Theme 1: Technology Utilisation.....	136
5.1.2 Theme 2: Debugging Strategies and Tactics	140
5.1.3 Theme 3: Error Spectrum	150
5.1.4 Theme 4: Cognitive Load Management	154
5.1.5 Theme 5: Challenges Faced	157
5.2 Python Code Analysis Findings	164

5.3 Interview Session Findings	169
5.3.1 Theme 1: Error Spectrum	169
5.3.2 Theme 2: Technical and Cognitive Skills	174
5.3.3 Theme 3: Challenges	182
5.4 Focus Group Discussion Findings	188
Theme 2: Technology's Role in Debugging Processes.....	190
5.4.1 Theme 1: Nature and Handling of Debugging Errors.....	190
5.4.2 Theme 2: Technology's Role in Debugging Processes	191
5.4.3 Theme 3: Strategies and Challenges in Debugging	193
5.5 Summary.....	196
Chapter 6: Conclusion	199
6.0 Introduction.....	199
6.1 Evaluation of Dyad's Case Studies	199
6.2 Research Questions.....	202
6.3 Refined Conceptual Framework Linking Research Outcomes to Distributed Debugging Processes	238
6.4 Novelty of this Work.....	243
6.5 Contributions.....	245
6.6 Significance of the Study	248
6.7 Trustworthiness of the Study	251
6.8 Limitations of the Study	253
6.9 Further Research and Recommendations.....	257
6.10 Conclusion.....	259
References.....	261
Appendix A: Participants Information Sheet - Apprentices	282
Appendix B: Participants Information Sheet - Work Based Mentors & Trainers	285
Appendix C: Participant Consent Form – Apprentices	287
Appendix D: Participant Consent Form – Work-Based Mentors & Trainers	288
Appendix E: Ethics Approval.....	290
Appendix F: The Bugged Python Code	291
Appendix G: Sample DYADs End of Session Codes	293
Appendix H: DYAD Interview Protocols	295
Appendix I: Focus Group Protocols	296
Appendix J: Sample Transcript of the Debugging Session.....	297

Appendix K: Sample Transcript of Dyad's Interview	299
Appendix L: Sample of Included Studies for the Critical Analysis.....	301
Appendix M: Debugging Session Codebook	304
Appendix N: DYAD Interview Codebook.....	305
Codes\\Interview\\Stage 1 & 2 - Familiarisation & Coding.....	305
Codes\\Interview\\Stage 3 - Theme Generation	313
Codes\\Interview\\Stage 4 - Theme Review.....	324
Codes\\Interview\\Stage 5 - Theme Definition	336
Appendix O: Focus Group Codebook	349
Codes\\Focus Group\\Stage 1 & 2 - Familiarisation & Coding	349
Codes\\Focus Group\\Stage 3 - Theme Generation	353
Codes\\Focus Group\\Stage 4 - Theme Review	358
Codes\\Focus Group\\Stage 5 - Theme Definition.....	364

List of Tables

<i>Table 1: Sample of a summary document for the critical analysis (CA) of selected studies.....</i>	38
<i>Table 2: Sample data extraction</i>	39
<i>Table 3: Sample data extraction (Continuation)</i>	40
<i>Table 4: Themes identified from data synthesis.....</i>	41
<i>Table 5: Strengths and weaknesses of the distributed pair debugging conceptual framework</i>	73
<i>Table 6: Relationship between the theoretical framework and the research methods (Layers 1 & 2).....</i>	79
<i>Table 7: Relationship between the theoretical framework and the research methods (Layers 3 & 4).....</i>	80
<i>Table 8: Relationship between the theoretical framework and the research methods (Centre Layer).....</i>	81
<i>Table 9: Participant details for the debugging sessions and the dyad's interview</i>	102
<i>Table 10: Evidence triangulation.</i>	106
<i>Table 11: List of bugs, bug type and difficulty level.</i>	120
<i>Table 12: Characteristics of the bugs' difficulty levels.....</i>	121
<i>Table 13: Interview protocol matrix adapted from Castillo-Montoya (2016).....</i>	124
<i>Table 14: Overview of key themes in dyads debugging sessions</i>	136
<i>Table 15: Technology Utilisation Subthemes in Dyadic Debugging Sessions</i>	136
<i>Table 16: Debugging Strategies & Tactics Themes in Dyadic Debugging Sessions.....</i>	140
<i>Table 17: Debugging Strategies & Tactics Themes in Dyadic Debugging Sessions (Continuation)</i>	141
<i>Table 18: Error Spectrum Subthemes in Dyadic Debugging Sessions.....</i>	151
<i>Table 19: Cognitive Load Management Subthemes in Dyadic Debugging Sessions</i>	154
<i>Table 20: Challenges Faced Subthemes in Dyadic Debugging Sessions</i>	157
<i>Table 21: Outline of the debugging sessions' core findings.....</i>	162
<i>Table 22: Outline of the debugging sessions' core findings (Continuation)</i>	163
<i>Table 23: Summary of bugs discovery, successful fixing and unsuccessful fixing.....</i>	167
<i>Table 24: Summary of specific syntax errors breakdown by discovery and resolution.</i>	168
<i>Table 25: Summary of specific logical errors breakdown by discovery and resolution.</i>	168

<i>Table 26: Summary of specific runtime errors breakdown by discovery and resolution</i>	168
<i>Table 27: Overview of key themes in interview sessions.....</i>	169
<i>Table 28: Error Spectrum Subthemes in Interview Sessions</i>	170
<i>Table 29: Technical and Cognitive Subthemes in Interview Sessions</i>	175
<i>Table 30: Challenges Subthemes in Interview Sessions.....</i>	182
<i>Table 31: Overview of key themes in Focus Group Sessions.....</i>	190
<i>Table 32: Overarching themes across the study</i>	198

List of Figures

<i>Figure 1: Integrative literature review (adapted from Lubbe et al., 2020)</i>	30
<i>Figure 2: Flow diagram for ILR (adapted from PRISMA – Preferred Reporting Items for Systematic Reviews & Meta-Analysis (Moher et al. (2009) cited in (Lubbe et al., 2020)).</i>	35
<i>Figure 3: AI tool connected papers</i>	36
<i>Figure 4: Distributed pair debugging conceptual framework</i>	71
<i>Figure 5: Participant demographic infographics recruited for the study.</i>	104
<i>Figure 6: The thematic analysis approach adapted from Braun and Clarke (2006).</i>	107
<i>Figure 7: Empirical research process.</i>	116
<i>Figure 8: Timeline of data collection and data analysis.</i>	117
<i>Figure 9: Debugging session research approach.</i>	118
<i>Figure 10: Python code seeded with syntax, logical and runtime bugs.</i>	119
<i>Figure 11: Refined Conceptual Framework Aligning Research Outcomes to Distributed Debugging Processes</i>	242

Chapter 1: Introduction

1.0 Introduction

Dating back to the era of Adam Smith, economists have recognised the substantial impact of a skilled workforce on an economy's productivity (Johnson, 1937). Such understanding brings to the fore the renewed focus on apprenticeships as a mechanism to bolster the upcoming workforce's skills (Guile & Young, 1998; Nash & Jones, 2013), although a study suggests that the primary positive impact on employment is achieved by retaining apprentices within the company where they received their training (Pierre & Jérémie, 2024).

Furthermore, as organisations strive to align their people, processes, and culture for long-term digital success (Kiron et al., 2016), digital transformation initiatives continue to drive up demand for talented software and technology workers. Against this backdrop, apprenticeships serve as a viable pathway for individuals to enter the labour market and contribute to the growing technology industry (Heyes, 2013; Hoeckel & Schwartz, 2010), and are proposed to be a promising answer to soaring youth unemployment (Steedman, 2012). In addition, England has been experiencing a digital skills crisis, with increasing demand from employers for skilled professionals to keep up with the ever-evolving technological landscape roles (Nania et al., 2019; Taylor-Smith et al., 2019). However, to effectively address unemployment and skill shortages in the software development sector, apprentices must acquire debugging abilities, which are an essential component of software development.

Moreover, the SDT apprenticeship standard (IfATE, 2024) is designed to offer specifics about what the apprentice will be doing and the abilities expected of them, allowing them to integrate into modern software development teams. These entry-level apprentices are also entrusted with developing a range of computer software and work in a variety of businesses, from huge enterprises to government organisations, and regularly contribute to multibillion-pound software solutions (Carter, 2015).

Therefore, the core of this role focuses on interpreting requirements, creating designs, and building and testing software solutions for bugs based on system specifications to achieve optimal results (IfATE, 2022). It is, however, crucial to highlight that software varies in size, complexity, and quality standards; even small applications are susceptible to defects. Therefore, regardless of the language used, bugs are an inevitable part of programming (Tsan et al., 2022). According to Lee et al. (2014), debugging is a significant, cognitively demanding process that is essential to the practice of programming rather than merely a supplementary activity.

Besides, Rich et al. (2019) contend that debugging is a distinct skill set that may be acquired outside of specific programming environments. Additionally, studies show that inexperienced programmers, such as those in the SDT apprentice category, frequently have trouble identifying bugs. According to Decasse and Emde (1988), this challenge stems from a lack of critical abilities required for bug isolation, understanding programming constructs, comprehending programme execution, and implementing efficient debugging procedures.

In the same vein, a plethora of studies have examined different aspects of debugging, including its challenges (Coker et al., 2019; Eisenstadt, 1993; Jeffries, 1982; Vessey, 1985), strategies (Katz & Anderson, 1987), and tools (Petrillo et al., 2019). Additionally, some research indicates that the most challenging part of debugging is identifying the bugs (Fitzgerald et al., 2008; Katz & Anderson, 1987). Successful completion of this stage typically results in the bugs being removed (Fitzgerald et al., 2010). However, more recent studies, such as those by Tsan et al. (2022), are increasingly focusing on how novice programmers approach debugging, which aligns with the focus of this investigation.

Nonetheless, some research suggests that collaborative interactions, such as pair debugging, can help mitigate the challenges associated with debugging (Jayathirtha et al., 2020; Murphy et al., 2010). In keeping up with this, modern software development practices often employ pair programming, in which two developers collaborate on a single code. This strategy minimises the cognitive load of an individual programmer (Kavitha & Ahmed, 2015) and reduces the potential for programming errors (Hannay et al., 2009).

Given the context of this study, pair debugging can be viewed as a subset of pair programming. Murphy et al. (2010) characterise it as “an important facet of pair programming” (p. 51), in which two developers collaborate to identify and rectify code issues while using a single computer. This implies that pair debugging involves the collaborative effort of two individuals, known as a dyad, often linked with the shared goal of debugging the code, with each providing their knowledge of the task. While this often

occurs with the dyad physically present in the same location, the concept also applies to distributed teams collaborating on the same programming code from separate locations.

With this in mind, the purpose of this study is to broaden the scope of pair debugging by examining its applicability in a distributed pair setting. It aims to investigate the transition of bug fixing from the traditional co-location setting to a distributed one, where debugging is heavily influenced by technology. Technology assumes a crucial role as a mediator in bug fixing, aiding in facilitating pair discourse and adopting debugging strategies. Simultaneously, the study intends to contribute to a better understanding of novice programmer debugging behaviours, particularly those of SDT apprentices, in distributed locations during collaborative pair debugging activities in the software development context.

The study specifically looks into the debugging strategies and tactics employed by geographically dispersed SDT trainees who collaborate to fix Python code issues utilising technology-mediated agents. The research does this by concentrating on a number of different areas, including compiler errors, verbal and non-verbal interactions between pairs, the roles of technology agents, the patterns of debugging activities, and how they resolve issues when they arise. Notably, while debugging research has been extensive, this study addresses a gap by specifically investigating distributed pair debugging in an educational context. While some research has centred on distributed pair programming, distributed pair debugging remains relatively unexplored, especially concerning debugging strategies within the educational context. This study aims to fill this gap and

contribute to understanding distributed pair debugging practices within educational settings.

1.1 Motivation for the Study

The motivation for this research stems from over 15 years of my professional experience within the apprenticeship system, particularly in digital education. In my roles as a director of training & assessment, curriculum manager, curriculum specialist, trainer, assessor, job coach, internal quality assurer, and end-point assessor, I have come to understand the paramount importance of practical, hands-on learning and skill development in apprenticeships. My involvement in educating apprentices on the standards for data technicians, data analysts, software development technicians, development and operations (DevOps) engineers, and network engineers at Levels 3 and 4 (Levels 3 and 4 are UK qualifications, with Level 3 akin to high school diplomas and Level 4 to first-year university studies) has exposed me to their challenges, especially in debugging, problem-solving, and collaborative programming. Over the years, I have worked closely with both apprentices and employers to ensure the effective delivery of digital learning programmes that align with the evolving needs of the industry (Fuller & Unwin, 2013; Lave & Wenger, 1991).

A major challenge I have observed is the difficulty apprentices experience with debugging, which is a critical aspect of software development. Debugging is an essential yet cognitively demanding process that often frustrates novice programmers. Research has shown that debugging requires a distinct set of skills, including isolating and identifying faults in code, an area where many apprentices struggle (Fitzgerald et al., 2008; Katz &

Anderson, 1987). In my professional experience, apprentices often struggle to understand programming constructs and apply debugging strategies effectively. These challenges are heightened in remote working environments, where limited access to peer support or mentorship exacerbates the difficulty. Hence, the motivation to explore distributed pair debugging arises from the need to address these challenges and improve apprentices' capacity to work collaboratively, even in geographically dispersed settings (Murphy et al., 2010).

This motivation is further informed by my role as an internal quality assurer and assessor, which has provided a unique perspective on the development of apprentices throughout their learning journeys. My experience highlights the importance of fostering practical problem-solving skills, especially in the context of distributed work environments. The shift towards remote work, accelerated by the COVID-19 pandemic, has underscored the need for new strategies for collaboration and skill development, particularly in the digital sector (Agerfalk et al., 2005; Espinosa et al., 2007). To adapt to these changes, apprenticeship programmes must integrate collaborative practices such as pair programming and debugging into their curricula, enabling apprentices to thrive in a world where remote work is becoming the norm (Cockburn & Williams, 2000).

Building on these insights, my professional background within the apprenticeship framework has significantly influenced my approach to this research. Through my active involvement in shaping, delivering, and assessing apprenticeship standards across various digital sectors, I have seen how apprenticeships are pivotal in equipping young professionals with essential workforce skills. However, I have also identified gaps in how

current frameworks prepare apprentices for the complexities of debugging in real-world scenarios, particularly in distributed environments. The apprenticeship model, traditionally rooted in hands-on learning, now faces the challenge of equipping apprentices to collaborate effectively across remote settings (Guile & Young, 1998). My insider knowledge of curriculum development has enabled me to critically assess how distributed pair debugging can bridge the gap between theory and practical application.

This study aligns with broader research, which suggests that debugging is often treated as a supplementary skill rather than a core component of the curriculum, leaving many apprentices underprepared for industry demands (Eisenstadt, 1993; Jeffries, 1982). From my perspective as a curriculum specialist, I have observed how this oversight limits the apprenticeship experience, preventing learners from acquiring the structured problem-solving skills necessary for industry success. By embedding systematic debugging strategies into the curriculum, apprentices could not only develop a more methodical approach to identifying and resolving software bugs but also enhance their technical competence, confidence, and workplace readiness. This integration would represent a critical step in modernising apprenticeship programmes to meet the demands of a rapidly evolving digital workforce.

The relevance of these changes is further underscored by the growing importance of remote work in software development. Studies have demonstrated that collaborative debugging practices, such as pair programming, lead to higher quality code, improved communication, and greater knowledge transfer (Hannay et al., 2009). However, the challenges of working in distributed teams, particularly for apprentices, remain

underexplored (Smite et al., 2021). This study seeks to address this gap by investigating how apprentices manage the cognitive load of debugging in distributed pair settings and how technology, such as integrated development environments, can facilitate this process (Beasley & Johnson, 2022). By doing so, it aims to provide actionable insights for designing apprenticeship programmes that equip learners with the skills and resilience needed for a remote-first workplace.

Building on these findings, my familiarity with apprenticeship standards, such as those established by the Institute for Apprenticeships and Technical Education, underscores the pressing need for these programmes to evolve in line with the demands of the digital workforce. While current standards emphasise knowledge, skills, and behaviours, the integration of collaborative problem-solving techniques, such as distributed pair debugging, remains underdeveloped. Through this research, I aim to bridge this gap by contributing insights that can inform the enhancement of apprenticeship standards, particularly in the digital sector, where collaboration and debugging are essential competencies (IfATE, 2022).

In conclusion, this research is deeply informed by my professional experience as a curriculum specialist and quality assurer and my understanding of the challenges apprentices face in developing collaborative debugging skills. As the demands of remote work reshape the modern workforce, it is critical that apprenticeship programmes adapt to equip learners with the skills needed to thrive in distributed environments. By investigating the debugging behaviours of SDT Apprentices in collaborative, remote settings using technology-mediated tools, this study will address a significant gap in the

literature. The findings aim to contribute to the advancement of apprenticeship programmes, ensuring they align more closely with the realities of the digital workforce and prepare apprentices for sustained success in their careers.

1.2 Research Background

Apprenticeship within the framework of collaborative learning represents a unique and effective approach to holistic skill development. In contrast to traditional classroom instruction, apprenticeships prioritise practical experience, allowing novices to acquire skills through direct observation and collaboration with experts who play an important role in imparting knowledge, skills, and guidance (Lave, 1995; Lave & Wenger, 1991), effectively embodying the “learning by doing” approach. This technique has received recognition for its effectiveness in traditional learning environments, notably in developing strategic and metacognitive skills (Sawyer, 2014) which are required for career success and modern workforce demands.

To add to that, apprenticeships in the modern period are great platforms for combining theory and practice (Mirza-Davies, 2015), allowing participants to use their newly gained skills in real-world scenarios (Wolter & Ryan, 2011). This suggestion is consistent with the arguments stated by Engeström et al. (2001), which emphasises the necessity of a balanced mix of theoretical learning and practical application. Similarly, apprenticeships, according to Lave and Wenger (1991), are intensive experiences that shape an individual's identity within a community of practice while also encouraging active learning and engagement with seasoned professionals. Thus, apprentices learn through active participation, progressing from peripheral to key members of their work communities

(Lave, 1996). In essence, this holistic approach teaches specific skills and prepares apprentices for meaningful workplace participation.

In relation to this research, the new apprenticeship standards replace the previous, more generalised framework-based apprenticeships and come with distinct characteristics such as Employer-Led Standards, Endpoint Assessment, and Funding Reforms (DfE/BIS, 2013). These new criteria were developed by groups of employers known as "trailblazers" and are intended to closely correlate with the specific skills, knowledge, and competencies required for each occupation (Fuller & Unwin, 2010, 2013). The associated Skills, Knowledge, and Behaviours (KSBS) were intended to make apprenticeships more relevant and challenging.

Not only that, the shift to employer-led standards serves a dual purpose: it fulfils both the skill demands of England's diverse sectors and the requirements of those seeking career advancement or a fresh start. As of August 2023, the standards have been customised to fifteen diverse industries, including agriculture, digital, and legal. This range ensures that a broad spectrum of skills and vocations is covered, reflecting the diverse demands of the national economy. Importantly, these apprenticeship standards are not static; they are constantly revised to reflect the ever-changing demands of the workforce and the economy.

1.2.1 SDT Apprenticeship

As of August 2023, the SDT Apprenticeship Standard in England, the standard that the SDT apprentices are learning, is one of 29 digital apprenticeship standards (IfATE, 2023)

developed and approved as a result of the Richard Review and following Trailblazer project. Following the Richard Review in 2012, the government introduced the Trailblazer programme, which enables employer-led groups to create new apprenticeship standards. The SDT Apprenticeship was one of the new standards (DfE/BIS, 2013; Richard, 2012), and it was intended to be more employer-led and occupation-focused.

The SDT Apprenticeship is a Level 3 qualification that normally lasts from 15 to 24 months. It addresses the fundamental skills, knowledge, and attitudes required to function as a software developer. Similarly, the SDT Apprenticeship standard is a customised occupational standard that specifies the fundamental 'knowledge, skills, and behaviours' (KSBs) required for proficiency in the job role. According to the Institute for Apprenticeships and Technical Education (IfATE), this standard has 62 KSBs divided into 25 knowledge, 32 skills, and 5 behavioural criteria (IfATE, 2024). These KSBs are further aligned with the 15 occupational duties that an apprentice is expected to perform. The standard also defines any qualifications required to complete the apprenticeship and how they connect with professional recognition, if applicable.

Also, apprentices are classified as novices (see Section 1.1.2) who have been trained to create, test, and maintain code, interpret design requirements, and communicate within a development team. They are generally responsible for assisting with software development throughout the whole software development life cycle. Given this, SDTs are entry-level team members who work in a variety of industries, ranging from financial services to public sector organisations. In addition, they are required to use basic debugging techniques as part of their occupational duties. Techniques include but are not

limited to interactive debugging, print debugging, and remote debugging. On top of that, the standard highlights the use of structured problem-solving methodologies, such as basic code debugging, in identifying and resolving issues (IfATE, 2024).

1.2.2 Novice Programmers

The gradual development of an individual's talents is described by the five-stage phenomenological model of skill acquisition proposed by Dreyfus and Dreyfus (2005). Although Dreyfus et al. (2000) classified these levels as novice, advanced beginning, competent, proficient, and expert. At the novice level, people use rules and drills to complete tasks without the benefit of real-world experience, frequently failing to grasp the context. The journey through these phases is a change from inexperience to mastery, where the novice is at the starting point and lacks perception and situational awareness.

In variance to Dreyfus and Dreyfus's suggestion, Shneiderman (1976) identified four levels of programming experience, namely naive, novice, intermediate, and advanced. Naive individuals are entirely new to programming; novices have completed an introductory course. Intermediates have finished two or three courses, while advanced programmers include graduate students, faculty, or professionals in the field. Based on Shneiderman's framework, the terms 'novice' and 'beginner' can be used interchangeably, and there is a suggestion that the first three categories might be too finely distinguished.

However, looking at the literature suggests a varied description of what constitutes a "novice". Allwood (1986) defines novice as either someone with minimal experience or someone completely new to programming, regardless of their actual knowledge. In

contrast, experts, on the other hand, effortlessly master tasks and respond intuitively to challenges (Dewey, 1922). Nonetheless, this thesis aligns with Dreyfus and Dreyfus (2005) in suggesting that novices rely on deliberative reasoning due to their limited contextual understanding, a limited repertoire of situational discrimination, and a detached comprehension of the phenomenon. This research reveals that while novices may have tacit knowledge that aids them in executing tasks, their reliance on stringent rules is evident, limiting their overall effectiveness.

In a related vein, Luxton-Reilly (2016) and Savage and Piwek (2019) argue that novice programmers lacking a solid understanding of core programming constructs like variables, arrays, recursion, and loops face challenges in crafting efficient functions and procedures. Similarly, Barbosa Rocha et al. (2022) emphasised novices' difficulty in properly integrating and implementing programming principles, which affected their abilities to develop and test code. Likewise, Klahr and Carver (1988) and Liu et al. (2017) argue that newbie programmers struggle because they lack particular domain knowledge and problem-solving skills.

Building on this, scholars such as Ahn et al. (2022), Denny et al. (2022), and Hassan and Zilles (2022) appear to agree that novice programmers frequently struggle with programming, albeit with differing perspectives on the root causes and potential solutions (Karvelas, 2019; Malik et al., 2022; Smith & Rixner, 2019; Tsan et al., 2019; Whalley et al., 2021).

Furthermore, other studies contend that novices, unlike experts, have fragmented knowledge structures, resulting in a shallow grasp of tasks and more frequent errors (Allwood, 1986). This is consistent with Jenkins (2002), who indicates that novices frequently underestimate the complexities of programming structures. Bonar and Soloway (1983) investigated whether rookie issues resulted from the nature of programming and the tools utilised. After closely observing and interviewing a subset of novice Pascal programmers, they inferred that using natural language influenced their early programming efforts. The incorrect utilisation of natural language strategies in programming seemed to be the root cause of their challenges. This conclusion echoes Soloway et al. (1981), who argued that traditional programming languages do not align well with the intuitive cognitive strategies used by novices familiar with natural language, causing discrepancies and misunderstandings.

Elaborating on this, a collection of research papers edited by Soloway and Spohrer on novice programmers reveals that their understanding often transcends mere rule memorisation but remains at a superficial level, focusing on line-by-line coding rather than grasping meaningful program structures (Soloway & Spohrer, 2013). Winslow (1996) deepens the understanding of this phenomenon by further noting that while novices might understand individual syntax and semantics, they struggle to combine them into coherent programs. This is echoed by Blackwell et al. (2002) and Lahtinen et al. (2005), highlighting novices' difficulties with programming constructs like loops, conditional statements, pointers, and recursion (Pane & Myers, 1996; Soloway & Spohrer, 1989). Du Boulay (1986) contends that insufficient domain comprehension and syntactic and semantic limitations are the primary causes of beginners' compounded problems. This is

supported by Pennington (1987), who adds that a shaky grasp of programming structures exacerbates their challenges.

Continuing from this, many studies pinpoint learners' attributes as the root cause of the challenges novices face (Guzdial, 1994; Lahtinen et al., 2005; McCracken et al., 2001; Robins et al., 2003; Soloway & Spohrer, 1989; Winslow, 1996). Buttressing this point, McCracken et al. (2001) address the challenges novice programmers face and emphasise common struggles such as understanding fundamental concepts and applying problem-solving techniques; this is further exacerbated by their shaky grasp of core programming concepts and inconsistent methodologies. Guzdial (1994), in his study, asserts the difficulty in grasping abstract notions and bridging the gap between theory and practice. Following that, Robins et al. (2003) draw attention to the challenge as the cognitive strain associated with problem-solving and comprehending programming principles. In their study, Soloway and Spohrer (1989) discuss frequent misunderstandings and mistakes, such as misinterpreting loops and conditionals. According to Lahtinen et al. (2005), other problems include a lack of past knowledge and overwhelming programming environments. Winslow (1996) in his study reiterates the huge difference between novices and experts, notably in algorithmic thinking and task management complexity. Adelson and Soloway (1985) and Mayer (1981) both affirm the critical role of a well-formed mental model in programming. The latter also points out issues like problem decomposition and syntax arising from immature mental models. Overall, these investigations demonstrate the layered and interconnected obstacles that novice programmers confront. However, reflecting on my personal experiences with teaching novices, these observations resonate.

Moreover, many novices perceive programming as daunting (Hanks et al., 2004; Jenkins, 2002; Robins et al., 2003). Jenkins (2002) attributes this to the extensive skill set needed while Sloane and Linn (1988) describe it as a layered skill acquisition, from basics to complexity. Jenkins (2002) and McKeithen et al. (1981) both present programming as a phased approach, transitioning from specifications to algorithms and, ultimately, to code. Other studies on novice programming have found that they struggle with fundamental concepts such as variables, control structures, and problem-solving strategies (de Raadt, 2007; Glezou & Grigoriadou, 2010; Hooper & Thomas, 1990; Lister et al., 2004; Papadakis & Orfanakis, 2018; Sajaniemi & Kuittinen, 2008; Van Someren, 1990). Novices often struggle with the complexities of language syntax and semantics (Robins et al., 2003), demonstrate alternative concept comprehension, and face difficulties in planning, writing, and debugging programs (Lister et al., 2004), owing to their limited understanding of programming (Kurniawan et al., 2019; Müller et al., 2019; Teague & Roe, 2007). While these struggles are well-documented, there is a noticeable gap in exploring the unique challenges faced by paired novice programmers debugging in a distributed environment (Kurniawan et al., 2019), suggesting a need for deeper investigation in this niche area.

1.3 The Rationale for this Study

The increasing prevalence of remote work has highlighted the need for insights into collaborative activities such as paired debugging, particularly in geographically dispersed contexts. While existing research has highlighted the benefits of paired programming in terms of code quality, improves team communication, fosters knowledge transfer, self-

efficacy, expertise sharing, and team collaboration (Bipp et al., 2008; Cockburn & Williams, 2000; Hughes et al., 2020), the specific challenges of working in geographically dispersed teams have been underemphasised (Jayathirtha et al., 2020; Murphy et al., 2010).

Moreover, while some research on distributed software development predates COVID-19, the practice has gained momentum since the early 2000s, with the COVID-19 pandemic further accelerating the shift to remote work (Agerfalk et al., 2005; Espinosa et al., 2007; Lacave & Molina, 2021; Miller et al., 2021; Neto et al., 2020; Sokolic, 2022). Thus, understanding how apprentices' debugging behaviours adapt in remote settings offers insights into apprentices' strategies and collaboration techniques during debugging (Adeliyi et al., 2021; Ying et al., 2021). Similarly, as remote work becomes more prevalent, it is crucial to understand the debugging strategies and technological usage of geographically distributed apprentices (Beasley & Johnson, 2022; Lynch et al., 2023; Smite et al., 2021). This research is both timely and crucial for sustaining software quality and efficiency.

Furthermore, the function of digital tools such as Integrated Development Environments (IDEs) in supporting the debugging practices of apprentices in remote paired programming scenarios (Hassan & Zilles, 2022) remains insufficiently examined. Considering the growing dependence on these tools in distributed settings, exploring their effectiveness and potential obstacles is important, thereby addressing a notable void in current studies.

1.4 Aims and Objectives

This empirical study examines the debugging strategies utilised by geographically dispersed Software Development Technician (SDT) apprentices debugging Python codes. The study zeroes in on their collective endeavours to rectify Python code using technology-mediated instruments. Additionally, it aims to uncover the factors influencing their debugging methods and identify their challenges.

To fulfil the aim of this investigation, the study will pursue the following objectives:

- Examine the types of errors made by geographically dispersed dyad apprentices when collaboratively identifying and correcting bugs in the provided Python code and errors they might miss or fail to rectify.
- To investigate the debugging strategies and tactics employed by these apprentice pairs in locating and fixing bugs in Python code and attempt to understand their problem-solving approaches.
- Gain insights into how geographically dispersed dyad apprentices share cognitive load during their bug detection and correction processes.
- Investigate the influence and efficacy of IDE tools and other technology-mediated aids in assisting or impeding the debugging tasks the geographically separated dyad apprentices tackled.
- Lastly, explore the challenges confronted by paired SDT apprentices working from different locations as they collaborate on bug identification and resolution and identify the underlying causes of these challenges.

By addressing these objectives, the study intends to provide an in-depth understanding of the synergies involved in remote, collaborative debugging among apprentice programmers.

1.5 Research Questions

Debugging is a costly and time-intensive task where programmers utilise a variety of tools and approaches for bug identification and resolution (Hirsch & Hofer, 2022). Although debugging is traditionally a solitary activity, paired debugging turns it into a collaborative effort that promotes engagement and accountability between pairs (Baker et al., 2004). This process necessitates a deep understanding of how they collaboratively reason through and resolve errors by investigating their mental models (Oman et al., 1989) and the ability to link observed behaviours to potential defects (Perscheid et al., 2017).

Current literature shows a paucity of research on paired debugging among novice programmers, especially in distributed settings using technology as a mediator. Thus, adapting industry practices for educational contexts is logical but needs exploration, particularly when pairing remote novice programmers. Therefore, this study aims to understand the debugging behaviours of SDT Apprentices in collaborative, remote settings.

To this end, this central question steers this study:

“How do the paired SDT in geographically distributed locations work collaboratively to fix Python programming bugs using the technology-mediated medium?”

Leedy and Ormrod (2021, p. 26) suggest that from a research design standpoint, the central research question can be broken down into several smaller, focused questions. By answering these sub-questions, researchers are better positioned to address the central question comprehensively. This approach allows for a more granular investigation of the subject matter and can lead to a fuller understanding of the studied issue. So, given this and in investigating this central question, this study proffers answers to the following five specific research questions:

- **RQ₁**: What bugs are generated by the paired geographically distributed SDT apprentices working collaboratively to solve a given problem using Python?
- **RQ₂**: What bug locating strategies and tactics are deployed by the paired geographically distributed SDT apprentices while attempting to fix defects in the given Python code? How do they go about finding the bugs in the program code?
- **RQ₃**: How do the paired geographically distributed SDT apprentices distribute cognitive load when resolving bugged code?
- **RQ₄**: How does leveraging IDE tools enhance the capabilities of distributed pair debugging and mitigate the challenges encountered in debugging programs?
- **RQ₅**: What challenges are experienced by paired geographically distributed SDT apprentices working collaboratively on debugging programming bugs, and why are they facing such challenges?

Based on Maxwell (2012), the study employs multiple research questions to provide a focused framework for investigating the debugging behaviour of remote, dyad SDT apprentices. The first question aims to identify the types of bugs these apprentices generate while working on Python code. These data are then compared with bugs

generated by solo and co-located novice programmers in previous studies (Miller et al., 2021; Neufeld & Fang, 2005; Ralph et al., 2020). This comparison helps us understand if remote settings influence bug generation.

The second question focuses on the debugging behaviour of these apprentices, aiming to inform workplace mentors and training providers on how best to support them. The third research question uses the vocalised thoughts of apprentices to understand how they distribute the cognitive load during debugging, leveraging the theories of distributed cognition (Hutchins, 1995) and information foraging (Pirolli & Card, 1999). These theories provide a framework for data collection and analysis, offering insights into thought processes and debugging techniques. The fourth question explores the role of technology, including integrated development environments and collaboration tools, in the debugging process. The question asks whether technology enhances or complicates remote debugging, contrasting with its role in co-located settings. Lastly, the fifth research question investigates the challenges remote dyad apprentices face in debugging Python code and seeks to identify the root causes of these challenges.

However, it is essential to note that the value of this research is premised on understanding how dyad SDT apprentices approach debugging.

1.6 Structure of the Thesis

The structure of this thesis facilitates the exploration of the debugging strategies of novice programmers, with a specific focus on distributed pair debugging, the role of

technology, and the conceptual framework guiding the study. Each chapter serves a distinct purpose and contributes to the overall research endeavour.

Chapter 1 - Introduction: This chapter introduces the research topic, explaining the rationale behind the study and its primary research question. It also offers background information on the apprenticeship system in England and describes the characteristics of novice programmers. The study's goals and objectives are presented, and the chapter concludes with an overview of the thesis format.

Chapter 2—Literature Review: This chapter critically reviews research on novice programmers' debugging strategies and errors. It defines key debugging concepts, analyses current knowledge, and identifies common debugging strategies and challenges. It then inquires about distributed pair debugging, explores the role of technology, and reviews relevant research. The chapter establishes a theoretical basis for the study by addressing gaps in existing literature that the thesis aims to fill.

Chapter 3 - Conceptual Framework: This chapter presents the conceptual framework that serves as the foundation for the study, detailing essential ideas and relationships that drive data collection and interpretation.

Chapter 4 - Research Methodology: This chapter provides an in-depth discussion of the research design, methods, and tools used in the study. It describes the approach to data collection, participant selection, and data analysis, detailing the steps taken to ensure the validity and reliability of results. The chapter also discusses ethical considerations

associated with the research, highlighting measures to protect participants' rights and ensure research integrity.

Chapter 5 - Data Analysis and Findings: This chapter presents and analyses the data collected during the research within the conceptual framework and literature review context. It identifies patterns, trends, and insights emerging from the data, addressing the research question. The chapter provides a detailed interpretation of the results, discussing the implications of the findings for novice programmers' debugging strategies, especially within the apprenticeship model of learning.

Chapter 6 - Conclusion and Future Research: This chapter summarises the main findings and discusses their implications for novice programmers, educators, and software development teams. It outlines the study's contributions to the existing knowledge on debugging strategies among novices within distributed settings and apprenticeship learning, highlighting potential areas for future research. The chapter concludes by acknowledging research limitations and providing recommendations for practitioners and researchers in computer science education.

Chapter 2: Literature Review

2.0 Introduction

A plethora of studies, like those by Dyba and Dingsoyr (2008), Mens et al. (2019) and Nosek (1998), highlights substantial advancements in embracing collaborative practices in the realm of software development. These advancements have accelerated the adoption of pair programming as a pivotal educational strategy (Hanks et al., 2011; Sobral, 2020), gaining traction across diverse learning environments due to its inherent benefits (Baheti et al., 2002; Chorfi et al., 2020; da Silva Estacio & Prikladnicki, 2015; Dyba & Dingsoyr, 2008; Faja, 2014). Notably, such practices enhance the learning experience and aptly support learners in future workforce demands (National Research Council, 2013; Yett et al., 2020).

Building on this foundation, the traditional implementation of pair programming or, in its extended form, pair debugging, has typically been a co-located activity centred around continuous communication and collaboration (Smite et al., 2021). However, the COVID-19 pandemic has significantly heightened the use of digital technologies in coding education (Chorfi et al., 2020; Lacave & Molina, 2021), leading to the mainstreaming of virtual collaborative programming. In a way, this transition bolstered interest in its pedagogical advantages. As a result, educational institutions and training providers began adapting to industry-aligned collaborative models, integrating collaborative programming to provide learners with industry-relevant experiences, skills, and environments (Phillips et al., 2021; Smite et al., 2021). This practice reaffirms the

necessity and value of remote pair programming and debugging in contemporary education.

Given this growing emphasis on remote collaboration, this literature review examines the current state of research on distributed pair debugging, which remains a relatively under-explored area despite its increasing relevance in modern software development and education. Distributed pair debugging, also referred to in the literature as “virtual pair debugging” or “remote collaborative debugging” (Baheti et al., 2002; Hanks, 2008; Smite et al., 2021), involves two individuals collaborating remotely to identify and resolve programming errors. As distributed software development expands, understanding how debugging is conducted in these environments is critical. This review aims to identify current knowledge, gaps, and the specific strategies, tools, and challenges related to distributed pair debugging.

Building on this focus, previous studies have extensively explored pair programming, a practice where two programmers work together at one workstation, sharing the roles of driver and observer (Williams et al., 2000). Traditionally, pair programming has been a co-located activity, allowing for direct interaction and immediate feedback (Hanks et al., 2011; Nosek, 1998). However, with the rise of distributed software teams, particularly driven by the global shift towards remote work, there is increasing interest in how this co-located collaborative approach translates to remote settings, often termed “distributed pair programming” or “virtual pair programming” (Baheti et al., 2002; da Silva Estacio & Prikladnicki, 2015). Despite this interest, there is limited understanding of

how the practice adapts to the debugging phase, a process that demands effective collaboration and communication.

This gap becomes even more pronounced in distributed pair programming, which introduces unique challenges in remote settings. These include a heavy reliance on digital tools and the absence of face-to-face communication, both of which are critical to the debugging process (Olson & Olson, 2000; Smite et al., 2021). This literature review examines how such challenges influence debugging performance, particularly for novice programmers, who often find debugging difficult even in co-located environments. Additionally, the review evaluates whether existing tools for remote pair programming, such as shared integrated development environments (IDEs) and screen-sharing applications, provide adequate support for debugging in distributed contexts (Phillips et al., 2021; Tsai et al., 2015).

At its core, debugging is a cognitively demanding task that requires identifying, isolating, and correcting errors in code. For novice programmers, this process is further complicated by their limited experience and the high cognitive load it entails (Fitzgerald et al., 2008; Katz & Anderson, 1987). When conducted in a distributed setting, debugging becomes even more complex due to the separation between collaborators, which can cause delays in communication and misunderstandings about code functionality (Wetton, 2021). This review explores how these additional factors affect the debugging process and examines the strategies novice programmers employ to overcome them. By understanding these dynamics, the review aims to provide valuable insights into improving distributed pair debugging practices.

Effective collaboration in distributed pair debugging also depends on the availability and functionality of tools designed for real-time interaction. Integrated Development Environments (IDEs) with built-in debugging tools, screen-sharing applications, and real-time code editors have become central to distributed pair programming (Lacave & Molina, 2021; Smite et al., 2021). However, research suggests that these tools are often not fully optimised to meet the iterative demands of debugging, such as testing and error correction (Hanks, 2008; Tsai et al., 2015). This review evaluates the performance of these tools in distributed contexts, where delays in the feedback loop between driver and observer can hinder the debugging process. By assessing these tools' strengths and limitations, the review seeks to identify practical improvements that can enhance their effectiveness for distributed pair debugging.

Beyond the tools themselves, effective distributed pair debugging also relies on cognitive strategies employed by programmers. Research on pair debugging in co-located settings has shown that novices often depend on trial-and-error methods, which are inefficient and time-consuming (Khalid et al., 2021; Murphy et al., 2010). In distributed environments, where non-verbal feedback is limited and coordination occurs via digital platforms, these challenges can become even more pronounced (Khalid et al., 2021). This review examines the literature on cognitive load management in distributed pair debugging, focusing on how programmers allocate tasks, share information, and maintain effective collaboration despite physical separation. By understanding these strategies, the review aims to provide insights into improving both the technical and cognitive aspects of distributed pair debugging.

In addressing these complexities, this literature review synthesises findings from various studies to clarify the current state of research on distributed pair debugging. It maps the existing research landscape, identifies gaps, and proposes future directions for studying the strategies and tools that can enhance debugging performance in distributed settings. By doing so, this review contributes to developing best practices for novice programmers working in distributed environments, ensuring that they are equipped with the necessary skills and tools to debug effectively.

To conclude, it is essential to establish a methodological foundation to ensure the review's findings are grounded in rigorous academic practice. The following sections outline the approaches taken in conducting this literature review, beginning with the choice of review type in Section 2.1, which explains the rationale for selecting an integrative literature review. This method synthesises diverse streams of research, offering comprehensive insights into the field. Section 2.2 provides a detailed account of the methods used to gather and analyse relevant studies, drawing on established systematic review practices (Arksey & O'Malley, 2005; Randolph, 2019). In Sections 2.3 and 2.4, the focus shifts to the critical appraisal and synthesis of the literature, where key themes are identified using best practices in thematic analysis (Braun & Clarke, 2006) and integrative reviews to address complex and interdisciplinary research questions (Hopia et al., 2016). Together, these sections lay the groundwork for systematically reviewing the current state of knowledge in this under-explored field of distributed pair debugging, ensuring a robust and comprehensive analysis (Greenhalgh & Peacock, 2005; Hart, 1998).

2.1 Choice of a Review Type

Literature review methodologies vary widely in focus and application. Hart (1998) describes narrative reviews as broad overviews suitable for initial explorations but lacking detailed critique. Arksey and O'Malley (2005) focus on scoping reviews which map research landscapes broadly without the constraints of systematic reviews. Jesson et al. (2011) discuss critical reviews that analyse methodologies profoundly but are limited in scope, while Randolph (2019) describes state-of-the-art reviews that highlight recent innovations but may not cover the broader field. Integrative Literature Reviews (ILRs), detailed by Whittemore and Knafl (2005) and Torraco (2005), blend various research types, enhancing theory building and filling gaps, especially in emerging fields where conventional reviews are inadequate, as noted by (Greenhalgh & Peacock, 2005).

In comparison to the other forms of literature reviews discussed above, the ILR technique stands out for its methodological flexibility and interdisciplinary scope, which allow it to handle complicated research questions. It integrates diverse sources across research methods, including published articles, grey literature, with both qualitative and quantitative studies, as noted by Broome (2000), Hopia et al. (2016), and Whittemore and Knafl (2005). This makes it a particularly effective approach for scholarly investigations requiring a detailed and all-encompassing examination, as required in this study.

2.2 Methods

To map existing knowledge and identify gaps in research on novice paired programmers' debugging in distributed environments, this study used an Integrative Literature Review (ILR) as a primary lens to synthesise various streams of literature, following the

framework proposed by Lubbe et al. (2020). In determining the most appropriate ILR methodology, this research scrutinised and contrasted various approaches recommended by leading academics. Notably, Whittemore and Knafl (2005), Torraco (2005) and Russell (2005) advocate for a five-phase model, albeit with slight differences in the objectives of particular phases, whereas Souza et al. (2010) suggest a six-phase technique. After evaluating the outlined ILR methods, the study embraced the concise five-step strategy that Lubbe et al. (2020) put forward, as depicted in Figure 1. This decision was influenced by its clarity and structured approach, which provides a straightforward path through the complex process of conducting an ILR. This clarity helps in systematically addressing the research objectives and ensuring that each step of the review is purposeful and contributes to the overarching goals of the study.

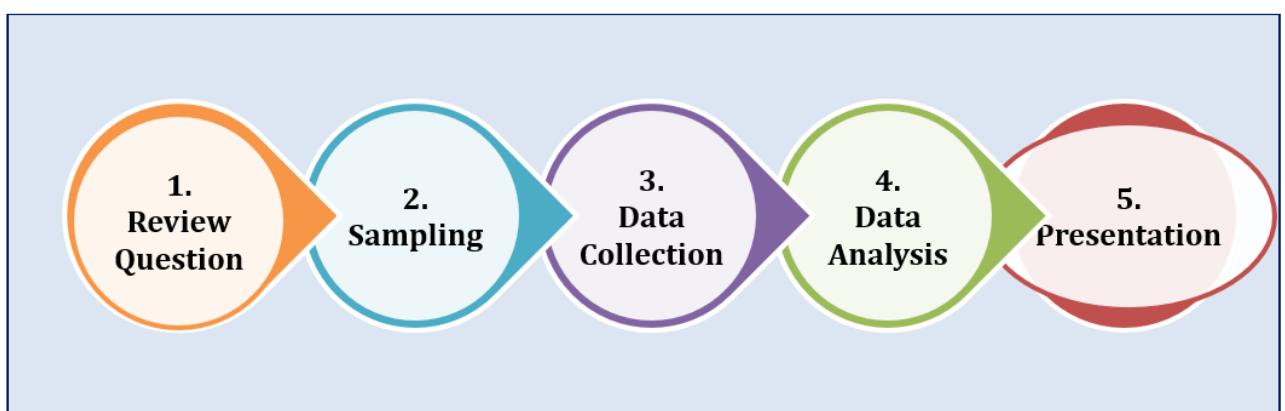


Figure 1: Integrative literature review (adapted from Lubbe et al., 2020)

2.2.1 Review question

The first step involves clearly defining the research question or problem that the review aims to address (Lubbe et al., 2020). This is crucial as it guides the search for relevant literature and the subsequent analysis. Through this exploration, the literature review establishes a solid foundation upon which the research questions can be thoroughly

examined and answered, enhancing the scholarly discourse surrounding the topic (Torraco, 2016).

The central question that steers this study centres on:

"How do the paired Software Development Apprentices in geographically distributed locations work collaboratively to fix Python programming bugs using the technology-mediated medium?"

As seen in the research question, this study explicitly explores the debugging strategies deployed, the tools employed and their effectiveness in addressing geographical separation. It also examines the types of programming bugs encountered and how paired novice programmers manage cognitive load. Ultimately, the study seeks to understand the interaction between technology, collaboration methods, task distribution, and the unique challenges of remote debugging.

2.2.2 Sampling

The literature sampling process consisted of two main steps, namely searching and screening. Relevant material was located during the searching phase by scanning academic databases, search engines, and other sources using precise keywords, Boolean operators, and search filters. Subsequently, titles and abstracts were screened for relevance using predetermined criteria such as subject relevancy, research type, and publication date. To complete the selection for data synthesis, the full texts of possibly relevant publications were compared to the inclusion criteria (Lubbe et al., 2020). This

methodical strategy ensured comprehensive coverage while excluding irrelevant or low-quality sources.

Searching

To answer the research question more fully, it becomes imperative to identify and collect relevant literature, a crucial step that typically involves database searches, manual journal reviews, and citation checks. Whittemore and Knafl (2005) emphasise a methodical and transparent approach and Carnwell and Daly (2001) stress the importance of a well-defined search strategy. In line with these recommendations, researchers, as highlighted by Kraus et al. (2022), frequently utilise multifaceted search techniques, including keyword searches, Boolean operators, and time-framed queries, to enhance their research effectiveness. Please refer to the visually illustrated processes presented in Figure 2.

In conducting the literature search for this study, a wide range of databases were utilised to ensure comprehensive coverage. These included Scopus, ProQuest, JSTOR, ERIC, IEEE Xplore Digital Library, ACM Digital Library, ScienceDirect, EBSCO, SAGE Journals, Web of Science, Education Full Text, PsycINFO, Academic Search Ultimate, and Google Scholar. The search was conducted between May 2021 and April 2024, using a structured query designed to enhance the relevance of the outcomes. The query employed specific terms and combinations: (“Distributed Pair Debugging” OR “Remote Pair Debugging” OR “Remote Collaborative Debugging” OR “Virtual Pair Debugging” OR “Distributed Pair Programming” OR “Virtual Pair Programming” OR “Distributed

Cognition" OR "Bug Location" OR Dyad*) AND (Error OR Bug OR "Bug Type*" OR "Error Type*") AND (Debug* OR "Pair Debug*").

This carefully constructed query represents a systematic approach to identifying relevant academic literature on distributed pair debugging. By using Boolean operators ("AND", "OR") and wildcards, it balances breadth and focus to capture studies spanning diverse contexts and terminologies. Terms like "Distributed Pair Debugging", "Remote Pair Debugging

" and "Virtual Pair Debugging" target collaborative debugging practices in distributed settings, while broader concepts such as "Distributed Cognition" and "Bug Location" ensure that related theoretical and practical dimensions are also covered. Wildcards (e.g., "Debug*" and "Dyad*") allow for variations in terminology, ensuring no relevant results are overlooked, and quotation marks around exact phrases maintain precision by avoiding irrelevant results. However, the complexity of the query may lead to an overwhelming number of results in less sophisticated databases, and terms like "Distributed Cognition" may retrieve studies beyond the primary focus on debugging. Additionally, the absence of exclusion criteria, such as the "NOT" operator to filter unrelated topics, could affect precision. Despite these limitations, the query is robustly designed to map the current state of research and identify gaps in distributed pair debugging, aligning closely with the study's objectives.

However, given the qualitative focus of this study on behaviour and experiences rather than quantifiable scientific data, the SPIDER tool (Cooke et al., 2012) was utilised to identify crucial aspects of the research question. This approach aimed to guide and unify

the search strategy, aligning it with the SPIDER tool's framework. Additionally, search terms were truncated as needed in both searches to ensure all pertinent articles were captured. Thus, entering this query into research databases facilitated a thorough and precise search. The terms were specifically selected to generate focused results, encapsulating the main areas of interest within the study.

ILR Process Steps	Sample of records
2. Sampling process	<p>Source of evidence: Scopus, ProQuest, JSTOR, ERIC, IEEE Xplore digital library, ACM Digital Library, ScienceDirect, EBSCO, SAGE Journals, Web of Science, Education Full Text, PsycINFO, Academic Search Ultimate and Google Scholar</p> <p>Keyword formulation ("Distributed Pair Debugging" OR "Remote Pair Debugging" OR "Remote Collaborative Debugging" OR "Virtual Pair Debugging" OR "Distributed Pair Programming" OR "Virtual Pair Programming" OR "Distributed Cognition" OR "Bug Location" OR Dyad*) AND (Error OR Bug OR "Bug Type*" OR "Error Type*" OR IDE) AND (Debug* OR "Pair Debug*")</p> <p>Inclusion criteria Primary research in peer-reviewed journals or grey literature Employing qualitative, quantitative, and mixed methods Published in English language Full-text available Published from 1999</p> <p>Exclusion criteria Distributed system Distributed program Studies with non-novice participants</p>

PRISMA Flow Diagram for ILR

```

graph TD
    A[Records identified through database searching (n = 839)] --> B[Records after duplicates were removed (n = 786)]
    C[Additional records identified through other sources, i.e. reference lists (n = 85)] --> B
    B --> D[Records screened (n = 786)]
    D --> E[Full-text articles assessed for eligibility (n = 53)]
    E --> F[Records included in critical appraisal process (n = 39)]
    F --> G[The final number of documents included for data synthesis (n = 35)]
    D --> H[Records excluded (n = 733)]
    E --> I[Excluded due to out of scope (n = 14)]
    F --> J[Records excluded due to poor methodological quality (n = 4)]
    
```

Identification

- Records identified through database searching (n = 839)
- Additional records identified through other sources, i.e. reference lists (n = 85)

Screening

- Records after duplicates were removed (n = 786)
- Records excluded (n = 733)
- Records screened (n = 786)

Eligibility

- Full-text articles assessed for eligibility (n = 53)
- Excluded due to out of scope (n = 14)
- Records included in critical appraisal process (n = 39)
- Records excluded due to poor methodological quality (n = 4)

Included

- The final number of documents included for data synthesis (n = 35)

Figure 2: Flow diagram for ILR (adapted from PRISMA – Preferred Reporting Items for Systematic Reviews & Meta-Analysis (Moher et al. (2009) cited in (Lubbe et al., 2020)).

Furthermore, the AI tool Connected Papers was employed to represent cited works visually, providing an extensive overview of the research landscape pertinent to the topic (see Figure 3). Connected Papers facilitates an innovative approach to academic research by visualising interconnections between papers, thus streamlining literature reviews, discovering trends, and identifying collaboration opportunities.

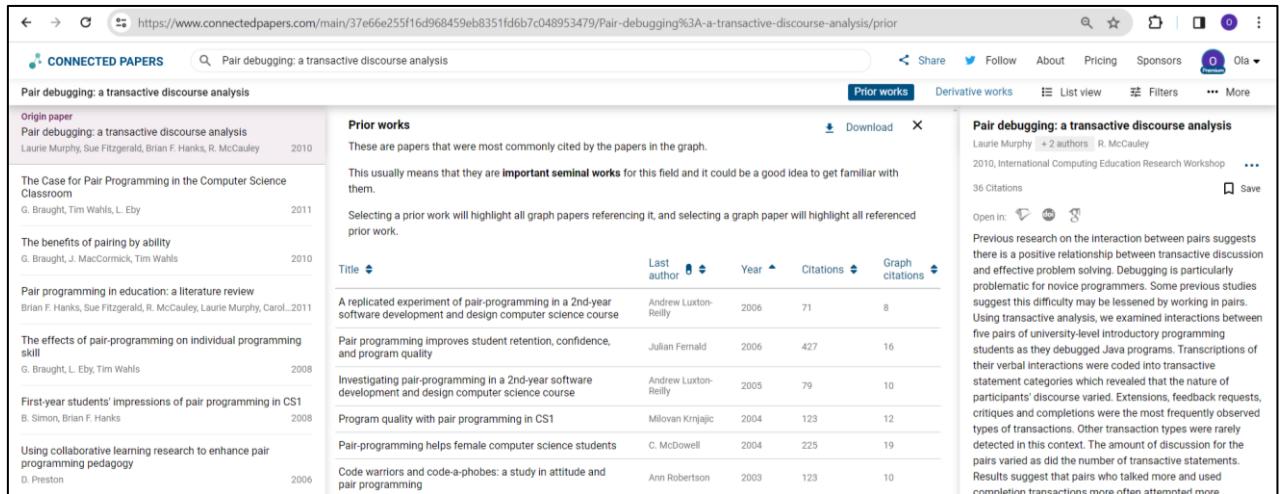


Figure 3: AI tool connected papers

Screening

In the preliminary literature review phase, as visualised in Figure 2, the titles and abstracts of papers identified through the initial search were examined. This process adhered to predefined inclusion and exclusion criteria to ensure the relevance and specificity of the study's objectives. Any duplicate papers identified during this review process were promptly eliminated to maintain the integrity and originality of the research material. Following this screening, full-text versions of the selected articles were procured for a more detailed evaluation, where they were once again scrutinised against the established inclusion and exclusion criteria. This approach ensured that only the most pertinent and informative papers were included in the study, thereby bolstering the research's foundational literature base.

2.2.3 Critical appraisal (CA) of sample (data collection)

Following Whittemore and Knafl's methodology (2005), the data were evaluated based on three essential criteria, specifically, methodological rigour, analytic precision, and conceptual relevance to the study aims. In adherence to this, this study adopted the Joanna Briggs Institute (JBI) checklists (Joanna Briggs Institute, 2017) to assess the methodological quality of various research types, including randomised controlled trials, cohort studies, case-control studies, cross-sectional studies, and systematic reviews. These checklists, tailored to each study type, feature specific criteria for assessing relevant methodological aspects, ensuring thorough evaluation of research designs.

The JBI checklist does not use a numerical scoring system. However, this study adapts it to a 0 to 1 scale for clear, quantifiable evaluation - 0 for "No", 0.5 for "Partially", and 1 for "Yes". Since JBI does not set a specific quality cut-off, score interpretation varies by context. Thus, specific guidelines are applied for this study:

- Outstanding (90-100%): Papers in this range meet most or all criteria, indicating well-executed research with thorough methodology and ethical considerations.
- Good (75-89%): Papers in this range are good to very good, meeting most criteria with minor areas for improvement.
- Fair (60-74%): Papers in this range are deemed adequate, but several areas need improvement.
- Poor (<60%): Scores below 60% indicate significant methodological weaknesses and such papers might require considerable revision to be of high quality.

Table 1 (refer to Appendix L for a full list) displays a sample summary document of studies appraised using the JBI checklist at the full-text screening stage. Critical appraisal from this study's perspective is a detailed, systematic review that assesses studies' methodological quality, validity, reliability, and relevance in a specific context (Porritt et al., 2014).

Table 1: Sample of a summary document for the critical analysis (CA) of selected studies

Included studies	CA tool	Quality rating	Evidence level
Allwood, C. M., & Bjorhag, C.-G. (1990). Novices' debugging when programming in Pascal.	JBI	Outstanding	90%
Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students.	JBI	Outstanding	90%
Murphy, L., Fitzgerald, S., Hanks, B., & McCauley, R. (2010). Pair debugging: a transactive discourse analysis.	JBI	Outstanding	95%
Yen, C.-Z., Wu, P.-H., & Lin, C.-F. (2012). Analysis of experts' and novices' thinking process in program debugging.	JBI	Outstanding	95%
Alqadi, B. S., & Maletic, J. I. (2017). An Empirical Study of Debugging Patterns Among Novices Programmers.	JBI	Outstanding	95%
Jayathirtha, G., Fields, D., & Kafai, Y. (2020). Pair debugging of electronic textiles projects: Analysing think-aloud protocols for high school students' strategies and practices while problem solving.	JBI	Outstanding	95%
Kim, C., Vasconcelos, L., Belland, B. R., Umutlu, D., & Gleasman, C. (2022). Debugging behaviours of early childhood teacher candidates with or without scaffolding.	JBI	Outstanding	95%
Alaboudi, A., & LaToza, T. D. (2023). What constitutes debugging? An exploratory study of debugging episodes.	JBI	Outstanding	95%
Liu, Q., & Paquette, L. (2023). Using submission log data to investigate novice programmers' employment of debugging strategies.	JBI	Outstanding	95%
Zhang, Y., Paquette, L., Pinto, J. D., Liu, Q., & Fan, A. X. (2023). Combining latent profile analysis and programming traces to understand novices' differences in debugging.	JBI	Outstanding	95%
Jayathirtha, G., Fields, D., & Kafai, Y. (2024). Distributed debugging with electronic textiles: understanding high school student pairs' problem-solving strategies, practices, and perspectives on repairing physical computing projects.	JBI	Outstanding	100%
Parkinson, M. M., Hermans, S., Gijbels, D., & Dinsmore, D. L. (2024). Exploring debugging processes and regulation strategies during collaborative coding tasks among elementary and secondary students.	JBI	Good	75%

2.2.4 Data analysis (Data extraction and synthesis, and thematic analysis)

Data extraction

Data extraction involves systematically gathering relevant details from selected studies to fulfil the research objectives (Lubbe et al., 2020). This includes identifying study

characteristics (e.g., author, publication year), aims/purpose, research design, population, and main findings. The aim is to compile comprehensive information from each study for comparison, synthesis, and interpretation, organised in a structured format such as a table, as seen in Tables 2 and 3.

Table 2: Sample data extraction

Authors	Aim/Purpose	Research design	Population (Sample, Sample Size, and setting)	Findings
Basma, S. A. Jonathan, I. M. (2017)	To study debugging patterns among novice programmers, particularly focusing on common logical errors and their debugging behaviours. Purpose ▪ Understanding Debugging Challenges ▪ Improving Educational Tools and Strategies ▪ Empirical Investigation of Debugging	Employed two structured experimental designs to investigate debugging behaviours among novice programmers.	▪ Novice programmers ▪ Included 142 subjects across two separate experiments. ▪ The study was conducted in a controlled experimental setting, likely within a laboratory or a dedicated testing room.	▪ Error Identification Challenges ▪ Types of Logical Errors ▪ Debugging Strategies Used ▪ Influence of Experience ▪ Educational Implications ▪ Tool Effectiveness
Kim, C Vasconcelos, L Belland, B. R. Umutlu, D Gleasman, C (2022)	The paper aims to examine how early childhood teacher candidates learn to program and debug block-based code with and without scaffolding. Purpose ▪ Influence of Scaffolding on Debugging Approaches ▪ Effectiveness of Scaffolding in Learning ▪ Comparative Analysis of Debugging Behaviours with or without scaffolding.	A qualitative case study involved undergraduates majoring in early childhood education. Data were collected through video recordings, semi-structured interviews, and scaffold responses (if used)	▪ Participants were 13 undergraduate students from early childhood education courses (11 females and 2 males) ▪ Most had little to no prior programming experience. ▪ Conducted in an academic setting, offering early childhood education programs	▪ Scaffolding enhanced engagement and persistence ▪ Participants with scaffolding exhibited more structured and effective debugging strategies, ▪ Scaffolding facilitated better collaborative interactions among participants.

Table 3: Sample data extraction (Continuation)

Authors	Aim/Purpose	Research design	Population (Sample, Sample Size, and setting)	Findings
Zhang, Y. Paquette, L. Pinto, J. D. Liu, Q. and Fan, A. X. (2022)	<p>To explore how different novices exhibit distinct debugging strategies and outcomes using latent profile analysis (LPA) of programming traces in an undergraduate CS1 course.</p> <p>Purpose</p> <ul style="list-style-type: none"> ▪ Use latent profile analysis to classify novice programmers into distinct debugging profiles based on their programming task interactions. ▪ Analyse the correlation between debugging profiles and performance metrics like error handling accuracy and speed. ▪ Use profiling insights to develop personalised instructional strategies for novice computer science students. 	<p>The study employed a quantitative approach using data from programming submission traces of undergraduates in a CS1 course.</p> <p>Latent profile analysis was used to identify different debugging profiles based on variables related to debugging.</p>	<p>The study involved 617 undergraduate students from a CS1 course at a public university in the U.S., who consented to the use of their data.</p> <p>These students submitted programming solutions via a web-based system, and their interaction data were analysed</p>	<ul style="list-style-type: none"> ▪ Three distinct debugging profiles were identified: one with higher debugging accuracy and speed, and two with lower performances in handling runtime, logic, and syntactic errors. ▪ The study found that demographic and self-assessed skill levels were predictors of profile membership. ▪ Additionally, differences in debugging profiles correlated with different academic outcomes, suggesting the need for tailored debugging instruction early in CS education.
Jayathirtha, G. Fields, D. Kafai, Y. (2024)	<p>Investigates how high school students collaborate and solve problems while debugging electronic textiles in physical computing projects, emphasising their strategies, practices, and viewpoints.</p> <p>Purpose</p> <ul style="list-style-type: none"> ▪ To examine how paired students collaborate to share cognitive tasks and use debugging strategies to solve programming errors in a distributed setting. ▪ To capture students' perspectives on their debugging experiences. 	<p>Research design is qualitative, based on the analysis of think-aloud interviews and video observations of seven pairs of high school students engaged in debugging activities with pre-designed buggy e-textile projects.</p>	<p>The study involved fourteen high school students from a U.S. charter school, paired up and enrolled in an introductory computer science class, participating in debugging exercises as coursework.</p>	<ul style="list-style-type: none"> ▪ Pairing students enhanced their ability to collaboratively troubleshoot, share cognitive tasks and fix bugs in e-textiles projects. ▪ Pairs employed a variety of debugging strategies, including iterative testing to help navigate and solve complexities in physical computing projects. ▪ Students appreciated the engaging hands-on experience, noting that collaboration was crucial for overcoming challenges and enhancing learning during debugging.

Data synthesis and analysis

After completing the critical analysis of the included studies outlined in the previous section, the synthesis phase began, focusing on identifying themes, relationships, patterns, and gaps. Hart (2018) describes data synthesis as connecting components identified during analysis initiated as soon as the literature is compiled. Adhering to Braun and Clarke's (2006) guidelines, this stage involved compiling various studies already critically analysed to uncover new insights. Specifically, a thematic synthesis was carried out, as detailed by Braun and Clarke (2006), aiming to distil the evidence into coherent themes that respond directly to the research question. Please refer to Table 4 for a detailed understanding of the themes that emerged from this synthesis and to see how these themes are connected to the studies that informed them.

Table 4: Themes identified from data synthesis.

S/No	Theme	N ^a	Studies that informed the themes
1	Complexity and Diversity of Errors	21	(Gould & Drongowski, 1974), (Vessey, 1985), (Katz & Anderson, 1987), (Allwood & Bjorhag, 1990), (McCauley et al., 2008), (Murphy et al., 2008), (Fitzgerald et al., 2010), (Murphy et al., 2010), (Yen et al., 2012), (Akinola, 2014), (McCall & Kolling, 2014), (Ettles et al., 2018), (Júnior et al., 2019), (Kohn, 2019), (Smith & Rixner, 2019), (Jayathirtha et al., 2020), (Jeffries et al., 2022), (Alqadi & Maletic, 2017), (Zhang & Norman, 1994), (Jayathirtha et al., 2024)
2	Tapestry of Debugging Strategies	14	(Gould & Drongowski, 1974), (Vessey, 1985), (Allwood & Bjorhag, 1990), (Katz & Anderson, 1987), (Ahmadzadeh et al., 2005), (Chintakovid et al., 2006), (Fitzgerald et al., 2008), (Fitzgerald et al., 2010), (Murphy et al., 2008), (Alqadi & Maletic, 2017), (Jayathirtha et al., 2020), (Jayathirtha et al., 2024), (Liu & Paquette, 2023), (Whalley et al., 2023)
3	Team Cognitive Management	3	(Chintakovid et al., 2006), (Jayathirtha et al., 2020), (Jayathirtha et al., 2024)
4	IDE Debugging Efficiency	4	(Smite et al., 2021), (Fitzgerald et al., 2008), (Whalley et al., 2023), (Jayathirtha et al., 2024)
5	Navigating Debugging Complexities	9	(Michaeli & Romeike, 2020), (Alqadi & Maletic, 2017), (Fitzgerald et al., 2008), (Fitzgerald et al., 2010), (Murphy et al., 2010), (Smite et al., 2021), (Kim et al., 2022), (Whalley et al., 2023), (Jayathirtha et al., 2024)
N ^a – Number of papers			

2.2.5 Presentation

After analysis, the findings are presented in a structured format, including a narrative summary, making it easy to understand the main findings. An initial search across chosen databases produced 924 papers. After reviewing titles and abstracts, applying exclusion criteria, and eliminating duplicates, 53 articles were shortlisted for closer examination. Following an in-depth full-text review, 14 articles were excluded for not directly addressing the review topic, and an additional 4 papers were excluded due to poor methodological quality, leaving 35 papers deemed suitable for inclusion in this review (see Figure 2).

Based on the review of the literature and alignment with the guidelines provided by Braun and Clarke (2006), the key themes that emerged, influencing the debugging behaviours of novice programmers across various settings were Complexity and Diversity of Errors, Tapestry of Debugging Strategies, Team Cognitive Management, IDE Debugging Efficiency, and Navigating Debugging Complexities. These themes collectively cover the factors influencing debugging behaviours, providing a comprehensive framework for understanding and enhancing the debugging skills of novice programmers in various settings.

Theme 1 - Complexity and Diversity of Errors

The studies reviewed, which span several decades, provide insights into the diverse types of errors encountered by novice programmers in various programming languages and environments.

This theme presents an overview of the findings regarding the different types of errors or bugs encountered by novice programmers, detailing commonalities and insights across various studies.

The study conducted by Gould and Drongowski categorised the debugging challenges faced by beginners in Java programming into syntax, logic, and runtime errors, providing early insights into the distinct categories of errors encountered by novice programmers (1974). Leveraging these insights, Katz and Anderson (1987) further emphasised the importance of understanding these categories to improve debugging efficiency, highlighting how different types of errors necessitate different approaches (Michaeli & Romeike, 2019). Offering a comparative angle, Vessey (1985) documented higher error rates among novices compared to experts in COBOL programming, illuminating the steep learning curve novices encounter and the more efficient debugging strategies experts use. Similarly, Yen et al. (2012) explored differences in debugging strategies between novice and expert programmers in C, revealing that novices struggle significantly with semantic and logic errors due to less effective use of compiler feedback.

Within specific software development environments, Ahmadzadeh et al. (2005) and Kölling et al. (2019) focused on how novices handle compiler errors and logical mistakes within environments like the BlueJ IDE, noting frequent logical missteps by novices. Fitzgerald et al. (2008), Murphy et al. (2008) and Murphy et al. (2010) elaborated on the particular types of Java errors, such as arithmetic bugs, malformed statement bugs, and incorrect logical expressions, pointing out the particular difficulties novices face in Java

environments. These studies collectively highlight the challenges and learning obstacles presented by Java programming.

Further investigations into novice errors in different programming languages were conducted by Alqadi and Maletic (2017), who explored how novices misapply logical operators and control structures in Java, emphasising the need for a deep understanding of logic to navigate debugging. Smith and Rixner (2019) focused on Python-specific errors, identifying frequent runtime errors such as `TypeError`s and `IndexError`s that persist among novices, highlighting the persistent challenges in modern programming languages.

Focusing on young learners and specific error types, Júnior et al. (2019) and Kohn (2019) documented Python coding mistakes among high school students, such as unclosed scanners and incorrect indentation, emphasising the need for clear compiler error messages and effective pedagogical methods to aid students in overcoming these foundational hurdles.

Moreover, in studies exploring physical computing and hardware-related errors, Jayathirtha et al. (2020) conducted a study that revealed programming errors such as missing initialisation, incorrect logical expressions, and mismatched variables, alongside circuitry issues like loose connections and reverse polarity problems, which were included intentionally to mimic real-world scenarios. Similarly, the study by Jayathirtha et al. (2024) further probed into pre-designed bugs affecting both hardware and software,

identifying errors like faulty conditional logic, wiring issues, sensor inaccuracies, and LED malfunctions.

Other miscellaneous studies provide broader insights into the challenges faced across various environments. The study conducted by Allwood and Bjorhag (1990) looked into Berkeley-Pascal programming errors on UNIX, categorising errors into syntax, semantic, and logic errors and underlining the distinct challenges in this environment. Ettles et al. (2018) analysed prevalent logic errors in C programming among first-year students, providing insights into common misconceptions and algorithmic mistakes. Jeffries et al. (2022) and Zhang et al. (2023) provided insights into syntax and runtime errors in Python and Java, respectively. Jeffries et al. (2022) focused on Python, identifying frequent mistakes such as indentation errors, incorrect use of variables, and misunderstanding of functions. Zhang et al. (2023), on the other hand, examined Java and pointed out typical errors like class and object mismanagement, improper exception handling, and issues with data types.

Lastly, Akinola (2014) compared debugging effectiveness between solo and pair programmers in Java, noting that collaborative approaches might mitigate some common errors, suggesting that pair programming could be a beneficial strategy in educational settings where teamwork and collaboration are emphasised. This study, among others, reinforces the varying dynamics of learning and debugging within programming education and points towards potential strategies for enhancing novice programmers' skills.

Together, these studies demonstrate the variety of programming errors and the critical need for all-encompassing instructional approaches that equip novice programmers with robust debugging skills. This foundational knowledge is pivotal for programming education as it aims to enhance learning outcomes by providing novices with the tools and methodologies to tackle the broad spectrum of debugging challenges they encounter.

Theme 2 - Tapestry of Debugging Strategies

This theme highlights debugging strategies across various studies, offering a deep insight into the complex techniques that novices use to tackle the task of debugging. This overview showcases the diversity of debugging strategies and tactics employed and reflects the evolution of debugging as a pivotal skill in software development.

Beginning with Gould and Drongowski (1974), the study laid the groundwork for understanding debugging strategies such as print debugging, code inspection, trial and error, collaboration, and utilising IDEs. These strategies highlight fundamental interactions between programmers and code, emphasising a dynamic approach to identifying and resolving errors. Following this, Vessey (1985) introduces a cognitive dimension by contrasting experts' holistic, systematic strategies with the more linear, focused approaches of novices, underlining the impact of cognitive processes on debugging effectiveness.

Adding further depth, Allwood and Bjorhag (1990) describe the debugging processes of novices using Pascal, incorporating error hypotheses, systematic problem-solving, and

iterative debugging phases. They emphasise understanding code logic and scrutinising error messages, which are crucial for a structured debugging approach. Katz and Anderson (1987) complement this by exploring debugging in LISP programming, identifying strategies like simple mapping, test-case execution, and causal reasoning. They note the use of both backward and forward reasoning, varying by whether novices debug their code or that of others, which introduces strategic flexibility in debugging. This observation is echoed by findings from Fitzgerald et al. (2008), Horwitz et al. (2009), LaToza et al. (2020), and Vourletsis et al. (2021). However, a deviation was reported by Yen et al. (2012), who found that students, while debugging C language programs crafted by others, also favoured the backward reasoning approach. Notably, Katz and Anderson observed a heightened use of causal reasoning, contrasting Jeffries (1982) earlier observation of its limited application. However, a deviation was reported by Yen et al. (2012), who found that students, while debugging C language programs crafted by others, also favoured the backward reasoning approach. Notably, Katz and Anderson observed a heightened use of causal reasoning, contrasting Jeffries (1982) earlier observation of its limited application.

The discussion of tactical debugging continues with Ahmadzadeh et al. (2005), who observe novice computer science students employing print statements, code commenting, and active code running to isolate bugs. This hands-on approach reflects an interactive engagement with the code, where manipulation and direct observation are crucial to understanding and fixing errors. Similarly, Chintakovid et al. (2006) extend debugging to spreadsheet environments, focusing on iterative debugging, testing values, and using visual cues to enhance error detection in formula-based contexts. Further

contributions from Fitzgerald et al. (2008) introduce pattern matching and different reasoning strategies, adding a layer of structured creativity to debugging. This is echoed by Murphy et al. (2008), who emphasise tracing, selective commenting of code sections, and systematic testing. Fitzgerald et al. (2010) then expand the range of strategies to include understanding code, gaining domain knowledge, and utilising resources, which deepen the cognitive and resource-based aspects of debugging.

Alqadi and Maletic (2017) emphasise logical errors, advocating strategies such as error hypothesis formation, systematic testing, code tracing, incremental and iterative testing, backtracking, and peer review. These methods emphasise the importance of a collaborative, iterative approach to uncovering syntactically correct but logically flawed errors. Jayathirtha et al. (2020) and their subsequent study in 2024 investigate debugging in electronic textiles and Arduino projects (Jayathirtha et al., 2024), highlighting the multidimensional challenges of integrating code with physical components through strategies like hypothesis generation, solution testing, and iterative problem-solving.

Lastly, Liu and Paquette (2023) incorporate modern data analytics to explore debugging through submission logs, revealing strategies such as minor code edits that contrast traditional debugging perceptions. Whalley et al. (2023) focus on effective strategies among novice Python programmers, emphasising the importance of understanding code, hypothesising and using deliberate actions to locate bugs.

Overall, these studies illustrate a complex tapestry of debugging strategies, highlighting a domain where technical, analytical, collaborative, and cognitive skills converge to address

one of the most challenging aspects of programming. Each study contributes unique insights, enriching our understanding of how diverse debugging strategies are applied across different contexts, languages, and platforms, underscoring the need for adaptive, context-sensitive, and collaborative approaches in both education and professional practice.

Theme 3 – Team Cognitive Management

Team Cognitive Management aptly encapsulates the cognitive strategies presented in the studies by Chintakovid et al. (2006), Jayathirtha et al. (2020), and Jayathirtha et al. (2024). This theme emphasises the efficacy of paired or group collaboration in managing cognitive load within programming and debugging contexts. The studies investigate the collaborative mechanisms participants use to distribute and share cognitive responsibilities, thereby enhancing the overall problem-solving process. Each study distinctly contributes to understanding how these synergies facilitate effective cognitive load management, demonstrating a range of strategies from role division and task coordination to adaptive problem-solving and joint attention.

In the study conducted by Chintakovid et al. (2006), the participants, all university students with limited programming experience, utilised several collaborative strategies to distribute cognitive load while engaging in programming tasks. They adopted Driver-Observer roles, allowing one person to handle direct manipulation of the code (the driver) while the other provided strategic oversight (the observer). This role division facilitated a balanced approach to task management and enhanced mutual support and effective communication. Participants also actively engaged in collaborative decision-making,

discussing potential solutions and strategies, which helped distribute cognitive responsibilities evenly across the pair. Moreover, task coordination was emphasised, with both members staying actively involved throughout the debugging process, ensuring that cognitive load was shared and not concentrated on a single individual.

Transitioning to the study by Jayathirtha et al. (2020), similar collaborative dynamics were observed among participants working on electronic textiles projects. Here, the divide-and-conquer strategy was notably effective, with tasks split between circuitry and coding based on individual expertise and task complexity. Such division allowed each participant to focus intensively on a specific project segment, reducing individual cognitive load. Collaborative task allocation was another significant strategy, where tasks were assigned based on each participant's skills, facilitating parallel progress and shared responsibility. The concept of joint attention to problem spaces was critical in this context; by co-investigating issues and verifying connections, participants could leverage their combined expertise to tackle complex problems effectively. Similarly, adaptive collaboration was highlighted as participants shifted strategies based on task demands, showing flexibility in managing cognitive load dynamically.

Finally, in a recent study by Jayathirtha et al. (2024), the focus shifted slightly towards more integrated collaborative strategies in debugging e-textile projects. Establishing joint attention was crucial, as students aligned their focus on various project elements, enhancing coordination and collective problem-solving. Sharing cognitive load was achieved through dynamic task division and continuous dialogue about strategies and solutions, enabling efficient use of collective cognitive resources. Collaborative problem-

solving was evident as students discussed, tested, and refined their approaches together. Fluid task division allowed for flexible role adjustments based on the immediate needs of the debugging process, further supporting effective cognitive load management. Coordinated strategies across different modalities ensured that all aspects of the projects were addressed comprehensively, facilitating a thorough approach to problem-solving.

These studies show that effective collaborative cognitive load management in programming and debugging involves a mixture of strategic role division, adaptive task allocation, and sustained mutual support. Each study highlights the benefit of collaborative approaches in distributing cognitive load and enhancing overall problem-solving efficiency and project success.

Theme 4 – IDE Debugging Efficiency

IDE Debugging Efficiency highlights how using IDE tools and related technologies significantly aids programmers in debugging tasks, enhancing efficiency and reducing cognitive load through various technological interventions.

In the work of Smite et al. (2021), technological tools were leveraged to facilitate remote pair programming sessions. Tools such as Tuple and various IDE extensions enabled screen sharing, control of each other's computers, and simultaneous programming. These tools significantly enhanced the collaborative experience and enabled real-time collaboration and code editing, thus boosting the debugging process's effectiveness and efficiency.

Fitzgerald et al. (2008) focused on the role of debuggers within IDEs like Eclipse, which automatically detect and highlight semantic errors such as missing brackets. This capability speeds up the identification and correction of errors and reduces the programmers' cognitive load. Additionally, the study emphasised the importance of online resources and programmers' familiarity with the IDE, noting that these factors significantly contribute to successful debugging efforts.

Moreover, Whalley et al. (2023) observed how novice programmers utilised IDE tools to manage their workspaces efficiently and execute code changes effectively. The participant's ability to organise their workspaces and engage in modify-and-test cycles showcased how IDE tools could simplify and streamline the debugging process. However, the study also noted challenges related to the participants' familiarity with IDE functionalities, emphasising the need for full training to leverage these technologies.

Lastly, Jayathirtha et al. (2024) examined using the Arduino IDE in debugging electronic textiles projects. This study highlighted the IDE's features, such as syntax highlighting, error detection, interactive debugging, and integrated tools and libraries. These functionalities facilitated the programming process and supported students in managing, navigating, and debugging their code more effectively.

Overall, each study contributes to the overarching theme by demonstrating how IDE tools and technologies are integral to enhancing debugging efficiency, which is the focus of this study. These studies jointly reinforce the transformative impact of these technologies in

programming education and practice, providing essential insights into their benefits and the necessity of familiarity with these tools for effective debugging.

Theme 5 – Navigating Debugging Complexities

The literature reviewed explores the diverse challenges novices encounter during programming debugging tasks. This theme captures novices' experiences across various educational backgrounds as they navigate the complexities of identifying and resolving errors within code. The subsequent findings detail the specific challenges and underlying reasons identified in each study, shedding light on the common obstacles faced during the debugging process.

Michaeli and Romeike (2020) introduce a complex educational scenario where participants face seven distinct challenges while debugging, namely, generating hypotheses, undoing changes, systematic testing, cognitive load, use of external representations, collaboration and communication, and application of domain knowledge and heuristics. Thus, the inability to generate effective hypotheses and the reluctance to undo changes post-failure indicate a lack of prior debugging experience, which hampers effective problem-solving strategies. Systematic testing is compromised by a shallow understanding of isolating and verifying system components, exacerbated by the high cognitive demands of managing multiple variables simultaneously. Ineffectiveness in using external representations and challenges in collaboration and communication are linked to insufficient collaborative skills and the inappropriate application of heuristics, compounded by the unique pressures of an escape room setting that distracts from focused debugging efforts. These challenges are intertwined with

reasons such as environmental novelty, educational gaps, and a lack of practical debugging exercises, which highlight the necessity for educational reforms to better prepare students for real-world debugging tasks.

Alqadi and Maletic (2017) detail the struggles of novice programmers with five primary challenges, such as difficulty with logical errors, lack of experience, understanding error messages, cognitive load, and time management. Logical errors, particularly challenging due to their requirement for a deeper understanding of the program's intent, highlight the novices' inadequate exposure to complex debugging and systematic strategies. Misinterpretation of error messages and an overwhelmed cognitive capacity due to the simultaneous management of multiple debugging elements like program flow and variable states further complicate the debugging process. These issues are often deepened by educational shortcomings that fail to equip students with necessary debugging skills and are exacerbated by psychological factors such as anxiety and frustration, which negatively impact problem-solving capabilities.

In Fitzgerald et al.'s (2008) research, seven challenges surface, encompassing understanding the system, testing the system, locating and repairing errors, using debugging tools, cognitive load, and fragile knowledge. Each challenge is rooted in a combination of lack of experience and insufficient foundational knowledge, which hinders effective engagement with debugging tools and systematic problem-solving. Cognitive overload and fragile knowledge, where concepts are not fully grasped, stress educational gaps that fail to prepare students for debugging's unpredictable nature. This

calls for an educational approach emphasising practical experience and systematic problem-solving skills in programming curricula.

Fitzgerald et al. (2010) explore the debugging experiences of novice programmers, identifying five main challenges, for example, fragile knowledge, troubleshooting, causal reasoning, understanding debugging tools, and reading complex code. The reasons for these challenges are intricately linked to the novices' superficial understanding of programming and debugging, compounded by high cognitive loads and ineffective use of debugging tools. The study suggests that enhancing educational practices to include more focused debugging exercises could alleviate these challenges.

Murphy et al. (2010) focus on the collaborative mechanics of debugging in pairs, noting five challenges related to transactive communication, cognitive load, collaborative dynamics, level of discussion, and strategic application of debugging methods. The additional cognitive burden and the need for effective transactive communication highlight the complexity of collaborative debugging, which is often not adequately supported by educational frameworks that fail to emphasise collaborative skills and systematic debugging strategies.

Smite et al. (2021) document the adaptation to remote pair programming, identifying challenges such as disruptions in communication and collaboration, adaptation to remote tools, loss of informal communication, psychological impacts, and adjustments to new work rhythms. These challenges stem from technological and organisational shifts

necessary for remote work, highlighting the need for better support systems and training to facilitate effective remote collaboration and debugging.

Another relevant study by Kim et al. (2022) on block-based programming involving early childhood teacher candidates identifies six challenges influenced by the presence of scaffolding. These include complex problem-solving, persistence, collaborative dynamics, cognitive load, technical understanding, and trial-and-error approaches. Prominent issues such as lack of experience and cognitive overload suggest that structured scaffolding could significantly aid in managing these challenges.

Whalley et al. (2023) investigate the debugging practices of novice programmers, identifying six key challenges, including difficulties with debugging tools, code navigation, strategy application, cognitive load, problem-solving constraints, and interpreting feedback. These challenges are predominantly due to limited experience with debugging tools and an inadequate understanding of code structure and flow, which could be mitigated by more comprehensive programming education emphasising practical debugging skills and tool usage.

Moreover, Jayathirtha et al. (2024) study the debugging of electronic textiles by high school students, identifying five challenges, including, complex multimodal debugging, distributed tasks, collaborative coordination, limited engagement, and system integration difficulties. The all-encompassing nature of these projects introduces unique challenges that require both collaborative efforts and an integrated understanding of

diverse systems, pointing to the need for educational tools and resources that support such complex, interdisciplinary learning environments.

Overall, these studies highlight novice programmers' broad challenges, particularly in collaborative and distributed settings. The findings suggest that debugging effectiveness heavily depends on managing cognitive load, proficiently utilising debugging tools, and maintaining effective communication and collaboration. For paired novices in distributed environments, these challenges are magnified by the additional barriers of remote collaboration.

2.3 Discussion

This section attempts to critically analyse and synthesise the findings from multiple studies relating to the five themes already identified in the previous section. It aims to integrate these diverse insights, providing an understanding of the study patterns and divergences. By examining these themes, which are the Complexity and Diversity of Errors, the Tapestry of Debugging Strategies, Team Cognitive Management, IDE Debugging Efficiency, and Navigating Debugging Complexities, the discussion will give a comprehensive analysis of the data. In this vein, this examination seeks to contextualise these findings within novice programming behaviour and highlights the broader implications for programming education. It also suggests potential strategies for improving novice programmers' learning experiences and outcomes.

Although the examined studies span several decades, they reveal profound educational insights, particularly highlighting the diverse types of errors encountered by novice

programmers across different languages and environments. In their foundational study on Java programming, Gould and Drongowski (1974) categorised debugging challenges into syntax, logic, and runtime errors, setting a precedent for recognising the diverse nature of programming errors. This fundamental categorisation reinforced the necessity for distinct pedagogical approaches tailored to different error types, a notion further bolstered by Katz and Anderson (1987). They stressed that understanding these categories is crucial for enhancing debugging efficiency (Robins et al., 2006), indicating that a one-size-fits-all approach to teaching debugging is insufficient. It thereby emphasises the need for a differentiated approach to teaching debugging tailored to the specific types of errors novices face (Lewis & Gregg, 2016). For example, novices tend to adopt a trial-and-error approach for syntax errors, while logic errors often require more structured problem-solving techniques (Ettles et al., 2018). Runtime errors frequently necessitate understanding dynamic program behaviour and error handling (Alqadi & Maletic, 2017). This detailed insight is essential for creating effective educational strategies that address novice programmers' specific needs to become more proficient and efficient in debugging their code.

Also, the steep learning curve faced by novices, as documented by Vessey (1985), and the differential debugging strategies employed by experts versus novices, as explored by Yen et al. (2012), highlight the critical need for specialised instructional methods. Vessey's work illustrates that novices are slower and less effective in debugging, which explains why Youngs (1974) observes that novices often spot fewer bugs and take longer to fix than experts. This suggests that educational interventions should focus on bridging this gap by imparting expert strategies to beginners. Yen and colleagues (2012) further

expand on this by showing how novices struggle with semantic and logic errors, primarily due to ineffective use of compiler feedback. This research aligns with Chen et al. (2013), whose findings suggest that, without proper guidance and extensive practice, beginners often misinterpret error messages and inefficiently use debugging aids. This supports Yen et al. (2012), who emphasised the need for curriculum designs incorporating real-world tools and environments to improve novices' debugging skills. However, though valuable, Yen et al.'s recommendation may be impractical for institutions with limited resources, indicating a need for adaptable, resource-sensitive strategies.

Likewise, research by Ahmadzadeh et al. (2005) and Kölling et al. (2019) reveals that novices often struggle with compiler errors and logical mistakes, particularly in environments like the BlueJ IDE. This finding implies that hands-on learning experiences, which allow novices to engage with code directly, are vital. Fitzgerald et al. (2008), Murphy et al. (2008), and Murphy et al. (2010) suggest that such an approach enhances technical proficiency and builds confidence in handling real-world programming challenges. By practising in realistic settings, novices can develop a deeper understanding of the complexities involved in debugging, making them better equipped to handle similar issues in professional scenarios (Robins et al., 2003; Soloway & Spohrer, 2013).

In addition, the consistent finding across studies that novices frequently misapply logical operators and control structures (Alqadi & Maletic, 2017; Smith & Rixner, 2019) indicates a persistent gap in understanding fundamental programming concepts. This calls for deeper instruction in programming logic, highlighting that a robust grasp of these basics is essential for effective debugging. Júnior et al. (2019) and Kohn (2019) also noted

common Python coding mistakes among high school novices, further supporting the need for early programming education to include clear compiler error messages and effective pedagogical methods. Addressing these issues early can prevent the accumulation of bad coding habits and foster a more intuitive grasp of programming principles, which is crucial for developing proficient programmers (Grover & Pea, 2013; Guzdial, 2015).

Besides, the role of collaborative learning environments in mitigating common errors and enhancing learning outcomes is another critical insight from these studies. A study conducted by Akinola (2014) comparing solo and pair programmers suggests that collaborative approaches, such as pair programming, can significantly benefit educational settings. This is echoed in studies by Chintakovid et al. (2006), Jayathirtha et al. (2020) and Jayathirtha et al. (2024), which highlights the importance of strategic role division, task coordination and adaptive problem-solving in managing cognitive load and improving problem-solving efficiency. Educators can foster peer learning and mutual support by incorporating collaborative projects into programming curricula, creating a dynamic and interactive learning environment that mirrors real-world software development practices. This approach can help distribute the cognitive load as well as encourage the development of critical teamwork skills essential in the professional realm (Bennedsen & Caspersen, 2007; McDowell et al., 2006).

On top of that, it is important to note that exploring diverse debugging strategies across various studies has implications for programming education. The research emphasises the complexity and evolution of effective debugging practices from techniques like print debugging and code inspection (Gould & Drongowski, 1974) to more advanced methods

involving systematic problem-solving and error hypothesis formation (Alqadi & Maletic, 2017). This suggests that programming education should focus on technical skills and teach cognitive and strategic approaches to problem-solving, helping novices develop a more holistic understanding of debugging. By integrating these strategies into the curriculum, educators can better prepare novices for the multifaceted nature of debugging in professional settings. Such an approach ensures that novices are technically proficient and capable of thinking critically and strategically about problem-solving (Linn & Dalbey, 1985).

Furthermore, integrating modern tools and technologies in debugging, as demonstrated by Smite et al. (2021) and Jayathirtha et al. (2024), highlights the transformative impact of these technologies in programming education. Using IDE tools (Fitzgerald et al., 2010), remote collaboration technologies, and data analytics to enhance debugging efficiency and reduce cognitive load suggest that educational programs should keep pace with technological advancements. Familiarity with these tools can streamline the debugging process and improve overall productivity, highlighting the necessity for novices to be proficient in using the latest programming tools and platforms (Fitzgerald et al., 2008; Whalley et al., 2023). Consequently, integrating these technologies into educational settings can provide novices with practical experience and prepare them for the technological demands of the modern workplace (Grover et al., 2014; Resnick et al., 2009).

Moreover, the literature also points to the critical need for structured support systems and scaffolding to aid novice programmers. Studies involving young learners and early

childhood teacher candidates (Júnior et al., 2019; Kim et al., 2022; Kohn, 2019) suggest that clear error messages and structured guidance are essential for managing cognitive load and enhancing learning outcomes. This implies that educational institutions should provide scaffolding that gradually increases task complexity, ensuring novices are not overwhelmed by the cognitive demands of debugging. Such an approach can help maintain novice engagement and foster a progressive learning curve, making learning more manageable and effective for all levels (Wing, 2006).

Additionally, the research by Allwood and Bjorhag (1990) and Ettles et al. (2018) provide broader insights into the challenges faced in different environments, such as UNIX and C programming. Their findings suggest that novices struggle with syntax and logical errors and face significant challenges in understanding the operating environment. This highlights the importance of contextual learning, where novices are taught programming languages and the environments in which these languages operate. By fostering an understanding of the broader technical ecosystem, educators can better prepare novices for the diverse contexts they will encounter in their professional careers (Spohrer & Soloway, 1986).

To add to that, the role of cognitive and collaborative strategies in debugging, as explored by Chintakovid et al. (2006), Jayathirtha et al. (2020) and Jayathirtha et al. (2024), reinforces the importance of cognitive load management and strategic collaboration in effective problem-solving. Their studies suggest that educational programs should focus on individual problem-solving skills (Wing, 2006) and collaborative skills that can enhance cognitive efficiency (Chintakovid et al., 2006; Murphy et al., 2010). For instance, the use

of strategic role division and adaptive problem-solving in team settings can help distribute cognitive load and improve overall problem-solving efficiency. This approach mirrors real-world software development practices, where collaboration and teamwork are essential (Palumbo, 1990).

Despite this, research on integrating modern data analytics and remote collaboration tools (Liu & Paquette, 2023; Smite et al., 2021) indicates that the future of programming education lies in the effective use of technology. By incorporating these tools into the curriculum, educators can provide novices with the skills to navigate the software industry's increasingly digital and collaborative nature. This prepares learners for current industry practices and ensures they are adaptable to future technological advancements (McDiarmid & Zhao 2023).

By and large, these insights collectively highlight the urgent need for comprehensive educational strategies that equip novice programmers with robust debugging skills. Addressing the diverse challenges identified in the research, including technical, cognitive, and collaborative difficulties, requires a holistic approach. This approach should integrate targeted instructional methods, practical hands-on experiences, collaborative learning environments, and modern technological tools. By doing so, programming education can better prepare learners to tackle the broad spectrum of debugging challenges they encounter, ultimately improving debugging proficiency and contributing to software development practices' overall quality and efficacy.

2.4 Summary

As novice programmers commence their developmental journey, they demonstrate various debugging strategies, varying significantly in effectiveness. This literature review critically evaluates existing research, highlighting emergent patterns and pinpointing deficiencies. As Colquitt (2013) advocates, it is vital to juxtapose new research against established work, thus paying an ‘intellectual debt’ and ensuring a comprehensive grasp of the relevant scholarly landscape. In this vein, this chapter examines the characteristics and common bugs of novice programmers, the impact of IDE tools, and the variety of debugging strategies used by novices, particularly those working solo, co-located, and in distributed settings. This analysis is crucial as it prepares the ground for a detailed synthesis of research on paired debugging by novices in both co-located and distributed environments. Despite the growing literature on pair programming and debugging, a significant research gap exists in distributed pair debugging among novices, with no studies specifically focusing on the debugging strategies of paired novices in distributed settings.

Taking this into account, while the lack of targeted research is a limitation, the broader literature does provide some basis for understanding how factors such as communication, expertise distribution, and task complexity could potentially impact the success of pair debugging in distributed settings. Thus, the existing research serves as a starting point, highlighting the need for more focused studies to comprehensively understand the unique challenges and the debugging strategies that novice deployed distributed pair debugging among novice programmers.

Considering the current research, while the lack of specific studies is a limitation, the existing literature offers insights into how communication, expertise distribution, and task complexity might affect pair debugging in distributed settings. This establishes a foundation, pointing to the need for targeted research to fully understand the challenges and strategies of novice programmers in distributed pair debugging. Given this backdrop, this review identifies a clear gap in the literature and emphasises the need for dedicated studies in this less-explored area. Future research should focus on debugging techniques suitable for novices in diverse educational settings and ages, particularly in distributed environments. Such endeavours could unveil more detailed insights, facilitating the creation of bespoke strategies and tools to enhance the debugging process for novice pairs operating in remote environments.

Chapter 3: Conceptual Framework

3.0 Introduction

This chapter presents the conceptual framework that provides a structure for examining both individual and collaborative aspects of debugging within disparate settings, guiding the data collection and analysis. It approaches this by contextualising the study and linking it to underlying theories that provide a solid foundation for investigating and interpreting findings. For the purpose of this study, which examines the debugging process among novice programmers, the conceptual framework draws on two complementary theories, as exemplified by Information Foraging Theory (IFT) (Pirolli & Card, 1999) and Distributed Cognition (Hutchins, 1995). These complementary theories will now be reviewed, followed subsequently by the presentation of the Critical Analysis of the Distributed Pair Debugging Conceptual Framework.

3.1 Information Foraging Theory (IFT)

IFT provides insightful perspectives on information navigation and extraction within digital environments, focusing on debugging strategies by novices. Previous research validated this approach (Fleming et al., 2013; Lawrence et al., 2008; Piorkowski et al., 2012) and emphasised its effectiveness in software maintenance. Drawing from biological foraging analogies, IFT takes the information seeker as a 'predator' in pursuit of 'prey', valuable information within a network of interconnected information patches. This model introduces 'information scent', perceived from environmental cues, as a fundamental component in information-seeking tasks (Chi et al., 2001). However, applying IFT's constructs, this study explores novice programmer pairs' decision-making and

navigational challenges in distributed debugging settings, aiming to enrich our understanding of debugging practices within the contemporary networked programming landscape.

Furthermore, the intersection of information foraging and sensemaking processes highlights a dual-phase learning loop of information gathering and interpretation within software development, particularly in debugging, as Pirolli and Card (2005) discussed. This theoretical approach is pivotal in understanding programmers' foraging behaviours, with studies like Grigoreanu et al. (2012) emphasising the foraging loop's dominance in sensemaking activities. Thus, IFT offers a comprehensive framework for analysing programmers' information-seeking behaviours, providing a more integrated view of the processes involved compared to theoretical efforts and ultimately enriching the discourse on programming practices in the networked era.

3.2 Distributed Cognition

As Hollan et al. (2000) and Hutchins (1995) expound, distributed cognition offers a comprehensive framework that transcends the traditional, individual-focused cognitive science by considering cognitive processes as inherently shared among people, tools, and various representations. This approach, which Hutchins describes as encompassing cognitive activities across individuals, artefacts, and environmental factors, has seen application in a variety of domains ranging from ship navigation and emergency medical dispatch to aviation and call centres, thereby demonstrating its versatility in analysing teamwork and the integration of technology within human activities. Furthermore, Rogers (1997) highlights that distributed cognition enriches our understanding of

cognition by weaving together cognitive science, anthropology, and social sciences, thereby exploring the complex interdependencies inherent in collaborative efforts and how both social and organisational contexts shape them.

Furthermore, this paradigm shift provides a detailed insight into human-computer interaction, as suggested by Hollan and colleagues, by extending the analysis of cognitive processes to encompass broader systems beyond the confines of individual minds. It asserts that cognition is a collective phenomenon distributed across social groups, internal and external structures, and temporal dimensions, thereby providing a robust toolkit for examining the dynamic interplay between humans and technology. This shift is crucial for understanding the collaborative nature of cognitive tasks, including software development and debugging, where distributed cognition has only begun to make its mark, notably through the work of Flor and Hutchins (1991) in software maintenance.

Building on this foundation, Tsai et al. (2015) observe that pair programming, and by extension, pair debugging, significantly alleviates cognitive load, particularly in the context of distributed settings where the challenge of debugging error-prone code is compounded by the interplay of various factors including individual cognitive abilities, technological tools, and the social dynamics of using debugging tools effectively. However, while distributed cognition provides a deep analysis of socially distributed cognitive activities, Artman and Wærn (1999) critique it for potentially neglecting the non-cognitive artefacts within complex systems. To address this, IFT is introduced as a complement, aiming to shed light on the behavioural patterns of pairs as they navigate through code in search of errors.

Ultimately, this study seeks to bridge the gap in the application of distributed cognition within software development research, particularly in understanding the debugging strategies employed by novice programmers working collaboratively in distributed environments. Through a conceptual framework grounded in distributed cognition and complemented by IFT, this research aims to provide a richer, more cohesive understanding of pair debugging behaviours, thereby contributing to both theoretical knowledge and practical applications in software development.

3.3 Integration of IFT and Distributed Cognition

The combination of Information Foraging Theory (IFT) and Distributed Cognition in this research provides a nuanced exploration of how apprentices interact with their peers and tools in debugging code. Specifically, the study opines that apprentices use information foraging strategies to efficiently locate resources and scents that may assist in solving bugs. Subsequently, once these resources and scents are identified, apprentices engage in distributed cognitive activities to collaboratively implement the solutions. For instance, while one apprentice might search for external resources and suggest potential solutions (information foraging), the other apprentice simultaneously works to implement and test these solutions within the debugging environment. Thus, this dynamic illustrates how cognition and problem-solving are effectively shared and distributed across the pair (see Section 5.1.4).

Building on this foundation, integrating these theoretical frameworks seeks to explore how apprentices manage the challenges of remote collaboration, especially in debugging

code within distributed settings. Moreover, technological tools, such as shared IDEs and debugging interfaces, serve as communication channels that extend and support the cognitive processes involved. As a result, by distributing cognitive load between individuals and tools, these technologies either facilitate or, in some cases, hinder the debugging process. Therefore, this distribution of cognitive effort is key in determining how effectively apprentices can collaborate to resolve complex issues.

In conclusion, this approach illuminates the interplay between information foraging, distributed cognition, and the use of technology in enabling or constraining apprentices' problem-solving capabilities.

3.4. Critical Analysis of Distributed Pair Debugging Conceptual Framework

This conceptual framework melds Information Foraging Theory and Distributed Cognition to address the multi-dimensional aspects of debugging, with the Bug being central. Designed to encapsulate debugging's complexity in paired and distributed environments, it portrays each layer as distinct, contributing insights into the debugging process.

The framework provides a detailed examination of debugging in distributed environments, merging individual cognition, collaborative interaction, and environmental factors. As seen in Figure 4, arrows as visual metaphors demonstrate the impact of both individual and collective cognition on tool selection, aligning with the findings of Chalmers (2003) and Endsley's (1995) perspectives on the relationship between cognition, situational awareness, and tool usage.

Furthermore, it demonstrates the external debugging environment's role in shaping cognitive processes, resonating with Hutchins' (1995) insights on socio-technical systems cognition. The inclusion of bidirectional arrows between debuggers emphasises shared cognitive space, reinforcing the emphasis on shared cognition in solving complex problems by Salas et al. (2005) and Salas et al. (2008). Highlighting its dynamic nature, the framework presents a comprehensive approach to improving the understanding of debugging across research, education, and practical applications.

Accompanying this, Figure 4 presents a visual representation of each layer, delineating their inherent characteristics, data points, information flow dynamics, as well as their respective strengths and weaknesses. This framework thus stands as a tool for dissecting the intricacies of debugging within distributed settings, underpinned by seminal references in the field.

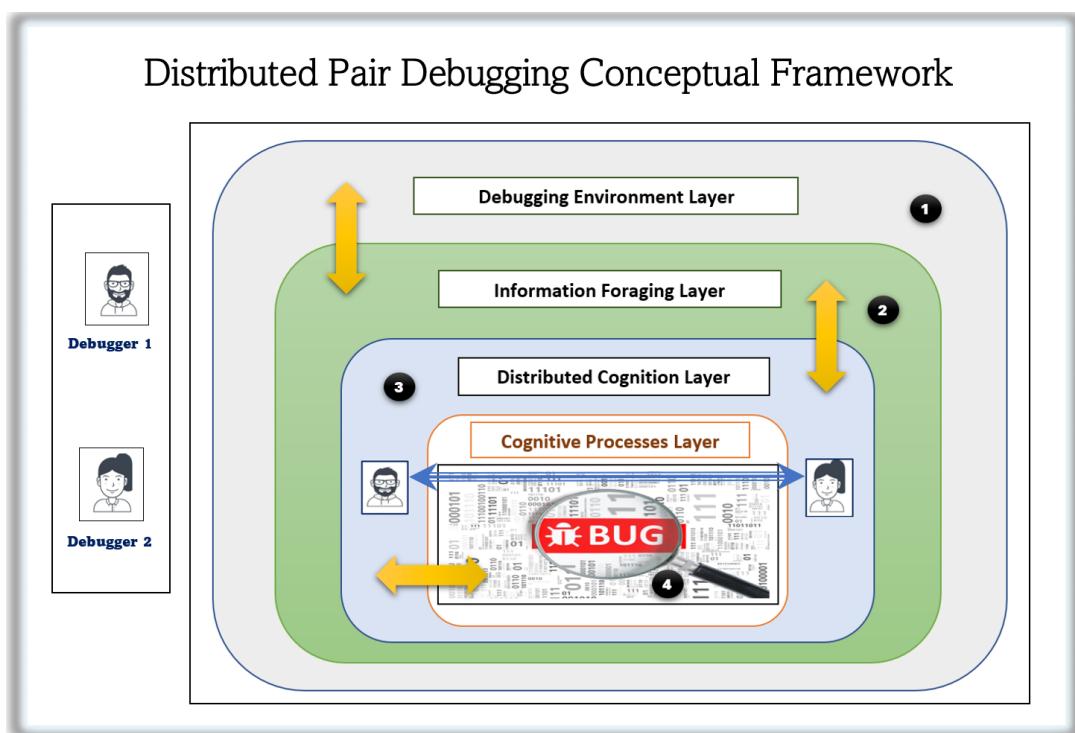


Figure 4: Distributed pair debugging conceptual framework

As a pioneering approach to understanding the intricacies of collaborative debugging in distributed settings, the conceptual framework offers valuable insights into the cognitive and collaborative processes involved. Although this framework has its limitations, it provides a thorough perspective on the complex nature of debugging tasks. It integrates theories of Distributed Cognition and Information Foraging to examine how pairs navigate and solve problems in a distributed debugging environment.

Furthermore, for a concise summary highlighting the strengths and weaknesses of the Distributed Pair Debugging Conceptual Framework, refer to Table 5. This table delineates the framework's key advantages and potential limitations, offering a visual representation to aid in understanding its comprehensive impact on debugging practices within distributed settings.

The following provides a thorough analysis of each layer and also evaluates its benefits and drawbacks, clarifying the different facets of each layer. As a result, this framework emerges as a tool for investigating the complexity of debugging in distributed contexts, backed up by fundamental references in the field.

Table 5: Strengths and weaknesses of the distributed pair debugging conceptual framework

Strengths	Weaknesses
<p>Nuanced Understanding of Human Behaviour: The framework's multi-layered approach enables a nuanced understanding of novice debugger behaviour by integrating Information Foraging and Distributed Cognition theories, exploring both the "what" and "why" aspects of individual and collaborative actions (Denzin & Lincoln, 2011). It also helps uniquely analyses the debugging process from environmental aspects to intricate cognitive processes. This holistic perspective captures a wide range of variables (Hollan et al., 2000; Pirolli & Card, 1999).</p>	<p>Cognitive Load on Novices: The framework's focus on studying novices necessitates consideration of their cognitive limitations, including attentional lapses and misunderstandings that may lead to increased cognitive load (Sweller, 1988). These factors could introduce extraneous variables into the data, potentially masking the core phenomena under investigation (Chandler & Sweller, 1991)</p>
<p>Versatility and Adaptability: The framework is versatile and adaptable, accommodating methodological pluralism. Qualitative research methods, like interviews, focus groups, or content analysis of communication channels among debuggers, can provide insights that empirical methods might miss (Flick, 2022). The framework's adaptability extends its relevance beyond novice debuggers, offering a potential universal model for debugging research (Ko et al., 2004).</p>	<p>Complexity and Time-Consuming: The complexity of the conceptual framework, while advantageous for a nuanced understanding, poses challenges in terms of time and resource allocation (Saldana, 2021). Particularly for early-career researchers, navigating the framework's multiple layers and variables may be daunting (Norman, 1993). This complexity could make the framework less suitable for studies seeking quick, straightforward outcomes, and could increase demands on resources for data collection and interpretation (Kirsh, 1995).</p>
<p>Depth of Understanding: Qualitative methods in the framework prioritise depth over breadth, offering in-depth insights into novice debugger processes through open-ended techniques like interviews and observations. This approach reveals tacit knowledge, unspoken emotions, and subtle dynamics often overlooked by quantitative data (Creswell, 2012; Patton, 2002)</p>	<p>Risk of subjectivity: Qualitative data can introduce subjectivity into analysis, requiring researchers to maintain rigor and validity when interpreting nuanced or ambiguous data (Tracy, 2010). While qualitative research excels at capturing detailed experiences, it can become a weakness if researchers' biases and preconceptions aren't actively addressed, especially within a comprehensive framework like this (Maxwell, 2012)</p>
<p>Iterative Exploration: The framework's flexibility supports iterative exploration, mirroring qualitative methods like grounded theory or other suitable methods. Researchers can revisit data, refine questions, and adapt the framework as new insights emerge (Charmaz, 2006).</p>	<p>Possible Redundancy: It's worth noting that certain aspects, such as individual cognitive processes and shared cognition, may contain overlapping elements that could result in redundancy during data collection or analysis (Zhang & Norman, 1994).</p>
<p>Empirical Focus: The framework's empirical approach emphasises measurable data, facilitating rigorous statistical analyses (McCauley et al., 2008). This aligns with the demand for scientific rigor in cognitive psychology and HCI studies (Shneiderman, 2010).</p>	<p>Bug-Centric Focus: While the bug is undeniably a central element, it's important to note that other factors such as team dynamics, individual learning, and system limitations can also significantly impact the debugging process, even though they may not receive equal emphasis within this framework (Endsley, 1995).</p>
<p>Measurement diversity: The framework's versatility in data collection spans from quantitative metrics like tool usage frequency to qualitative insights like perceived relevance, offering diverse measurement options (Endsley, 1995).</p>	<p>Data Saturation Challenges: The framework's multiple layers might necessitate substantial time and effort to achieve data saturation, as it could demand extensive interviews and observational periods (Fusch & Ness, 2015).</p>
<p>Information flow: The bidirectional arrows symbolise the dynamic interaction among layers, depicting the flow of information and actions in debugging (Pirolli & Card, 1999).</p>	<p>Static Nature: It's important to recognize that the framework may not inherently depict changes over time. For instance, a bug initially classified as minor might evolve into a more complex issue during the debugging process, necessitating dynamic adjustments to the model (Ko et al., 2004).</p>

3.4.1 Layer 1: Debugging Environment Layer

This layer delves into the debugging ecosystem within a remote setting. It brings to the fore the complexities due to geographical distances among novice programmers,

impacting collaboration, information sharing, and task allocation. This layer contains a broad spectrum of resources accessible to debuggers, including IDEs like PyCharm and VSCode, debugging tools, code repositories such as GitHub, and online forums or documentation, supplemented by communication platforms like Slack for enhanced collaboration in distributed debugging. The environment's data points are diverse, tracking debugger interactions with resources, tool usage, access frequency and duration, and forum contributions, demonstrating a reciprocal relationship between the environment and debuggers' information foraging tendencies. The environment has obstacles in spite of its abundance of resources, which provide a variety of ways for problem-solving and dynamic engagement through information exchange. The obstacles include potential cognitive overload from the environment's complexity, uneven resource utility, and shortcomings in current techniques for capturing refined human-environment interaction. This scrutiny reveals how novice debuggers within distributed contexts navigate and collaborate, marking the facilitators and barriers encountered.

3.4.2 Layer 2: Information Foraging Layer

This layer emphasises debuggers' search for information, grounded in Pirolli and Card (1999) Information Foraging Theory. This exploration highlights how debuggers traverse the debugging environment, seeking information akin to animals foraging. It focuses on the dynamic interaction between the debugging environment and foraging behaviours, a relationship depicted through bidirectional arrows linking this layer with both the Debugging Environment and Distributed Cognition layers, as detailed by Hollan et al. (2000). The layer captures a variety of data points, including the types and numbers of information sources accessed, engagement durations, and perceived relevance, blending

quantitative and qualitative assessments to evaluate information-seeking effectiveness, following Marchionini (1995). Its strengths encompass a thorough data collection approach and flexible exchange of information, mirroring the adaptability of debuggers' information-seeking within their operational context. Nonetheless, it acknowledges obstacles like the complexity of real-time data analysis and the subjective nature of determining information relevance.

Ultimately, Information Foraging is a crucial aspect of the framework, merging the theoretical perspectives of Pirolli and Card (1999) with Hollan et al. (2000) observations on distributed cognition. This layer sheds light on debuggers' methods and challenges in sourcing and applying information and elucidates the interplay between individual and shared cognitive processes, thereby enriching discussions on debugging methodologies in distributed computing settings.

3.4.3 Layer 3: Distributed Cognition

This layer highlights the synergy of collective intelligence in debugging, rooted in the foundational works of Hutchins (1995) situated within the "Information Foraging" framework. It examines how cognitive tasks are dispersed and managed among debugger pairs, informed by Hollan et al. (2000), illustrating that cognition is a shared function extending beyond individuals to encompass a network of collaborators and tools. Metrics such as communication patterns, task distribution, and decision-making processes, as detailed by Rogers and Ellis (1994), provide an analysis of how cognitive labour is dynamically shared and executed. Also, the integration with adjacent layers depicts a seamless flow of information and cognitive activities, promoting a comprehensive view

of the debugging strategy. Moreover, its strengths lie in its facilitation of collaboration and presenting a detailed view of the interaction. At the same time, challenges include the complexity of data interpretation and the potential variability in the effectiveness of distributed cognition. This layer, therefore, serves as a critical component of the conceptual framework, synthesising insights from Hollan et al. (2000) and Rogers and Ellis (1994) to deepen understanding of collaborative debugging within distributed settings, thereby paving the way for future inquiries into the complex interplay of cognitive processes and socio-technical dynamics.

3.4.4 Layer 4: Innermost Circle: Cognitive Processes

This layer encapsulates debugger-specific mental activities, from problem comprehension to hypothesis testing and learning, grounded in cognitive theories (Anderson, 2015). It utilises methods like think-aloud protocols and, possibly, eye-tracking to explore debuggers' mental models and decision-making processes (Oh et al., 2013), demonstrating the complexity of individual cognitive efforts. Interlinking with the "Distributed Cognition" layer shows the symbiosis between individual and collective cognition in debugging (Hutchins, 1995), emphasising the significance of understanding personal cognitive tasks alongside shared efforts. Despite its comprehensive approach to cognitive analysis, the layer faces challenges like potential data collection intrusiveness and high demands on resources, suggesting a need for further methodological development. Overall, the layer significantly contributes to the framework by elucidating the individual cognitive basis of debugging within a collective context, emphasising the need for dynamic and diverse methodological approaches to fully grasp cognitive dynamics in debugging in distributed settings.

3.4.5 Centre: The Debuggers

The Distributed Pair Debugging Conceptual Framework's "Debuggers" layer is symbolised by two avatars representing the debugging participants. This layer is pivotal, emphasising human-centred design and marking where conceptual layers merge with human action, as discussed by Rogers and Ellis (1994). It facilitates an exploration of the debuggers' roles, responsibilities, and skills, highlighting the essential human element in debugging. Through this layer, a rich array of data is collected, ranging from quantitative metrics like success rates to qualitative insights on joint efforts, illustrating the symbiotic cognitive relationship between debuggers (Hollan et al., 2000).

This methodology's strengths include its ability to analyse how debuggers' traits influence debugging, acknowledging the bug's dynamic evolution and the complexity of debugging scenarios more comprehensively. This approach would better align the framework with the debugging process's realities, leveraging insights from Hutchins (1995), Endsley (1995) and Zhang and Norman (1994).

The conceptual framework integrates Information Foraging Theory (Pirolli & Card, 1999) and Distributed Cognition (Hutchins, 1995) to offer a comprehensive model for examining paired novice debuggers in distributed environments. Moving beyond the notion of debugging as merely an individual cognitive activity, it embraces the complexity of effective information foraging and the distribution of cognitive tasks among team members. This approach highlights the importance of both individual and collective cognitive efforts and their interactions within the debugging context. As such, the

framework presents a mechanism for advancing research in collaborative software development and human-computer interaction.

Employing a qualitative methodology, the framework incorporates methods, such as interviews and observations for in-depth exploration of debuggers' experiences, drawing on Creswell's (2014) emphasis on contextual richness, adhering to the empirical standards of human-computer interaction research (Card et al., 2018). This approach enables a thorough investigation of the pluralistic debugging process, particularly suited to the complexities of distributed settings. While the research predominantly employs a qualitative approach, the framework's design is inherently flexible, allowing for rigorous empirical studies.

3.5 Deployment for data collection and data analysis

To demonstrate how the conceptual framework is applied, Tables 6-8 show the connections between each layer of the framework and the specific methods utilised for data collection and analysis. Emphasis is placed on showing how each layer has directly shaped the selection of data collection methods and how these, in turn, have contributed to the insights generated during the analysis process. This detailed mapping helps to clarify the relationship between the theoretical foundation and the practical research methods employed.

Table 6: Relationship between the theoretical framework and the research methods (Layers 1 & 2)

Layer	Description	Data Collection Methods	Data Analysis Approach
Layer 1: Debugging Environment	Focuses on tools, technologies, and the remote setting in which debugging occurs.	Screen and voice recordings capture how participants interact with the tools (e.g., IDEs, version control). Logs from tools and documentation websites track which resources are accessed during debugging.	The collected data are analysed to understand the frequency and types of tool usage. Patterns such as tool-switching, reliance on documentation, or using IDE features (e.g., debuggers, version control) are tracked. These insights help assess how well the environment supports or hinders debugging efforts.
Layer 2: Information Foraging Layer	Based on Information Foraging Theory (IFT), examines how participants search for and gather information.	Screen recordings and think-aloud protocols capture the information search processes, revealing how participants look for information (e.g., documentation, StackOverflow, forums, code navigation, etc.). This layer investigates the cognitive strategy of “information foraging”.	The collected data are coded to categorise search strategies, such as direct queries or exploratory navigation. Successful foraging is identified when information retrieved directly contributes to bug resolution, while unsuccessful attempts highlight areas where further learning is required.

Table 7: Relationship between the theoretical framework and the research methods (Layers 3 & 4)

Layer	Description	Data Collection Methods	Data Analysis Approach
Layer 3: Distributed Cognition	<p>Examines how cognitive tasks are shared between the pair and technology.</p> <p>This layer captures how cognitive tasks are distributed across team members and tools, focusing on collaboration and shared understanding.</p>	Transcripts from debugging sessions and interview, focus group discussions explore how cognitive tasks (e.g., task switching, communication) are distributed between participants and across tools.	A content analysis approach maps how the pair shares cognitive responsibilities. The focus is on how tasks are allocated and communicated during debugging. Key insights regarding collaboration efficiency are drawn, such as which partner assumes leadership in specific debugging activities.
Layer 4: Cognitive Processes	<p>The innermost layer deals with each debugger's mental activities (e.g., problem comprehension and hypothesis testing).</p> <p>Focuses on individual mental activities such as problem comprehension and hypothesis formation.</p>	The think-aloud protocols and post-session interviews focus on individual cognitive processes during debugging (e.g., hypothesising bug causes and formulating solutions).	Cognitive processes are thematically coded into categories such as problem-solving tactics (e.g., trial-and-error, hypothesis testing). The analysis also tracks shifts in cognitive load and mental strategies as the debugging session progresses.

Table 8: Relationship between the theoretical framework and the research methods (Centre Layer)

Layer	Description	Data Collection Methods	Data Analysis Approach
The Debuggers (Centre Layer)	<p>The roles and interactions of the two participants focus on their collaboration and individual contributions.</p> <p>This is the core of the framework, symbolising their human-centred activity.</p>	<p>Demographic surveys and performance logs collect metrics on individual debuggers, such as experience, expertise, and bug resolution performance.</p> <p>Demographics of participants (skills, experience). - Dyadic performance metrics.</p>	<p>Combining performance metrics and qualitative insights (from interviews and focus groups) helps explore individual contributions and teamwork dynamics.</p> <p>Statistical analysis can also be applied to debugging success rates.</p>

Data Collection: Tables 6-8 illustrate how different aspects of the conceptual framework are tied to specific data collection methods. Screen recordings and think-aloud protocols provide insight into the debugging environment and cognitive processes layers. Interviews and focus groups also gather data on distributed cognition and individual contributions.

Data Analysis: Each layer of the conceptual framework informs distinct parts of the data analysis process. For example, the Information Foraging Layer drives thematic analysis of search patterns, while Distributed Cognition focuses on the content analysis of communication and task allocation. The Debuggers (Centre) layer integrates both quantitative (performance metrics) and qualitative (collaboration dynamics) insights.

Within this context, the study adopts a rigorous approach by systematically linking each phase of the conceptual framework to the corresponding data collection and analysis methods. This alignment ensures a focused exploration of how novice programmers operate within a distributed debugging environment. Central to this process are the tables provided (Tables 6-8), which serve as crucial tools for establishing explicit connections between the theoretical constructs underpinning the framework, such as Information Foraging Theory and Distributed Cognition, and their practical application during the data collection and analysis stages.

Furthermore, this structured mapping ensures that the conceptual elements transcend theoretical abstractions, being operationalised in a way that directly guides the research process. By deconstructing the debugging process into its fundamental components, the framework enables a systematic and thorough analysis of novice programmers' behaviours and cognitive strategies in distributed settings.

In addition, this approach illuminates the interactions between individuals and their tools and provides critical insights into how these factors influence collaboration, problem-solving, and the overall efficiency of the debugging process. Ultimately, deploying this conceptual framework enhances the clarity, depth, and relevance of the study's findings in understanding the complexities of distributed debugging for novice programmers.

3.6 Summary

This study employs Information Foraging Theory and Distributed Cognition to analyse the debugging process, particularly focusing on novice programmers in networked environments. Information Foraging Theory (Pirolli & Card, 1999) assesses how novices navigate and extract valuable information, with previous studies validating its application in understanding software maintenance challenges (Fleming et al., 2013; Lawrence et al., 2008; Piorkowski et al., 2012). Distributed Cognition extends the analysis beyond individual cognition to include social and technological interactions, impacting fields from aviation to software development (Hollan et al., 2000; Hutchins, 1995). This chapter explores explicitly how these frameworks apply to novice programmers working in pairs in distributed settings, aiming to illuminate the collaborative aspects of debugging and the role of tools in this context.

Chapter 4: Methodology

4.0 Introduction

This chapter describes the research's methodological structure, including key principles and strategic direction. It begins by presenting the chosen research paradigm, tailored to unravel the complexities of the subject, thereby identifying the need for a solid foundation for inquiry. Following this, the chapter explores the specific methods and research designs employed, leading to detailed scrutiny of the theoretical foundations and highlighting methodological integrity's paramount importance. By evaluating these methodological aspects, the chapter strives to carve out a definitive path for the research endeavour, aiming to augment the scholarly landscape significantly.

4.1 Research Question

Drawing on existing studies, debugging is acknowledged as an integral, yet time-intensive component of software development projects (Beller et al., 2018; Zhao et al., 2008), necessitating programmers to identify and rectify software glitches and look deeply into the intricate architecture of the software (Oman et al., 1989; Perscheid et al., 2017). Unfortunately, there is a noticeable lack of studies on how novice programmers debug code while working together, whether in the same location or across diverse geographies.

Additionally, there is a reasonable tendency to tailor existing industry methods to fit educational environments. Nevertheless, the changes in circumstances, particularly with novice programmers working together from distant locations, require careful analysis.

To this end, this central question steers this study:

“How do the paired Software Development Apprentices in geographically distributed locations work collaboratively to fix Python programming bugs using the technology-mediated medium?”

However, according to Leedy and Ormrod (2021), a researcher from a design standpoint splits the central question into several smaller questions where the outcome of each smaller question can possibly answer the central question. So, given this and in investigating this central question, this study proffers answers to the following five specific research questions:

- **RQ₁:** What bugs are generated by the paired geographically distributed SDT apprentices working collaboratively to solve a given problem using Python?
- **RQ₂:** What bug locating strategies and tactics are deployed by the paired geographically distributed SDT apprentices while attempting to fix defects in the given Python code? How do they go about finding the bugs in the program code?
- **RQ₃:** How do the paired geographically distributed SDT apprentices distribute cognitive load when resolving bugged code?
- **RQ₄:** How does leveraging Integrated Development Environment (IDE) tools enhance the capabilities of distributed pair debugging and mitigate the challenges encountered in debugging programs?
- **RQ₅:** What challenges are experienced by paired geographically distributed SDT apprentices working collaboratively on debugging programming bugs, and why are they facing such challenges?

Building upon the premise established by Maxwell (2012) that research questions hone the focus of a study; this research primarily seeks to identify the bugs produced by dyad SDT apprentices working remotely on Python tasks. Given their novice status, the study aims to discern patterns or similarities in the bugs they produce, especially compared to collocated novices. The goal is also to ascertain if being in remote dyads influences the type of bugs, especially given the documented challenges faced by remote teams, such as collaboration, productivity, and communication issues (Miller et al., 2021; Neufeld & Fang, 2005; Ralph et al., 2020). By addressing this question, the study endeavours to draw parallels in the bugs from each dyad and cross-reference them with existing literature on bugs from solo and collocated novice programmers, shedding light on potential bug causatives in remote novice settings.

Following the elucidation of the types of bugs encountered, understanding how dyad SDT apprentices, situated remotely, identify and rectify these programming inconsistencies becomes paramount for this study. Consequently, the inquiry encapsulated in the second research question is instrumental in unravelling the debugging behaviours. Building upon any insights garnered about debugging behaviours; the third research question aims to understand how dyad SDT apprentices distribute cognitive load during their bug-searching and fixing endeavours. This question employs a verbal protocol to vocalise thoughts, thereby shedding light on the manifestation of distributed cognition within the dyad (Hutchins, 1995). In parallel, this exploration delves into how thought processes inform specific actions undertaken while pursuing bugs in the programming code, aligning with the premises of information foraging theory (Pirolli & Card, 1999). The combination of distributed cognition and information foraging theory forms a key framework, which

seeks to capture the complex cognitive processes and detailed debugging methods, thus shaping the direction of data gathering and analysis.

Expanding upon previous research, the fourth research question delves into the intricate relationship between technology and debugging. Specifically, it investigates the roles of integrated development environments, compilers, and synchronous collaboration tools in potentially streamlining the bug foraging and rectification processes. The inquiry is centred on how SDT pairs utilise these technologically-mediated tools to potentially enhance distributed pair debugging capabilities, thereby mitigating challenges encountered during debugging. While evidence suggests that technological tools have been instrumental in collocated dyad debugging scenarios, this research seeks to investigate if their impact remains consistent or introduces new facets to the remote debugging process.

Building on the exploration of technology's role, the fifth research question focuses on the challenges remote dyad apprentices encounter during their synchronised and collaborative efforts to debug Python codes. This inquiry extends beyond merely identifying the challenges, aiming also to uncover the underlying causes behind such difficulties in collaborative debugging scenarios.

4.2 Context and Study Site

Drawing from the insights of Dey (2001), which emphasises the vital role of context in shaping an entity's implicit situational information, whether that pertains to an individual, location, or object, it becomes clear that empirical studies are intricately bound to the

inherent nature of human behaviour. As Van Oers (1998) articulated, this context-driven behaviour aids in refining specific meanings, ensuring they are holistically intertwined within a broader spectrum rather than isolated instances.

Expanding upon this, the current study zeroes in on distinct contexts involving 30 SDT apprentices dispersed across twelve diverse organisations. The primary focus revolves around the debugging strategies of 15 dyad apprentices, who, as novice programmers, operate under the guidance of workplace mentors. As characterised by Bonar and Soloway (1983) and reaffirmed by Lau and Yuen (2009) and Jenkins (2002), these novices stand at the initial stages of programming, often lacking expertise in crucial areas such as problem-solving, abstraction, and, notably, debugging. Also, despite a plethora of reasons pinpointed for the debugging struggles of novice programmers (de Raadt, 2007; Denny et al., 2022; Lahtinen et al., 2005; McCauley et al., 2008; Vourletsis et al., 2021; Whalley et al., 2021), mapping out these patterns could further help understand their challenges.

It is imperative to highlight that the study's milieu was predominantly digital, leveraging technology-driven platforms like Microsoft Teams and specialised debugging software. In this regard, Visual Studio Live Share, commonly referred to as "Live Share", serves as the specialised debugging software and is a collaborative development tool introduced by Microsoft for Visual Studio and Visual Studio Code. This extension empowers apprentices to share synchronously and co-edit code with peers, fostering joint coding, debugging, and issue resolution. This eliminates the necessity for participants to share a local network or identical development configurations. As a result, Live Share offers a fluid co-

coding platform, proving indispensable for apprentices collaborating from distant or varied locations.

Also, this multi-layered study spans a wide range of elements, from individual cognitive aspects and technological infrastructures to the intertwined socio-technical dynamics related to optimal debugging tool usage. Similarly, the research also inquires into the interaction between external (software tools and share code) and internal representational (mental models, problem-solving strategies, knowledge base) frameworks. In essence, this aspect of the research attempts to bridge the gap between the tangible tools and methods used in debugging and the intangible cognitive processes programmers employ, especially in the context of collaborative, distributed environments. This complex interplay includes discussions in pairs using the think-aloud protocol, the debugging patterns of the SDT, joint efforts in addressing software bugs, how the use of a particular debugging platform shapes or guides an apprentice's internal thought process or problem-solving strategy, and the specific code being examined.

4.3 Philosophical Perspectives of this Study

Beginning with the foundational principles posited by Lincoln et al. (2011) and Cresswell and Plano Clark (2011), it is evident that a researcher's philosophical leanings and worldviews deeply inform every facet of the research process, especially concerning the origins and nature of knowledge. These predispositions hold tangible ramifications. Thus, a lucid understanding of one's philosophical principles becomes indispensable, offering a robust foundation to delve into the study's paradigm, ontology, epistemology, and methodology, as elucidated by Fitzgerald and Howcroft (1998).

Building on this idea, this study's approach is deeply influenced by embedded philosophical perspectives, as highlighted by Creswell and Poth (2018) and Crotty (1998). These perspectives inform the research questions and data collection methods while supporting the study's paradigm, ontology, epistemology, and methodology. According to Saunders et al. (2019), research philosophy acts as a belief system that critically informs the methodology, strategy, and analysis of data, reflecting the interplay between a researcher's philosophical stance and their investigative approach. The study navigates the objectivism-subjectivism continuum, recognising the dichotomy between viewing reality as an external, observable entity and understanding it as a socially constructed mosaic. This philosophical grounding provides a robust foundation for exploring the specific research strategies and analytical frameworks employed in this study. The position of this research, in relation to these philosophical underpinnings, is further elaborated in subsequent sections.

4.3.1 Paradigm

This study adopts an interpretive paradigm, conceptualising it as a set of philosophical assumptions about the nature of reality and methods to understand it, as suggested by Mittwede (2012) and elaborated by Christensen et al. (2020) and Creamer (2017). This paradigm serves as a lens through which the research on SDT apprentices' debugging strategies is viewed, aligning with Kuhn (1970) interpretation of paradigms as collective exemplars that influence evidence collection. Within this framework, the study embraces the comprehensive paradigm dimensions, ontology, epistemology, methodology, and

axiology, as described by Guba and Lincoln (1994), which dictate diverse perspectives on knowledge and its formation.

Concluding, this interpretive approach enables a deep exploration of the human aspects of software debugging, focusing on apprentices' experiences, behaviours, and perceptions. By understanding these elements, the research provides qualitative insights into the apprentices' interactions and learning processes in debugging within a collaborative environment. This aligns with Cohen et al. (2007), who advocate for the interpretive paradigm's utility in examining complex human behaviours and social interactions, thus offering a detailed perspective of educational and professional practices in technological settings. Further details on the study's paradigm position are explored in subsequent sections.

4.3.2 Ontology

Ontology stands out as a crucial dimension, encapsulating philosophical assumptions pertinent to the nature of truth and reality. Connecting these ontological perspectives to the current study, the interpretive paradigm is utilised, encapsulating the belief in "reality as socially and discursively constructed by human actors" (Grix, 2004, p. 61). From an ontological standpoint, the study asserts the pluralistic nature of reality, suggesting diverse experiences and approaches among SDT apprentices in program debugging. Consequently, acknowledging diverse experiential worldviews, the study is geared towards exploring multiple realities (Lincoln & Guba, 2000), wherein each apprentice constructs meaning through interactions and engagements (Bryman, 2016). This approach is proposed by Guba and Lincoln (1994), who propose that relativism serves as

the ontology for interpretivism, advocating the subjective and individualistic perception of reality.

Given these considerations, this study adheres to a pluralist view of reality, ensuring a harmonious alignment of the adopted ontology with the epistemological perspective and, consequently, influencing the research design.

4.3.3 Epistemology

This research aligns with the perspective that epistemology is intertwined with assumptions analysing the relationship and dependencies between the researcher and the research focus, affecting the objectivity and detachment inherent in research processes (Creswell & Poth, 2018; Leavy, 2017).

In the context of this study, the epistemological perspective of qualitative research implies a substantial investment of time in engaging with participants to gain insights through detailed descriptions of their lived experiences and viewpoints. It emphasises the co-creation of knowledge and subjective reality, considering the influence of social interactions and the researcher's interpretations of contextual actions. In this light, a deeper understanding of knowledge and meaningful reality will be attained regarding the approaches of paired SDT apprentices in debugging Python's bugged code within specific social settings facilitated by interaction with technology agents (Guba & Lincoln, 1994). However, the richness of the interpretive paradigm's descriptions is juxtaposed with challenges in validity and trustworthiness, stemming from the subjective nature of the data and varying participant interpretations (Rolfe, 2006). In order to tackle these

potential vulnerabilities, this research incorporates Maxwell's strategies for addressing validity concerns (Maxwell, 2008), laying a robust foundation for the research effort (see Section 4.7).

4.4 Methodological Framework

This study aligns with a qualitative research methodology, drawing from the interpretive paradigm to explore the debugging behaviours of SDT apprentices in distributed settings. This choice is underpinned by the study's ontological belief in the subjective construction of reality and its epistemological stance that knowledge is best understood through interpreting these subjective experiences.

Leedy and Ormrod (2021) emphasise research as a process that goes beyond mere data collection to include deep analysis and interpretation to enrich understanding of a specific phenomenon. This perspective shapes the research methodology, which, as Cameron (2011) and Brannen (2005) articulate, is inherently linked to the researcher's ontological and epistemological assumptions. These assumptions inform the choice of qualitative research for this study, which seeks to capture apprentices' complex, intricate interactions with their work environments.

According to Leavy (2017), research methodology involves harmonising methods and theoretical frameworks guided by underlying philosophical convictions. This approach is vital for understanding apprentices' subjective and constructed realities as they navigate debugging tasks, making qualitative methods particularly suitable. Gray (2021) and Saunders et al. (2019) further argue that the choice of methodology influences the

research design, which in this case focuses on multiple case studies to provide in-depth insights into each apprentice's experiences and interactions within natural settings.

While quantitative research offers a systematic exploration of variables and mixed methods provide a comprehensive blend of quantitative and qualitative data, the qualitative approach was chosen for its strengths in generating rich, contextual, and detailed narratives (Christensen et al., 2020; Gray, 2021). Such depth is necessary to grasp the full scope of apprentices' debugging experiences and the dynamic, often tacit aspects of their skill development in real-world contexts.

Therefore, this study's methodological framework does not isolate it within a single paradigm but reflects a pragmatic blending of influences that supports its goals. It utilises a multiple case study approach as described by Merriam (1998) and Yin (2014), which allows for examining the 'how' and 'why' behind apprentice behaviours in natural settings, thereby aligning the philosophical underpinnings with the practical inquiry methods. This alignment ensures that the research is methodologically sound and deeply reflective of the interpretive paradigm's focus on understanding human experiences within their naturally occurring contexts.

4.4.1 Case study design and rationale

The qualitative case study methodology is highly suited to the SDT distributed pair debugging research due to its ability to provide in-depth insights into complex processes and interactions within specific real-life contexts (Creswell, 2014; Merriam, 2009; Yin, 2014). This approach is invaluable for comprehending the complexities of social

interactions, structures, and the debugging processes that SDT apprentices engage in, enabling researchers to capture the intricate details of how and why certain behaviours and practices occur (Baxter & Jack, 2008; Creswell, 2014).

A key strength of the qualitative case study lies in its contextual sensitivity, which allows for a detailed examination of the environmental, temporal, and locational factors that influence apprentices' debugging practices. This sensitivity is essential for understanding the complex dynamics between paired SDT programmers and how external variables, such as technological agents, impact their debugging strategies (Gray, 2021; Geertz, 1973). Such a methodological approach is critical for generating deep insights into the interactions and dependencies within the debugging environment (Ridder, 2017).

Furthermore, the holistic nature of qualitative case studies supports the integration of multiple data sources, enhancing the robustness and comprehensiveness of the analysis. This capability is indispensable for exploring various dimensions of the debugging process, allowing researchers to draw meaningful correlations and interpretations vital for theoretical and practical advancements (Gerring, 2017; Stake, 1995).

While qualitative case studies offer significant theoretical contributions and facilitate the exploration and conceptualisation of new paradigms, their specificity and contextual depth may limit the generalisability of findings. However, the richness of the collected data compensates for these limitations, providing detailed, context-specific insights crucial for understanding the unique phenomena of distributed pair debugging (Creswell, 2014; Merriam, 2009; Ridder, 2017).

In sum, despite potential challenges such as resource intensiveness and issues with generalisability, the qualitative case study methodology aligns effectively with the SDT distributed pair debugging research goals. It enables a dynamic and adaptable exploration of processes, which is essential in settings characterised by rapid technological and procedural changes (Flyvbjerg, 2006; Saunders et al., 2023). Also, the depth and adaptability of this approach ensure that it supports the development of practical solutions tailored to the specific needs and contexts of SDT apprentices.

4.4.2 Sampling

Qualitative research inherently focuses on depth rather than breadth, aiming for rich insights over broad generalisations. Nevertheless, this focus does not negate the need for carefully crafted sampling strategies. Indeed, rigorous sampling is pivotal to ensure data validity and to address key research questions effectively, a process critical to deriving meaningful interpretations (Flick, 2022; Patton, 2015; Saunders et al., 2019). Moreover, developing a suitable sampling frame for case studies is complex, demanding a careful balance between study objectives, seeking richness over range, and the careful application of findings (Creswell, 2014).

Keeping this in perspective and in the context of a qualitative multiple case study focusing on dyads of apprentices debugging Python code in distributed settings, the choice of purposive sampling is a deliberate and strategic methodological decision. The employment of purposive sampling in this study enables the deliberate selection of cases that facilitate an investigation into the dynamics of collaboration and cognition among

apprentices in distributed settings. This methodological choice is instrumental in capturing rich, multifaceted interactions and the evolving cognitive processes that characterise the apprentices' experiences as they engage in debugging Python code together. As Flick (2022) suggests, qualitative research should not default to random sampling as in quantitative studies, but instead should employ a thoughtful approach to select participants, ensuring the data's richness and relevance to the research questions.

In this specific study, purposive sampling was employed to select apprentice pairs who could provide insights into the debugging process. This selection was driven by the intent to understand the individual actions and the interpersonal dynamics and communication patterns that might facilitate problem-solving in a distributed setting (Lincoln & Guba, 1985). The iterative nature of purposive sampling, inclusive of snowball, quota, and convenience sampling methods, allowed for a layered and rich collection of data, contributing to a desired well-rounded understanding of the case (Patton, 2015).

In sum, the purposive sampling method was integral to the research design, ensuring that the cases chosen for this study were informative and closely related to the central research questions. This methodological choice, underpinned by scholarly discourse, provided a framework for examining the collaborative interactions of apprentices in distributed settings, ultimately leading to findings that can be both insightful and trustworthy.

4.4.3 Participants

This study categorises its participants into two main groups, in particular, SDT apprentices and workplace mentors and trainers from the training organisation, each offering critical insights into the research objectives. The central focus of this investigation is on the SDT apprentices, the first participant category. Their engagement in debugging bugged Python code is vital for understanding various aspects, such as their debugging strategies, the role of technology in this process, how they manage cognitive load during collaborative debugging, and the challenges they face in this context (Patton, 2015). This aspect of the study is crucial in revealing both the individual and collaborative dimensions of their software development skills.

Prior to initiating the recruitment of apprentices for this study, the necessary ethical approval was acquired, reflecting stringent adherence to academic research protocols (Creswell, 2014). This foundational step was followed by an extensive outreach effort, wherein 110 emails were dispatched to a selection of organisations known for fostering apprentices at the targeted standard. As detailed in Appendices A to D, these emails introduced the study's aims and enclosed essential documentation, including Participant Information Sheets (PIS) and consent forms for employers and apprentices, ensuring informed participation (Saunders et al., 2023). Key criteria outlined in the emails included the specific age bracket, the necessity for apprentices to fall within the novice programmer classification, and a commitment to contribute a maximum of four hours throughout the study.

From the 135 emails disseminated using DocuSign, around 45 organisations expressed their willingness to participate, encompassing a total of 89 SDT apprentices who were available for the study duration, along with their workplace mentors. Of the 89 SDT apprentices, 58 apprentices completed and returned the necessary consent and survey forms.

Upon examination of these forms and cross-referencing the apprentices' profiles in terms of educational background and programming experience, a cohort of 46 apprentices was ultimately selected. This selection process, unfortunately, led to the exclusion of 12 candidates who did not meet the set criteria, thereby reducing the number of participating organisations to 36. The study targeted apprentices who had been part of the training programme for over three months but less than nine, ensuring they possessed basic programming knowledge per the SDT Standard.

Acknowledging that most of the apprentices were unfamiliar with each other and came from varied organisational backgrounds, a 30-minute familiarisation debugging session was organised. This session, not formally part of the study, was crucial for the apprentices to practice the think-aloud protocol while engaging in collaborative debugging. It allowed them to understand what participation entailed and assess their willingness to continue in the study. Subsequently, 11 apprentices withdrew, reducing the number of participants to 35, all volunteered for the study. Among these, 30 were actively paired for the study, while 5 were placed on standby, ready to step in should any shortlisted apprentices withdraw. It should be noted that the recruitment happened on two different occasions due to a break in the study.

Thirty apprentices participated in the study, each randomly paired within their age group. They were anonymised using shorthand notation ('STD<number>') and briefed, spanning ages 16 to 50 years (see Table 6 for the participants' details for the debugging sessions and the dyads' interviews). Predominantly, these participants were young males, primarily falling within the 16 to 21 year age bracket, highlighting the study's focus on a younger demographic.

The recruitment criteria for apprentices in this study emphasise a foundational background in software development, with formal education being essential. Typically, participants are expected to have completed secondary education and introductory programming courses, ideally in Python, to equip them with the skills necessary for debugging. This foundation helps ensure that participants are not overwhelmed by the complexity of the debugging tasks (Robins et al., 2003). The study categorises apprentices as novice programmers, grouped into three subgroups based on age. Participants aged 16 to 18 years must have no more than two years of programming exposure, just transitioning from secondary education into software development and hold a General Certificate of Secondary Education (GCSE) level qualification or equivalent Level 2 qualification on the national occupational framework. Those aged 18 to 25 years should have less than one year of programming experience, ideally with Level 3 qualifications, while those aged 25 to 50 years should have three to nine months of hands-on experience. This structured approach ensures that participants have enough exposure to contribute meaningfully to the debugging process while still encountering the challenges typical of novices (Allwood, 1986).

In addition to educational qualifications, the study required apprentices to demonstrate a commitment to the debugging sessions and interviews, dedicating up to five hours for participation. Their readiness to collaborate was equally vital, as the study focused on paired debugging and think-aloud protocols, exploring how apprentices communicated and shared cognitive loads in real-time problem-solving situations. This collaborative approach ensured that participants contributed effectively to the study's objectives. Although apprentices were expected to have basic Python knowledge, their experience was still developing. As outlined in Table 9, the recruitment process aimed to select individuals with the appropriate educational background, experience, and willingness to engage in collaborative work, contributing to the study's success.

Despite the variations in age and background, it is crucial to recognise that all apprentices were uniformly classified as novice programmers. This classification reinforces the study's objective to evaluate individuals' learning and developmental trajectories at the nascent stages of their careers in software development, thereby contributing to the field of programming education and research.

Table 9: Participant details for the debugging sessions and the dyad's interview

Dyad ID	Participant ID	Age Bracket	Gender	Programming experience
Dyad1	SDT1	16 – 18 years	Female	Low < 2 years
	SDT2	16 – 18 years	Female	Low < 2 years
Dyad2	SDT3	16 – 18 years	Male	Low < 2 years
	SDT4	16 – 18 years	Female	Low < 2 years
Dyad3	SDT5	16 – 18 years	Male	Low < 2 years
	SDT6	16 – 18 years	Male	Low < 2 years
Dyad4	SDT7	16 – 18 years	Male	Low < 2 years
	SDT8	16 – 18 years	Male	Low < 2 years
Dyad5	SDT9	16 – 18 years	Male	Low < 2 years
	SDT10	16 – 18 years	Male	Low < 2 years
Dyad6	SDT11	16 – 18 years	Male	Low < 2 years
	SDT12	16 – 18 years	Male	Low < 2 years
Dyad7	SDT13	16 – 18 years	Male	Low < 2 years
	SDT14	16 – 18 years	Male	Low < 2 years
Dyad8	SDT15	16 – 18 years	Female	Low < 2 years
	SDT16	16 – 18 years	Female	Low < 2 years
Dyad9	SDT17	18 – 25 years	Male	Low < 1 year
	SDT18	18 – 25 years	Male	Low < 1 year
Dyad10	SDT19	18 – 25 years	Male	Low < 1 year
	SDT20	18 – 25 years	Female	Low < 1 year
Dyad11	SDT21	18 – 25 years	Male	Low < 1 year
	SDT22	18 – 25 years	Male	Low < 1 year
Dyad12	SDT23	18 – 25 years	Male	Low < 1 year
	SDT24	18 – 25 years	Male	Low < 1 year
Dyad13	SDT25	25 – 50 years	Male	Low >3 Months and <9 Months
	SDT26	25 – 50 years	Male	Low >3 Months and <9 Months
Dyad14	SDT27	25 – 50 years	Male	Low >3 Months and <9 Months
	SDT28	25 – 50 years	Male	Low >3 Months and <9 Months
Dyad15	SDT29	25 – 50 years	Male	Low >3 Months and <9 Months
	SDT30	25 – 50 years	Male	Low >3 Months and <9 Months

This study also incorporated a second category of participants, comprising Workplace Mentors and Trainers from the training organisations, whose contribution was crucial to the research's success. These professionals, with their extensive background in software development, bring a wealth of expertise and knowledge, particularly in grasping the intricacies of debugging strategies and how novice programmers, like the apprentices in this study, approach code debugging (Glesne, 2016; Patton, 2015). Their deep insights

into the apprentices' debugging processes and developmental stages are invaluable for comprehensively analysing their debugging strategies.

To ensure the relevance and value of their contribution, strict qualifications were set for the Workplace Mentors and Trainers. These experts are required to have a minimum of ten years of programming experience, which underlines their deep understanding and mastery of the field. Additionally, they should have worked with at least ten apprentices, ensuring they possess technical expertise and practical experience mentoring novice programmers. This prerequisite is essential as it guarantees that the mentors and trainers can offer detailed insights into the apprentices' debugging abilities, their software development methodologies, and the application of technology in these processes. Such depth of understanding is crucial for meeting the study's aims and adds significant value to the research objectives (Lincoln & Guba, 1985).

To summarise, the involvement of both SDT apprentices and seasoned Workplace Mentors and Trainers creates a rich and diverse pool of participants (see their demographic infographics showing their classification in Figure 5), thereby enhancing the study's depth and breadth. Through their combined perspectives and experiences, the study aimed to make contributions to the understanding of debugging strategies deployed by novice programmers and practices in software development. The insights gained from these two groups are expected to be instrumental in advancing knowledge in the domain, particularly regarding the apprentices' debugging skillsets and the role technology plays in the field.

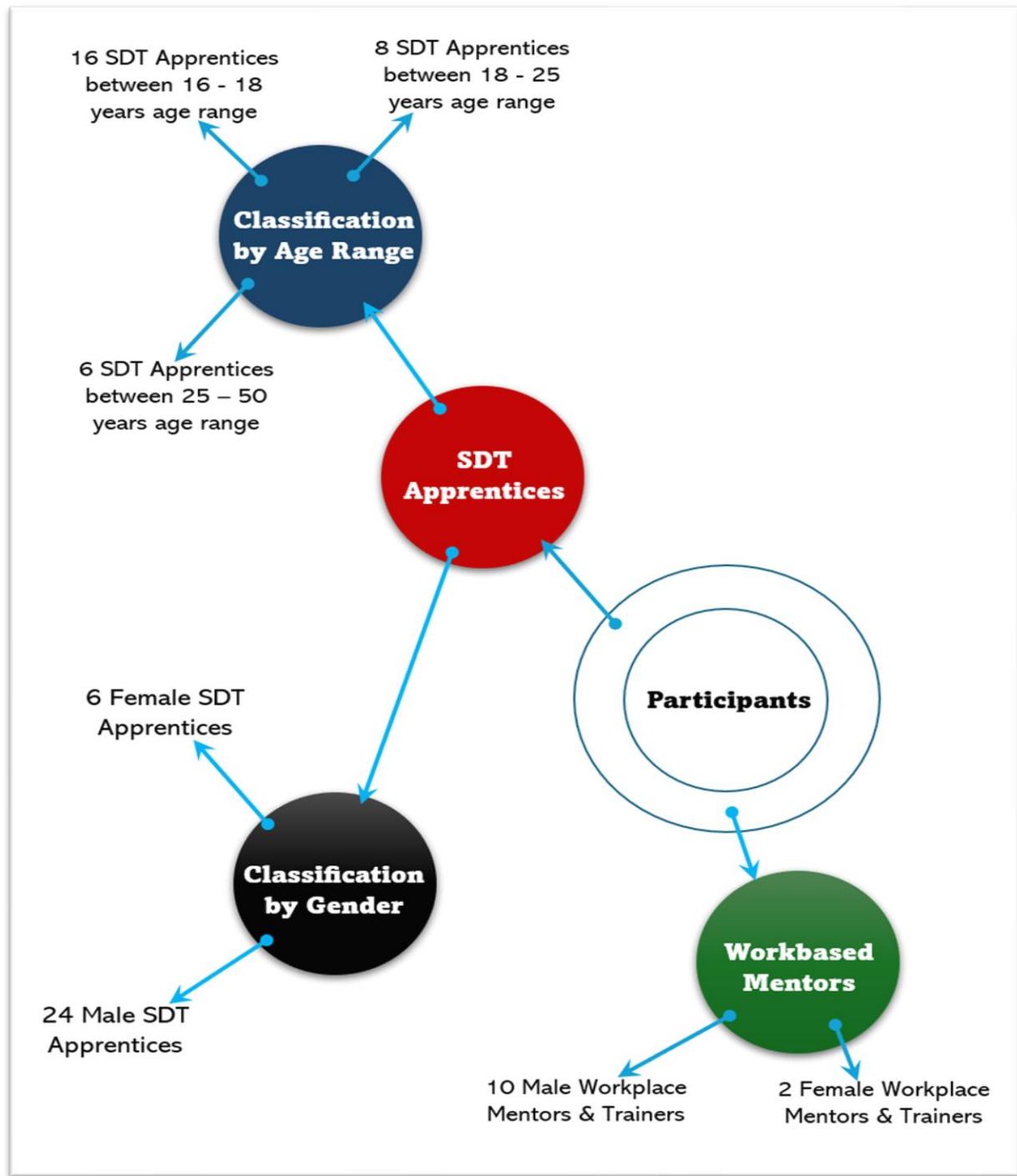


Figure 5: Participant demographic infographics recruited for the study.

4.4.4 Data Analysis

Reliable qualitative research pivots on comprehensive data analysis (Maguire & Delahunt, 2017). Considering this context, this study adopts Merriam's analytic inductive approach,

which encourages the simultaneity of data collection and analysis in such a way that the data collection and analysis are to be approached in a concurrent and interactive process (Merriam, 1998). By so doing, it is a recurring process involving consolidating, reducing and interpreting the data and making sense of it.

For the analysis of verbalisation, thematic analysis based on Braun and Clarke (2006) was employed. This process involved coding the data, organising it into categories and themes, and systematically interpreting the findings in a sequential manner.

First, the verbalisation through the usual conversation and the thought process to be exposed through the thick-aloud protocol were transcribed and annotated with the actions visible in the video. Codes were developed and guided by the principles of information foraging theory based on the steps taken to different bug location strategies, and the cognitive burden sharing and the affordances of the technology were considered simultaneously.

To strengthen the data validation to a great extent, each data source within each setting was analysed, triangulated, and converged across settings to understand the similarities and differences between the settings. In this respect, the overall understanding of the cases was established, and the data validity was enhanced. Also, three types of textual data were collected for the data sources. For example, interview transcripts, focus group transcripts, and observational field notes were transcribed and imported into NVivo to help with the organisation by coding to extract themes (Welsh, 2002).

Table 10 provides evidence triangulation, mapping the research questions to the data source and the methods of evidence collection.

Table 10: Evidence triangulation.

Research Question	Data Source(s)	Evidence collection methods
What bugs are generated by the paired geographically distributed SDT apprentices working collaboratively to solve a given problem using Python?	<ul style="list-style-type: none"> ▪ Journal articles & conferences ▪ Apprentices ▪ Workplace mentors ▪ Programming codes ▪ Recorded videos ▪ Compiler reports 	<ul style="list-style-type: none"> ▪ Literature review ▪ Observation ▪ Interview ▪ Focus group ▪ Software artefacts
What bug locating strategies and tactics are deployed by the paired geographically distributed SDT apprentices while attempting to fix defects in the given Python code?	<ul style="list-style-type: none"> ▪ Journal articles & conferences ▪ Apprentices ▪ Workplace mentors 	<ul style="list-style-type: none"> ▪ Literature review ▪ Observation ▪ Interview ▪ Focus group
How do the paired geographically distributed SDT apprentices distribute cognitive load when resolving bugged code?	<ul style="list-style-type: none"> ▪ Journal articles & conferences ▪ Apprentices ▪ Recorded videos 	<ul style="list-style-type: none"> ▪ Literature review ▪ Observation ▪ Interview ▪ Software artefacts
How does leveraging IDE tools enhance the capabilities of distributed pair debugging and mitigate the challenges encountered in debugging programs?	<ul style="list-style-type: none"> ▪ Journal articles & conferences ▪ Apprentices ▪ Workplace mentors ▪ Programming codes ▪ Record videos ▪ Compiler reports 	<ul style="list-style-type: none"> ▪ Literature review ▪ Observation ▪ Interview ▪ Focus group ▪ Software artefacts
What challenges are experienced by paired geographically distributed SDT apprentices working collaboratively on debugging programming bugs, and why are they facing such challenges?	<ul style="list-style-type: none"> ▪ Journal articles & conferences ▪ Apprentices ▪ Workplace mentors ▪ Professional codes ▪ Recorded videos ▪ Compiler reports 	<ul style="list-style-type: none"> ▪ Literature review ▪ Observation ▪ Interview ▪ Focus group ▪ Software artefacts

This qualitative multiple case study examines dyadic SDT apprentices debugging Python code through a think-aloud protocol, supplemented by in-depth interviews and focus group discussions with mentors. Thematic analysis, as outlined by Braun and Clarke (2006), is effectively adapted and particularly suitable in this context, with each stage interlinked to build a comprehensive understanding of the data. Figure 6 visually represents the data analysis stages, with the subsequent sections detailing the processes undertaken.

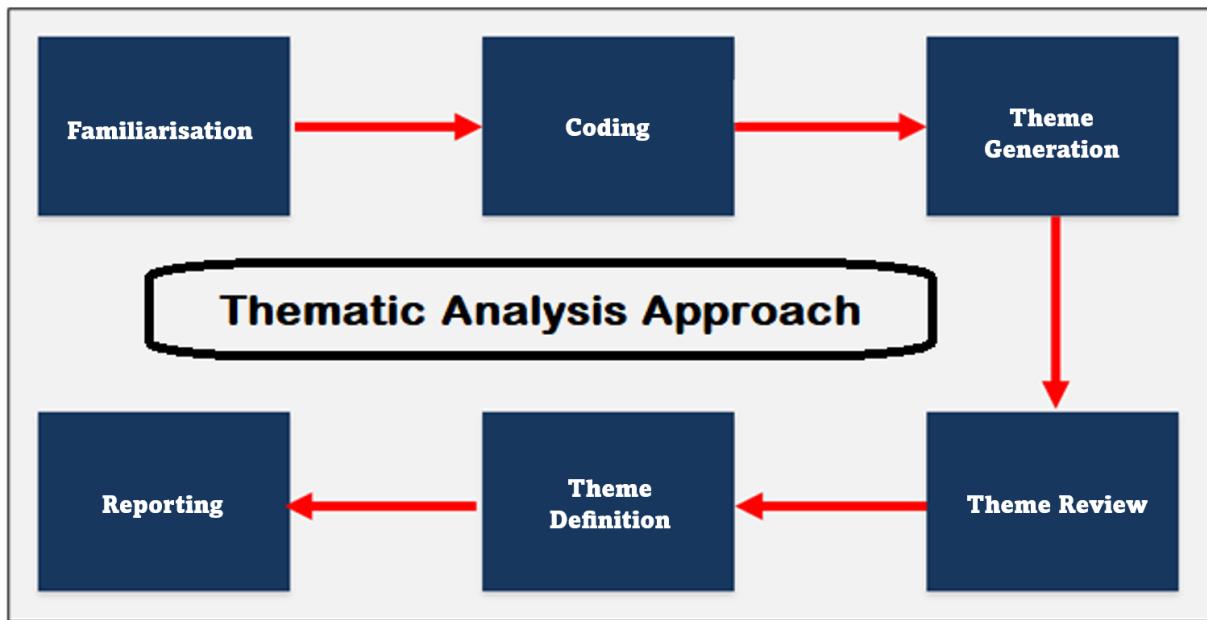


Figure 6: The thematic analysis approach adapted from Braun and Clarke (2006).

Stage 1: Familiarisation

Familiarisation with the data is a fundamental stage in thematic analysis, particularly in qualitative research involving think-aloud sessions, interviews, and focus groups. This stage provides the foundation for subsequent analysis by ensuring that all insights remain anchored in the authentic experiences of participants. The process began with transcription, capturing spoken words, pauses, intonations, and non-verbal cues to preserve the richness of the data, as underscored by Riessman (2008) and Braun and Clarke (2006). The transcriptions were then imported into NVivo, which played a crucial role in organising, managing, and systematically exploring the data, allowing for a structured approach to the analytical process.

Following transcription, repeated reading of the transcripts facilitated immersion in the data, moving beyond basic comprehension to uncover deeper context, nuances, and meanings. This active engagement, marked by questioning, annotating, and noting initial

thoughts, revealed recurring themes, distinctive expressions, and key participant reactions. These emerging patterns, which highlighted problem-solving strategies and interactions with technology, informed the direction of coding and theme development, as advocated by Braun and Clarke (2006) and Saldaña (2015). Furthermore, NVivo enhanced the process by enabling precise tagging and refinement of patterns, ensuring a critical and reflective approach. Together, these stages provided a robust and systematic foundation for thematic analysis, ensuring a comprehensive and data-driven exploration of the participants' perspectives, as emphasised by Riessman (2008), Braun and Clarke (2006), and Saldaña (2015).

Stage 2: Coding

Building on Step 1: Familiarisation with the Data, the second stage of thematic analysis, as outlined by Braun and Clarke (2006), focused on generating initial codes. This process systematically organised data from think-aloud protocols, interviews, and focus groups into meaningful units. NVivo played a pivotal role in this stage, providing the tools to systematically tag, group, and visualise data. The software's query functions and categorisation features streamlined the identification of key aspects of apprentices' experiences and strategies, aligning with the insights of Gibbs (2007) and Miles, Huberman, and Saldaña (2014).

The initial segmentation of datasets by source was critical in adhering to the multiple case study framework discussed by Eisenhardt (1989). This segmentation facilitated a detailed understanding of each dataset, enabling the identification of unique themes and patterns. NVivo's analytical tools enhanced this process by supporting annotations,

detailed queries, and data visualisations. These features helped uncover apprentices' problem-solving strategies and interactions with technology, a process underscored by Saldaña (2013) as essential for extracting meaningful insights.

Furthermore, NVivo supported comparative analysis by enabling the integration of separately coded datasets. This functionality was vital for identifying commonalities and differences in apprentices' strategies and experiences, following the methodological recommendations of Baxter and Jack (2008). By combining these insights, the process established broader patterns while maintaining the depth and nuance of individual cases. In summary, generating initial codes, facilitated by NVivo, was a pivotal step in thematic analysis. The integration of NVivo's structured tools ensured a robust, data-driven approach to exploring apprentices' experiences and strategies during debugging.

Step 3: Theme Generation

The third stage of thematic analysis, as outlined by Braun and Clarke (2006), involved collating initial codes into meaningful themes, a vital step in organising and interpreting qualitative data. This process required systematically sorting and grouping codes based on their relationships and their relevance to the research questions, as supported by Fereday and Muir-Cochrane (2006). By exploring these connections, broader patterns and themes were identified, revealing both shared and unique aspects of apprentices' experiences and strategies across the data sets. This process aligned with Stake's (2006) emphasis on capturing the depth and complexity of multiple case studies. NVivo played a crucial role in this stage, with its thematic mapping capabilities providing a visual framework for organising and refining themes.

As the analysis progressed, the iterative movement between coding and theme generation ensured a deeper engagement with the data, creating opportunities to refine emerging themes further. The critical examination of themes at this stage was essential, ensuring they aligned with the research objectives and were grounded in both the data and the study's theoretical framework. The researcher's interpretive role was pivotal, requiring careful judgement to determine how themes integrated into the broader narrative of the study. NVivo's comparative tools further supported this process by allowing the visualisation of relationships between themes and highlighting overlaps and distinctions across cases. These capabilities enriched the analytical process by enabling a more systematic exploration of patterns across the apprentices' experiences.

By leveraging NVivo's tools and maintaining a reflexive approach, this stage established a robust analytical structure that contributed to identifying and understanding significant patterns within the data. The generation of themes built on the insights from the coding stage, creating a cohesive framework that provided the foundation for deeper exploration in subsequent stages of analysis. This process addressed the study's research questions and offered a comprehensive perspective on the apprentices' strategies and experiences within the context of the multiple case study.

Step 4: Theme Review

The fourth stage of thematic analysis, as outlined by Braun and Clarke (2006), focused on rigorously reviewing the identified themes to ensure they accurately represented the data. This process involved refining, merging, or separating themes as needed, a critical

step emphasised by Braun and Clarke (2006) and Bazeley (2013) to guarantee that the themes were truly reflective of the data. The review ensured that the themes were coherent and meaningful across various cases and stages of data collection, supporting the reliability and validity of the analysis, as highlighted by Yin (2018). NVivo was instrumental in facilitating this process, with its visual tools allowing for systematic comparisons and refinements of themes.

The iterative review process required a critical examination of how each theme related to the research questions and objectives, contributing to the broader narrative of the study. This involved identifying patterns common across multiple cases while also recognising themes unique to particular cases, which is essential in a multiple case study approach. NVivo's capabilities for visualising relationships and overlaps between themes were crucial. These tools enhanced the reflexive nature of the review, ensuring themes were not only descriptive but also interpretative, aligning with the principles of qualitative research. This stage was pivotal in capturing the complexities and nuances across cases, providing a robust foundation for the final interpretation of findings and ensuring the reliability and validity of the thematic analysis.

Step 5: Theme Definition

The fifth stage of thematic analysis, as outlined by Braun and Clarke (2006), focused on defining and naming themes. This involved conducting a detailed analysis to refine the specifics of each theme, interpreting its essence, and exploring its relationship to the overall narrative of the data. The interpretive process, further elaborated by Clarke and Braun (2017), was essential in distilling the core meaning of each theme and ensuring it

aligned with the research questions and objectives. This stage was particularly significant in a multiple case study context, as it required articulating how each theme manifested across different cases and data sources, as supported by Eisenhardt and Graebner (2007). NVivo facilitated this process by providing tools for visualising and organising themes, helping to ensure they reflected underlying patterns and insights accurately.

Critically examining and contextualising themes within the broader study scope was integral to this phase. The themes needed to strike a balance between being descriptive enough to represent the data and interpretative enough to offer deeper insights into the research problem. This process ensured that the themes were both grounded in empirical data and connected to the theoretical framework of the study. NVivo's visualisation capabilities further supported this balance by enabling comparisons and in-depth exploration of thematic relationships. The researcher's reflexive approach was pivotal in shaping these final themes, ensuring they resonated with the study's aims while providing a meaningful and insightful representation of the data. This stage was instrumental in preparing the groundwork for the final synthesis and interpretation of findings, contributing to a comprehensive understanding of the research problem across multiple cases.

Step 6: Reporting

The sixth stage of thematic analysis, as outlined by Braun and Clarke (2006), centred on producing a coherent and compelling report of the study's findings. This process involved presenting the identified themes using vivid examples from the data, a strategy emphasised by Braun and Clarke (2006) and Creswell (2013) to effectively illustrate the

insights derived from the analysis. The findings were carefully linked to the research questions and the broader literature, ensuring the analysis was anchored in both empirical data and the wider academic context. Synthesising findings from the study's multiple data sources was a complex but essential task, as highlighted by King (2004) and Yin (2018). This required integrating insights from each data source to reflect the breadth and depth of the data, particularly given the multiple case study design. The report offered a holistic view of the research problem, comprehensively understanding apprentices' experiences and strategies.

Producing the report adhered to established principles of qualitative academic writing, creating an integrated narrative that presented a clear and insightful understanding of the research problem. Scholars such as Marshall and Rossman (2016) and Merriam and Tisdell (2015) advocate for this narrative approach, which ensures an engaging and coherent representation of qualitative findings. The report included a detailed account of the themes, effectively conveying the complexity and depth of the data, as suggested by Silverman (2016) and Ritchie et al. (2013). By integrating the findings with existing literature, the study situated itself within the broader academic discourse. Furthermore, a comparative approach, as recommended by Baxter and Jack (2008) and Eisenhardt (1989), was employed to synthesise findings across multiple case studies. This approach highlighted both unique and shared experiences, offering a reflective synthesis that provided a comprehensive understanding of the research problem. As the culmination of the thematic analysis, the report encapsulated the study's insights and presented a cohesive narrative of apprentices' strategies and the role of technology in debugging.

4.5 Empirical Research Process

There have been decades of studies investigating how students learn to debug (Katz & Anderson, 1987; Murphy et al., 2008; Perkins & Martin, 1986), including multiple think-aloud studies examining student debugging (Fitzgerald et al., 2008; Liu et al., 2017; Perkins & Martin, 1986; Yen et al., 2012). However, despite extensive work on understanding student debugging, there are few detailed, qualitative studies of the debugging practices of novice programmers (Whalley et al., 2023). In addition to the well-acknowledged fact that novice programmers encounter substantial debugging challenges, as Bottcher et al. (2016) noted, there is a notable gap in the current literature concerning the debugging strategies employed by novice programmers in distributed environments.

To address this gap, the study adopted a holistic approach to data collection, employing a range of research instruments. These included non-participatory observations, observation notes, think-aloud protocols, screen capturing, audio recording, code analysis, in-depth interviews, and focus groups. This comprehensive approach was designed to capture data from varied perspectives and settings, focusing on how paired apprentices interacted, their use of technology, their verbalisation of thought processes in line with Ericsson and Simon (1984) think-aloud protocol, in-depth interviews and a focus group. The utilisation of these diverse data collection methods is a cornerstone of this multiple case study research, enhancing the credibility of the data (Bogdan & Biklen, 2007; Patton, 1990).

This study utilised a comprehensive, multi-step method to examine the debugging process within an apprentice pairing context, as visually represented in Figure 7 and the data collection timeline and analysis timeline in Figure 8. It began with directly observing and recording debugging sessions, capturing key interactions and challenges. A detailed analysis of these sessions followed this to identify patterns and difficulties in the process. In-depth interviews with apprentice pairs, 'dyads', were then conducted for qualitative insights, with subsequent analysis of these interviews to glean further details. The study's scope expanded to include focus groups with the workplace mentors and trainers, allowing for diverse apprentice perspectives. The final stage involved synthesising all data to fully understand the apprentices' debugging experience, offering an in-depth investigation of the complexities in apprentice learning environments.

Step 1: Observation & recording of debugging sessions



Participants: 30 Paired Apprentices (15 Dyads)

Deployed the Distributed Pair Debugging Conceptual Framework theoretical lenses to guide the investigation of dyad interactions and with technological agents, thought processes and strategies deployed them to localise and fix Python bugs. Also, check the cognition distribution sharing pattern.

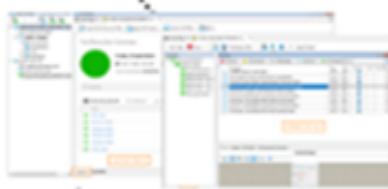
No participant was involved.

Analysis, taking notes of the emerging patterns taken into account the Distributed Pair Debugging Conceptual Framework, i.e., cognitive load sharing mechanisms in place, the gulf in the expected and actual actions and the navigation strategies. The keynotes and emerging patterns to guide the questions to be followed up during interview sessions.

Step 3: Paired apprentice 'Dyads' interview



Step 2: Analysis of recorded debugging sessions



30 Paired Apprentices (15 Dyads)

- Interviewed each dyad to validate the observation findings and to clarify grey areas from the video recording.
- The interview guided by the Distributed Pair Debugging Conceptual Framework, as mentioned earlier.

No participant was involved.

- Analysis, categorisation of the emerging patterns
- Use the theoretical lens, i.e., Distributed Pair Debugging Conceptual Framework as a framework for the analysis of the data.

Step 5: Focus group



Step 4: Analysis of recorded interviews

Participants: 12 workplace mentors & trainers

- Carried out two focus groups sessions, with 6 participants per group/session.
- Use the forum to elucidate views and validate findings from apprentice interviews and observations.
- Summarise the findings and solicit

No participant involved

- Analysis of emerging themes from the focus groups and extracting themes
- Consolidate the themes from the entire research instruments to answer the research questions.



Step 6: Analysis of recorded focus group sessions

Figure 7: Empirical research process.

Data Collection & Analysis timeline					
Observation of Debugging Session	Data Analysis	Interviews	Data Analysis	Focus Group	Data Analysis
	→				
• April – July 2022 • October 2022 • March – April 2023	→	• May – August 2022 • April – June 2023		→	• June 2023
				→	

Figure 8: Timeline of data collection and data analysis.

4.5.1 Step 1: Debugging sessions.

During this stage of the research, an exploration of debugging methods employed by novice programmers was conducted. This involved leveraging various primary data sources, including observation notes, code analysis, insights from think-aloud protocols, and audio and video recordings. Central to this was the implementation of the think-aloud protocol, which provided a window into the cognitive processes of paired apprentices collaboratively debugging Python code. This technique proved critical in illuminating aspects such as cognitive load sharing and the myriad challenges faced during the debugging process, highlighting the pivotal role of technological tools within this framework.

Adding further dimensionality to the data collection for this phase was the non-participatory observational notes recorded as the apprentices navigated the debugging tasks. The richness of these data was enhanced by audio and video recordings, offering a robust mechanism for corroborating and reinforcing key aspects of the investigation (Yen et al., 2012). This multimodal approach was integral to dissecting the primary issues at the heart of the study. All this helped in addressing RQ1, RQ2, RQ3, RQ4 and RQ5. This

multi-faceted research approach and data sources are visually represented in Figure 8, forming the bedrock of the study.



Figure 9: Debugging session research approach.

Furthermore, all 15 pairs (dyads) were given the same Python code, which contained 20 intentional bugs comprising of 11 syntax errors, 6 logical errors, and 3 runtime errors (for the Python Code, refer to Figure 10; for additional details on the Python code, see Tables 11 and 12). This code was used to demonstrate a variety of errors, ranging from those typically made by novices due to unfamiliarity with Python's syntax to errors stemming from a lack of logical understanding or inadequate planning before coding. While syntax and runtime errors are often more readily identifiable through error messages, logical errors can be particularly challenging for novice programmers to detect. Understanding the nature of these varying levels of difficulty is essential for beginners in programming and educators and mentors. This understanding aids in designing educational materials and establishing realistic learning goals for apprentices.

```

***  

The Python program intends to calculate an employee's payroll using hours worked and hourly rate. It provides details like gross pay, net pay, and potential bonuses based on the employee's role. However, numerous errors hinder its smooth execution.  

***  

def calculate_payroll(hours_worked, hourly_rate)  

    # SYNTAX ERROR (SE01): Missing colon  

    gross_pay = hours_worked x hourly_rate  

    # SYNTAX ERROR (SE02): Used 'x' instead of '*'  

    # LOGICAL ERROR (LE01): Incorrect tax value  

    tax_rate = 15  

    # SYNTAX ERROR (SE03): Missing colon after 'if'  

    if gross_pay > 6000  

        tax = gross_pay * (tax_rate/100)  

    else  

        # LOGICAL ERROR (LE02): Wrong tax rate  

        tax = gross_pay * 0.05  

    net_pay = gross_pay - tax  

    # LOGICAL ERROR (LE03): Shouldn't subtract tax if gross_pay is below a certain threshold  

    return 'Total Pay: ', str(gross_pay) + ', Net Salary: ' + str(net_pay)  

    # SYNTAX ERROR (SE04): Mismatched string concatenation  

# SYNTAX ERROR (SE05): 'def' typo  

def main():  

    hours = input("Input hours: ")  

    rate = input("Input rate: $")  

    # RUNTIME ERROR (RE01): Input is string and not converted to number  

    payroll_info = calculate_payroll(hours, rate)  

    # SYNTAX ERROR (SE06): print without parentheses  

    print payroll_info  

    # RUNTIME ERROR (RE02): Undefined variable 'rates'  

    print(rates[0])  

# LOGICAL ERROR (LE04): Improper use of '__name__'  

if name == '__main__':  

    # SYNTAX ERROR (SE07): Single '=' used instead of '=='  

    main()  

# SYNTAX ERROR (SE08): Incorrectly closed string  

role = input("Enter employee's role:")  

# SYNTAX ERROR (SE09): Incorrect indentation  

if role == "Manager":  

    # LOGICAL ERROR (LE05): Bonus amount doesn't make sense without context  

    bonus = 2000  

    print("Bonus: ", bonus)  

# SYNTAX ERROR (SE10): Else without a prior if (due to the indentation error above)  

else:  

    print("No bonus")  

# LOGICAL ERROR (LE06): Redundant and incorrect code  

bonus = 100  

print("All employees get a bonus of: ", bonus)  

# SYNTAX ERROR (SE11): Incomplete 'for' loop  

for i in range (5)  

    print()  

    # RUNTIME ERROR (RE03): Infinite loop due to missing colon and indentation

```

Figure 10: Python code seeded with syntax, logical and runtime bugs

Table 11: List of bugs, bug type and difficulty level.

Bug ID	Bug Description	Bug Type	Difficulty Level	Explanation
SE01	Absence of colon in function definition	Syntax	Easy	Python consistently uses colons before blocks of code (e.g., loops, conditions, functions). A clear error message like "expected ':'" typically points directly to this mistake.
SE02	Wrong operator for multiplication	Syntax	Easy	This mistake can be spotted through error messages or by familiarity with basic arithmetic operators in most programming languages.
SE03	Missing colons	Syntax	Easy	Similar to SE01, once a programmer knows the pattern of using colons, these errors become less common and easier to identify.
SE04	Mismatch in string concatenation	Syntax	Moderate	String manipulation can be tricky initially, especially for beginners who are juggling with the different ways to concatenate or format strings.
SE05	'def' misspelled	Syntax	Easy	Clear error messages guide toward these typographical errors. Recognising that 'def' is a keyword in Python makes it easier to spot this error.
SE06	Missing colons	Syntax	Easy	Similar to SE01, once a programmer knows the pattern of using colons, these errors become less common and easier to identify.
SE07	Wrong comparison operator	Syntax	Easy	This is one of the common mistakes beginners make. Error messages typically indicate an assignment inside a condition, which can hint at the problem.
SE08	Incorrectly closed string	Syntax	Easy	Syntax errors will highlight where the string or line of code breaks, making it easier to identify the incorrectly closed string.
SE09	Indentation errors	Syntax	Easy	Indentation in Python denotes code blocks. Novices sometimes miss the importance of consistent indentation, leading to errors. However, Python error messages about indentation are generally clear.
SE10	Indentation errors	Syntax	Easy	Indentation in Python denotes code blocks. Novices sometimes miss the importance of consistent indentation, leading to errors. However, Python error messages about indentation are generally clear.
SE11	Missing colons	Syntax	Easy	Similar to SE01, once a programmer knows the pattern of using colons, these errors become less common and easier to identify.
LE01	Incorrect value for tax rate	Logical	Moderate	Logical errors don't throw actual error messages. Spotting an incorrect value might require either knowledge of the domain (i.e., actual tax rate) or manual verification of results.
LE02	Wrong tax calculation logic	Logical	Moderate	Similar to LE01, verifying calculation results against expected outcomes is necessary. Debugging skills play a crucial role here.
LE03	Tax deduction logic flaw	Logical	Moderate to Difficulty	Understanding the overall logic of how and when tax should be deducted can be intricate. Debugging and step-by-step verification can help.
LE04	Misuse of the special variable ' <code>__name__</code> '	Logical	Difficulty	Beginners might not fully understand the purpose of the ' <code>__name__</code> ' variable in Python, making it harder to spot the misuse.
LE05	Bonus calculation lacks context	Logical	Moderate	This error highlights the importance of understanding the bigger picture or context when writing a program.
LE06	Redundant bonus logic	Logical	Moderate	Redundancy can lead to confusion and unexpected behaviour. Recognizing redundancy requires a good grasp of the entire codebase.
RE01	Not converting string input to number before mathematical operations.	Runtime	Moderate	Python will raise a 'TypeError' when trying to perform arithmetic on incompatible types. The error message hints at the problem, but beginners might be confused if they assume all inputs are numbers.
RE02	Attempt to use an undefined variable.	Runtime	Easy to Moderate	The error message " <code>name 'variable_name' is not defined</code> " is direct. However, beginners might struggle to understand why the variable isn't defined, especially if they believe they have defined it.
RE03	Infinite loop due to missing colon in 'for' loop.	Runtime	Moderate	Spotting infinite loops requires observation of the program's behaviour. The program not terminating or becoming unresponsive is a clue. However, tying this behaviour to a missing colon might take a bit of thought.

Table 12: Characteristics of the bugs' difficulty levels.

Difficulty Level	Characteristics of difficulty levels
Easy	<ul style="list-style-type: none"> Instant Feedback: These errors often produce immediate feedback in the form of compiler or interpreter error messages that directly hint at the mistake. Commonality: These mistakes are frequently made by beginners and are thus well-documented in learning resources and forums. Simple Resolution: Once identified, the error can be fixed with minimal changes to the code. Clear Symptoms: The symptoms of the error are evident and do not require deep investigation. For instance, a syntax error would halt the program before it even runs. Low Impact: Errors of this category don't generally affect other parts of the program when corrected.
Moderate	<ul style="list-style-type: none"> Less Direct Feedback: While some of these errors might yield error messages, the messages might not directly point to the root cause. Requires Debugging Skills: To resolve these, a beginner might have to use basic debugging techniques like print statements or step-through debugging. Variable Symptoms: The symptoms can vary in clarity. For instance, a logical error might not halt the program but produce incorrect results. Interconnected Impact: Fixing a moderate error might necessitate changes in other parts of the code, especially if the mistake involves a fundamental logic flaw. Experience Helps: Those who have encountered similar issues before can identify and fix these errors more quickly.
Difficult	<ul style="list-style-type: none"> Vague or No Feedback: These errors do not always yield clear error messages. Logical errors, in particular, might not produce any error messages at all. Deep Investigation Required: Resolving these might require a deep understanding of the code, the problem domain, or even the programming language's intricacies. Subtle Symptoms: The symptoms might not be evident at first glance. For instance, a program might seem to work correctly for some inputs but fail for others due to a deep-seated logical error. High Interdependence: Errors of this type are often intertwined with the program's core logic and fixing them might require significant restructuring. Experience and Domain Knowledge: Beyond just coding experience, domain knowledge (e.g., understanding specific algorithms, mathematical concepts, or industry-specific knowledge) can be crucial in identifying and resolving these errors.

4.5.2 Step 2: Analysis of recorded debugging session

In the research approach depicted in Figure 9, three distinct data analysis tasks transpired following the observation of the debugging session. These included the analysis of observation notes and the transcript of the audio and video recordings. For both these research instruments, the data analysis adhered to the thematic analysis approach adapted from Braun and Clarke (2006), as detailed in Section 4.4.4 of the thesis.

Additionally, another pivotal element of the data analysis was the examination of artefacts from the final Python code. These artefacts, representing the various stages of

code modification undertaken by the apprentices during their debugging sessions, were crucial in identifying the nature of errors. This included categorising the errors into those that were rectified, those identified but left unresolved, and those that remained unnoticed by the apprentices. The scrutiny of these code artefacts proved vital in pinpointing specific challenges encountered by novice programmers.

Furthermore, insights gained from the analysis of code artefacts were then juxtaposed with established literature on programming errors. This included studies by Ettles et al. (2018), Grandell et al. (2005), Jeffries et al. (2022), Júnior et al. (2019), Kohn (2019), Kohn and Manaris (2020), Pritchard (2015), Smith and Rixner (2019), and Veerasamy et al. (2016). This comparative analysis played a crucial role in enabling a detailed comprehension of well-known and emerging difficulties faced by novice programmers in the field. Such an understanding was crucial in enabling the research to suggest the types of bugs generated by the paired, geographically distributed SDT apprentices who collaboratively worked on resolving bugs in Python code.

4.5.3 Step 3: Interview sessions

In this phase of the study, semi-structured interviews were conducted using a dyadic interview approach, as outlined by Kendall et al. (2009). The significance of these interviews lay in their ability to provide rich and detailed qualitative data, which was essential for understanding participants' experiences, their descriptions of these experiences, and the meanings they derived from them, a concept suggested by Rubin and Rubin (2011). The interview protocol (see Appendix H), inspired by the framework of Castillo-Montoya (2016), underwent four distinct phases, including, alignment with

research questions, constructing inquiry-based conversations, receiving feedback on protocols, and piloting the protocol. These phases were designed to develop a research instrument that fit the study's participants and aligned with its research goals, as Jones et al. (2013) emphasised. The Interview Protocol Rigor (IPR) framework was employed to provide a shared language for indicating the steps taken in developing interview protocols and ensuring their congruency with the study (Jones et al., 2013).

The in-depth interviews aimed to understand the dyads' experiences, particularly their strategies for debugging code errors and the role of technology in these activities. A matrix, shown in Table 13, was used to ensure the alignment of interview questions with research questions. This matrix was instrumental in identifying potential gaps and ensuring a balanced focus on each research question.

The interviews focused on four main areas, including (1) investigating how paired apprentices located bugs in the Python code and exploring cognitive load sharing during debugging. This was guided by theories such as information foraging and distributed cognition; (2) exploring apprentices' levels of knowledge, misconceptions, and the impact of technology, particularly IDEs, on the debugging process; (3) reaffirming the dyads' understanding of error messages generated by the IDE and how this informed their bug location strategies; and (4) clarifying issues from observation field notes, video recordings, and think-aloud reports. The dyadic interview approach was chosen for its ability to elicit diverse views (Martens, 2005) and clarify meanings (Britten, 1995). Furthermore, the interview sought to elaborate on the dyads' verbalisations and specific events observed in their problem-solving process. Example questions included inquiries

about specific moments in the video recordings and how the dyads collaborated to find potential solutions.

At the conclusion of the interviews, participants were offered the opportunity to review the provisional findings. This step was taken to gain clarity, improve accuracy, and strengthen the validity of the study, a practice recommended by Lincoln and Guba (1985).

The interview protocol can be found in Appendix H.

Table 13: Interview protocol matrix adapted from Castillo-Montoya (2016).

Interview Questions (IQ)	Research Questions (RQ)				
	RQ ₁	RQ ₂	RQ ₃	RQ ₄	RQ ₅
IQ1	X				
IQ2	X				
IQ2		X	X	X	
IQ4		X	X	X	
IQ5				X	X
IQ6					
IQ7	X	X			
IQ8			X		
IQ9				X	X
IQ10					
IQ11	X	X	X		
IQ12	X		X		X
IQ13		X		X	X
IQ14			X		
IQ15		X		X	

4.5.4 Step 4: Analysis of the dyadic interview session

The data analysis of the dyadic interview sessions in this study was structured to scrutinise the transcripts of the interviews, with a specific focus on the concepts of information foraging theory and distributed cognition. These theoretical frameworks provided a comprehensive lens through which the interactions and cognitive processes of the apprentice pairs could be understood and analysed. Information foraging theory, as articulated by Pirolli and Card (1999), offered a valuable perspective on how individuals

seek and gather information, which, in this context, is related to the apprentices' strategies in locating and addressing bugs in Python code.

Concurrently, the principle of distributed cognition, as explored by Hollan et al. (2000), provided insight into how cognitive processes are shared and distributed across individuals working collaboratively, particularly pertinent in examining the cognitive load sharing between the apprentice pairs. This analytical approach was further underpinned by the work of Hutchins (1995), whose work on distributed cognition in real-world contexts enriched the analysis of the collaborative problem-solving observed in the interviews.

In analysing the interview transcripts, the study followed the thematic analysis approach, which was adapted from the framework proposed by Braun and Clarke (2006). This methodology, described in Section 4.4.4, provided a structured and comprehensive means of evaluating and interpreting the data gathered from the interviews.

4.5.5 Step 5: Focus group session

The focus group conducted in this study played a pivotal role in investigating the perspectives of workplace mentors and trainers from the training organisation. The key areas of inquiry included the types of errors made, the bug location strategies commonly used by novice programmers, and the general challenges faced by apprentices or novice programmers. Additionally, this focus group provided an essential platform for eliciting their opinions on the findings gathered from apprentice observations, interviews, and analysis of software artefacts, thereby aiming to gather and reaffirm their interpretations

and understanding of the phenomenon under study, in line with the approach recommended by Khan and Manderson (1992).

To facilitate in-depth discussion and ensure clarity on the issues related to the phenomenon, the focus group, as detailed in Section 4.4.3, comprised 12 participants. However, this group was strategically divided into two cohorts, each consisting of six members. This division was intended to enhance the depth and quality of the discussions, as smaller groups are often more manageable and can provide more detailed feedback, a methodological approach supported by Liamputpong (2011).

In selecting participants, significant emphasis was placed on recruiting workplace mentors and trainers with relevant profiles, as described in Section 4.4.3. This selection process ensured that each cohort comprised individuals with appropriate expertise and experience. Furthermore, the decision to limit the size of each cohort to six participants was informed by the guidelines suggested by Greenbaum (1998), who noted the importance of group size in generating valuable and in-depth data in focus group research. This approach was deemed essential to ensure that the data collected were relevant and rich in insights pertinent to the research questions.

4.5.6 Step 6: Analysis of recorded focus group session

The focus group in this study was instrumental in validating the initial findings obtained from the apprentices and enriching these findings with additional perspectives from experts who work closely with the apprentices. To facilitate a thorough analysis, the transcripts of the two focus group discussions were imported into NVivo, as outlined in

Section 4.4.4. This software provided an organised framework for managing and analysing the data.

The primary analysis within NVivo focused on the content of the verbalisations during the focus group sessions. This approach aligns with the methods advocated by Jordan and Henderson (1995), who suggest the significance of focusing on the participants' verbal expressions in group discussions. Such an emphasis guarantees that the minor distinctions and depth of the participants' perspectives and experiences are captured and analysed comprehensively.

By employing NVivo for this analysis, the study systematically categorised and explored the rich qualitative data provided during the focus group sessions. This facilitated a detailed examination of the themes and patterns within the verbalisations, thus enabling a robust understanding of the experts' perspectives on the apprentices' experiences and challenges.

4.5.7 Limitation of the Chosen Methodology

Whilst this study offers valuable insights into the debugging strategies of novice apprentices, it is not without its limitations, which stem from both the research design and the practical constraints of the methodology employed.

One significant limitation is that the dyads of apprentices did not know each other prior to the debugging sessions, which likely impacted their collaborative dynamics. Without sufficient time to establish rapport, the participants may have been less comfortable

sharing ideas freely, which could have influenced the fluidity of their interactions and the effectiveness of their collaborative debugging strategies (Murphy et al., 2010). Additionally, the study was conducted over a short time frame, meaning the apprentices had limited time to become accustomed to each other's working styles. This restricted the ability to observe how their collaborative skills might evolve with extended practice (Jayathirtha et al., 2020).

Another limitation is the focus on only novice apprentices, which means the findings may not extend to more experienced programmers, whose strategies and collaboration in debugging may differ significantly. Similarly, the reliance on self-reported data during interviews presents a challenge, as participants may have unintentionally underreported or misrepresented their experiences, introducing potential biases. This could reduce the possibility of *participant conformity*, where interviewees might have aligned their responses to perceived expectations (Finlay, 2002).

Finally, the exclusive focus on Python and Microsoft Visual Studio as the development environment may limit the applicability of the findings to other programming languages or IDEs. Each language and tool presents unique challenges in debugging, and as such, the conclusions drawn from this research may not fully generalise to different technical settings (Alqadi & Maletic, 2017).

4.6 Reliability and Validity

This research was guided by the array of strategies outlined by Maxwell (2012), which are specifically designed to address and mitigate threats to validity within qualitative

research paradigms. Recognising that the enumeration of these strategies may differ across various editions or interpretations of Maxwell's work, this particular study incorporated seven of the eight widely recognised techniques to fortify its validity. The methodologies utilised were rich data (Becker, 1971), respondent validation or member checks (Bryman, 2003; Hammersley & Atkinson, 1995; Lincoln & Guba, 1985), intervention, searching for discrepant evidence and negative cases, triangulation, numbers, and comparison (Miles & Huberman, 1994). The study utilised specific strategies outlined by Maxwell (2012) that align with and enhance validity tests Yin (2009) put forth, focusing mainly on construct and external validity areas. This helps tackle two broad types of threats to validity often raised in qualitative studies, which are researcher bias and reactivity.

Construct Validity

As Yin (2009) articulated, this validity facet scrutinises the accuracy of the research measures in capturing the intended concepts. It necessitates that the operational mechanisms in the research reliably reflect the constructs they are meant to measure. To bolster construct validity, strategies such as employing data triangulation, maintaining a chain of evidence, and engaging in member checking are pivotal (Yin, 2009). In alignment with these techniques, the current study integrates Yin's (2009) framework with Maxwell's (2009) methodological insights, implementing triangulation and comprehensive data collection to substantiate construct validity within the context of SDT apprentices' debugging practices.

The apprentices' cognitive processes are documented using the think-aloud protocol, providing a dynamic and participatory view of problem-solving, as underpinned by the work of Ericsson and Simon (1984). Observations of dyadic interactions within authentic coding settings offer a narrative-rich perspective on collaborative problem-solving, an approach augmented by reflective interviews that probe deeper into the apprentices' decision-making processes (Kvale, 1996). These reflective interviews transform apprentices into active narrators, thereby providing rich data by adding layers to the observational data and painting a more intricate picture of their cognitive experiences during debugging (Becker, 1971).

Additionally, engaging with mentors and trainers furnishes a deeper insight into the apprentices' problem-solving strategies, corroborating the study's findings with the apprentices' real-world debugging activities, an approach supported by Merriam (2009). These professionals in this situation confirm the findings and provide critical analysis of observed behaviours and outcomes. This triangulated and detailed methodological approach captures cognitive activities, debugging strategies, and team synergy, fortifying the study's construct validity (Maxwell, 2008; Yin, 2009). The research, therefore, intertwines various data threads to construct an authentic narrative of the apprentices' engagement with complex programming challenges, resonating with Stake's (1995) emphasis on creating rich, qualitative narratives.

External Validity

In the context of a qualitative study investigating the debugging practices of dyad apprentices working with Python code across multiple sites, internal validity is critical to

the integrity of the research. Yin (2003) suggests the importance of replication logic in multi-case studies to underpin external validity, drawing parallels to experimental research. By documenting the recurring emergence of the same phenomenon across various settings, this study substantiates the external validity of the findings, as supported by Baxter and Jack (2008), who advocate for the replicability of qualitative studies as a means to broader applicability. While the primary emphasis of such studies often lies in the depth of understanding rather than generalisability, the consistent replication across cases provides a compelling foundation for claims of wider relevance (Stake, 2006).

In sum, employing these two measures is vital to the assurance of rigour and reliability in this study, which utilises case studies along with diverse qualitative research techniques. However, Campbell (1988) emphasises the significance of employing strategies that focus less on confirming findings and more on scrutinising the credibility of one's conclusions and identifying any possible risks to their validity. Similarly, Maxwell (2012) agrees with this approach, advocating for the active search for information that might contest one's conclusions or relate to the likelihood of identified potential risks.

4.7 Ethical Issues and Concerns

The study included participants aged 16 and older who were engaged in the Software Development Technician Apprenticeship standard within diverse workplaces. Ethical considerations took precedence for the duration of the study (see Appendix E). Activities such as pair programming and debugging were assessed as low-risk, typically offering benefits to the participants without foreseeable adverse outcomes.

Before initiating the empirical research, it was essential to obtain formal ethical clearance from the Ethics Committee at Lancaster University and secure consent from both the apprentices' training providers and their workplaces. This ethical approach was reinforced through the documentation of consent, both in writing and verbally, with all involved parties, ensuring strict adherence to ethical research standards. Additionally, participants were fully informed about the study's objectives and were clearly advised, as stated in Appendices A to D, of their right to withdraw at any point during the study.

With the commencement of the data collection phase, maintaining ethical standards became paramount. The process included acquiring informed consent from apprentices and ensuring the confidentiality and privacy of the data collected. Transparency regarding the study's goals and methods was consistently upheld, alongside a dedicated effort to protect participant autonomy and rights, thereby avoiding coercion and ensuring all participants' welfare. Data were handled with the utmost integrity, with secure storage and ethical use in accordance with both the trust of the participants and the stipulations of the Ethics Committee. All written and verbal communications incorporated core ethical principles, including beneficence, non-maleficence, informed consent, confidentiality, and anonymity, solidifying participants' understanding of their autonomy within the research.

When drafting the research report, particular attention was given to confidentiality measures. To preserve the anonymity of the research findings, personal and corporate identifiers were meticulously omitted. Recognising the unique challenges of a multiple-case study, which inherently carries a higher risk of disclosing participant identities,

especially through detailed descriptions, the research implemented judicious modifications to the contextual information presented, thus ensuring the protection of participant identities.

Finally, the multifarious data collated during this study, spanning videos, codes, and compiler reports, were stored electronically with the utmost security. Ensuring compliance with the UK GDPR, all data were meticulously housed on a Microsoft OneDrive account associated with the researcher's Microsoft Office, positioning the researcher as the chief custodian of this vital information.

4.8 Summary

This chapter describes the research methodology adopted for this study, providing an overview of the approaches and procedures employed to address the research questions. It commences by justifying the selection of a qualitative research design deemed most suitable for an in-depth investigation of the phenomena of interest. The rationale behind this choice was grounded in the exploratory nature of the study, which sought to gain rich, contextualised understandings rather than broad generalisations.

Following this, the chapter outlined the specific methods of data collection utilised. A multi-case study approach was employed to allow for a detailed examination of each instance within its real-life context. The selection of cases was based on purposive sampling, informed by the criteria of information richness and relevance to the research aims, as suggested by Patton (2015). The detailed process of obtaining ethical clearance

from institutional review boards and informed consent from participants was then described, pointing out to the ethical rigour underpinning all stages of the research.

The data collection methods were varied, including in-depth interviews, participant observations, and document analysis. These methods provided a triangulated view that enhanced the reliability and validity of the findings. The procedures for data analysis were explained, noting the iterative process of coding and theme development in line with the established qualitative analysis frameworks.

Subsequently, the chapter discussed the measures taken to ensure the study's trustworthiness and credibility. Strategies such as member checking, audit trails, and reflexive journaling were employed to bolster the study's integrity. Lastly, the methodology's limitations were acknowledged, with a candid discussion about the potential implications for the study's findings and their applicability.

In summary, this chapter has articulated the systematic approach taken to ensure that the study's results are as robust as possible, ethically sound, and contribute meaningfully to the body of knowledge in the field.

Chapter 5: Findings

5.0 Introduction

This chapter presents the findings of a study involving paired apprentices situated in different locations who collaborated to debug Python code. The study primarily evaluates their strategies for resolving coding errors, how they have deployed technological tools, their methods for sharing the cognitive load, and the challenges encountered while solving problems as a team. To address specific research questions, the study gathered data from various sources, including observational notes and videos from debugging sessions, interviews with pairs of apprentices, and discussions with mentors and trainers in work-based settings. The study's findings reveal key aspects of debugging practices among participants, identifying some themes, namely, the use of technology in debugging, specific strategies and tactics employed, the variety of errors encountered, how cognitive load is managed, and the challenges faced during the debugging process.

5.1 Dyads Debugging Session Findings

The study encompassed a total of 15 debugging sessions, which took place over seven months and collectively lasted for 30 hours. These sessions occurred between April and July 2022, October 2022, and March to April 2023. The research involved 30 apprentices who were paired into 15 dyads. As discussed in Section 4.4.4, the data analysis of the transcribed video recordings and the observation notes adhered to the thematic analysis approach adapted from Braun and Clarke (2006). Utilising Braun and Clark's thematic analysis, the study identified themes illuminating different facets of the debugging process, including technology utilisation, debugging strategies and tactics, error

spectrum, cognitive load management, and encountered challenges as seen in Table 14, which summarises the main themes identified during the 15 dyads' debugging sessions. . These themes are critical to understanding how the participants tackled the debugging process, the tools they employed, and the challenges they encountered.

Table 14: Overview of key themes in dyads debugging sessions

Theme	Description
Theme 1: Technology Utilisation	The critical role of various technological tools in the debugging process, especially Microsoft Teams and IDEs.
Theme 2: Debugging Strategies and Tactics	Diverse strategies such as tinkering, trial and error, and print statement debugging employed by the dyads.
Theme 3: Error Spectrum	Types of errors encountered by dyads: syntax, logical, and runtime errors.
Theme 4: Cognitive Load Management	How dyads shared the mental effort and utilised collaborative strategies to manage the debugging process.
Theme 5: Challenges Faced	Key difficulties encountered, including technical challenges and the complexities of collaborative debugging.

5.1.1 Theme 1: Technology Utilisation

In the debugging sessions, 'Technology Utilisation' emerged as one of the prominent themes, reinforcing the critical role of various technological tools in the debugging process, especially Microsoft Teams, as seen in Table 15.

Table 15: Technology Utilisation Subthemes in Dyadic Debugging Sessions

Subthemes	Description
Collaborative Tools: Microsoft Teams	Microsoft Teams enabled real-time communication and collaboration, enhancing problem-solving, visual interaction, and task coordination in remote debugging sessions.
Real-Time Collaboration with Live Share	Live Share facilitated real-time code editing, error navigation, and role transitions, boosting productivity and problem-solving in Visual Studio.
Integrated Development Environments (IDEs)	IDEs like Visual Studio were crucial in debugging, with features like syntax highlighting, error detection, breakpoints, and code comparison aiding error identification, execution flow analysis, and code clarity.
Version Control and Documentation	Participants used OneDrive and documentation to preserve scripts, record processes, and maintain organised collaboration and problem-solving.

As seen in Table 15, a significant subtheme is the role of Microsoft Teams in supporting collaborative debugging. Microsoft Teams served as a vital communication hub, enabling real-time interaction and idea sharing among participants. Its features, like chat, video calls, and screen sharing, were instrumental in problem-solving. This is echoed in SDT27's statement, "I am glad we could all connect seamlessly on Microsoft Teams for this session. Seeing each other's reactions and screens while we discuss the errors has made our debugging much more effective". SDT29's comment, "great progress today! I will upload our revised script to the Teams channel now for us to review the changes together. We can use the screen sharing feature to walk through the code". These quotes showcase how Microsoft Teams was crucial for messaging, file sharing, and enhancing the debugging experience through visual interaction and effective communication.

Similarly, the debugging sessions heavily relied on Visual Studio and Live Share and beginning typically in Visual Studio, as SDT1 exemplified, "Okay, SDT2, I have got the script open here in Visual Studio. Let's run it and see what initial errors we're dealing with". It is obvious here that the role of Live Share was visible for collaborative efforts. SDT22 also emphasised, "just launched Live Share for our session. This tool is going to be crucial for us to jointly edit the code, making our debugging way more efficient". This also confirmed the role played by technology, which allowed simultaneous code work, with SDT15 remarking, "while you navigate to the error section using Live Share, I will start tweaking the function above".

Furthermore, Live Share also, from the available data, smoothed role transitions, a point highlighted by SDT10, "Okay, I'm handing over the reins to you now in Live Share. You will

see the changes I have made instantly on your screen". These tools were instrumental in the sessions, enhancing efficiency and fostering a collaborative debugging environment. They suggest the importance of such technologies in modern coding practices, particularly in team-based projects where real-time collaboration and quick role swaps are essential.

Likewise, the application of IDEs was fundamental, as captured in the observation notes and video transcripts. These IDEs, which were central to the debugging process, were equipped with advanced features like syntax highlighting, error highlighting, auto-indentation, and breakpoints. The participants harnessed these tools to quickly spot and fix syntax errors, grasp the execution flow, and conduct detailed variable inspections, highlighting the invaluable role of IDEs in streamlining code analysis and error resolution. Furthermore, the use of syntax highlighting in accelerating error detection is marked by SDT1's comment, "right, making that change now. I'm also keeping an eye on the IDE's syntax highlighting feature. It's really helping to spot these kinds of errors much quicker". In like manner, SDT2's mention of utilising the IDE's auto-indent feature, "while you're fixing that, I'll take advantage of the IDE's auto-indent feature", illustrates how such functionalities aid in maintaining code clarity and structure.

Additionally, the use of IDEs extended to deeper code analysis and debugging. For instance, SDT18 mentioned, "let's make use of the IDE's features. Set a breakpoint and step through the code to catch any subtle errors", and SDT22 remarked, "I've taken the helm now. Let's harness the IDE debugger for a deeper analysis". Similar points are echoed by SDT25, SDT29, and SDT30, who stress the importance of IDEs' advanced

debugging tools and user-friendly interfaces. SDT25 highlighted using the IDE's code comparison tool, "I'm using the IDE's code comparison tool to spot differences". SDT29 stressed the value of advanced debugging tools by suggesting, "I'm setting a breakpoint here in Visual Studio to pinpoint where our code deviates. These advanced debugging tools are a lifesaver for tracking down elusive errors during runtime". Lastly, SDT30 appreciated the user-friendly interface, "I really appreciate how user-friendly Visual Studio's interface is. It makes navigating through our code and identifying these syntax errors so much easier, especially for newcomers like us". These features are crucial for tracking elusive errors and assisting newcomers in navigating complex code. Collectively, these participant statements reinforce how integrating IDE tools in the debugging process significantly boosts productivity, accuracy, and learning, particularly in collaborative settings.

To add to that, the role of version control and systematic documentation was profoundly emphasised. Participants like SDT1 and SDT12 recognised the significance of saving work on OneDrive and documenting the debugging process for future reference, as they stated, "before we wrap up, let's save our final version of the script to OneDrive" and "that's a great idea. Documenting our process will provide valuable insights for future debugging sessions". These practices aid in record-keeping and enhance the collaborative experience, allowing for a structured approach to problem-solving. On the other hand, SDT23's approach to saving notes in the project file, as mentioned, "I'm saving these notes in the project file", demonstrates a methodical approach to debugging, ensuring a thorough understanding for future review. SDT24 and SDT26 further reiterated this sentiment by uploading their final scripts to OneDrive and documenting their process, as

they commented, “I’m uploading the final script to OneDrive now” and “we’ve meticulously documented our endeavours and the remaining challenges”. These quotes collectively illustrate the participants’ commitments to maintaining a detailed record of their debugging sessions, highlighting the importance of version control and documentation in the collaborative development process.

In summary, integrating technology, particularly in remote settings, is essential in enhancing the debugging process and improving team coordination and task management. The findings include excerpts highlighting various technological tools such as IDEs, debuggers, Microsoft Teams, OneDrive, and version control systems. These tools enhance the debugging process, making it more efficient and effective. Participants used these technologies collaboratively to solve complex debugging challenges, demonstrating these tools’ crucial role in modern software development.

5.1.2 Theme 2: Debugging Strategies and Tactics

This theme showcases diverse strategies and tactics to address various coding problems during the debugging sessions, as seen in Table 16.

Table 16: Debugging Strategies & Tactics Themes in Dyadic Debugging Sessions

Subtheme	Description
Tinkering	Participants engaged in incremental modifications and re-execution to gradually refine their understanding and improve the functionality of the code.
Trial & Error	Debugging involved systematic experimentation with inputs, variable types, and small code adjustments to identify and resolve errors through an iterative process.
Print Statement	This simple yet effective debugging technique was widely used to trace variable values and program flow, offering real-time insights into execution logic.

Table 17: Debugging Strategies & Tactics Themes in Dyadic Debugging Sessions (Continuation)

Subtheme	Description
IDE Debuggers	Participants leveraged IDE features like syntax highlighting, breakpoints, and step-through debugging to efficiently locate and resolve coding errors.
Slicing	The method of isolating specific code blocks and testing them independently helped pinpoint errors more efficiently, particularly in complex scripts.
Rubber Duck Debugging	Articulating code logic aloud, whether to a partner or an imaginary listener, helped participants identify overlooked errors and clarify their reasoning.
Code Review	Reviewing code systematically allowed participants to identify syntax, logical, and structural errors, ensuring clarity, maintainability, and functionality.
Pattern Matching	Recognising recurring error patterns enabled participants to apply known solutions quickly, improving efficiency in debugging and problem resolution.
Divide & Conquer	Breaking down large problems into smaller, manageable segments allowed for a more focused and effective debugging process.
Tracing	Following error messages and execution paths back to their source helped participants systematically track and resolve programming errors.

As a debugging strategy, tinkering gained prominence among five dyads during their debugging session, involving the process of making incremental adjustments and testing the script for changes. This approach, encapsulating both anticipation and progression, is vividly illustrated by SDT6's positive stance, "alright, let's execute it again and keep an eye out for what comes next. I have a feeling we are making good progress here". Such a dynamic method accentuates the essence of debugging as an adaptive process where programmers persistently evaluate the effects of their modifications, thereby incrementally enhancing their grasp of the code's behaviour. In a similar vein, SDT5's modification of inputs, "I've made the necessary changes to the inputs. Let's execute the script again and see if that resolves where the input strings were not converted to numbers", along with SDT20's adjustments, "I've made a few tweaks here and there. Let's run it once more to see where we stand", further exemplify this disciplined yet exploratory strategy. This narrative seamlessly integrates the essence of tinkering in

debugging, highlighting its role in fostering a thorough and evolving understanding of code through careful experimentation and adjustment.

In addition, this iterative process of refinement, characterised by minor yet calculated modifications, reflects a broader principle in software development of fine-tuning code to achieve optimal performance. As stated by SDT19's focused intervention, "String fixed. Let's check if that clears the error", further highlights the importance of targeted debugging efforts. By isolating and addressing specific issues before retesting, programmers demonstrate a precise and effective method of troubleshooting that emphasises the critical role of identifying and correcting individual elements for the overall functionality of the code. Through a cycle of continuous tweaking, testing, and reassessment, novices navigate the intricate coding challenges, showcasing a persistent and adaptive mindset that is indispensable in software development.

The trial and error method emerges as a crucial debugging tactic, informed by experimentation in the pursuit of solutions, vividly illustrated through novice experiences. This strategy's essence, characterised by resilience and adaptability, plays a pivotal role in debugging as novice programmers navigate through challenges with persistence and a willingness to experiment. For instance, SDT9's endeavours, "I tried several different inputs to see where the code breaks", capture the exploratory nature of this method, aiming to discern the code's boundaries and behaviour under various scenarios. Similarly, SDT11's experience, "changing variable types was a bit of trial and error, but it worked eventually", sheds light on the iterative debugging journey, emphasising the importance of trial and feedback in overcoming coding obstacles.

The iterative cycle of trial and error is further exemplified by SDT1's approach, "I've made the necessary changes to the inputs. Let's execute the script again and see if that resolves the `TypeError`", highlighting the discipline of implementing, testing, and reassessing modifications to refine the code. SDT7's meticulous attention to detail is evident in "typo fixed. I'm running the script to see if we've cleared the error", emphasising the significance of addressing even minor errors for code functionality.

Besides, SDT8's contribution, "Sure, adding the colon now. Let's see if that solves it", demonstrates the value of minor yet impactful code adjustments in debugging. This highlights the iterative and insightful nature of trial and error, with each minor adjustment or test serving as a step towards solving complex coding puzzles.

Similarly, print statement debugging is presented as a cornerstone of the diagnostic process within the dyads, lauded for its simplicity and capability to deliver real-time insights into program behaviour. This method is notably appreciated for its straightforwardness, offering a direct window into the inner workings of a program, as testified by several participants who highlighted its practicality across various coding situations. SDT2 champions this approach for tackling complex logical segments, advising, "I suggest we use print statements to trace variable values, especially in complex logical segments. It's always helpful to see exactly what's happening in real-time". This sentiment suggests print statements' value in unravelling code complexities by providing immediate, tangible feedback. Additionally, SDT2 emphasises their importance in validating data type conversions, stating, "right, I'm applying `int()` to the input statements. To ensure we've got it right, I'm also adding some print statements to check

the type of inputs after conversion”, which illuminates the role of print statements in averting and diagnosing potential type-related errors.

Likewise, the flexibility of print statement debugging is further illustrated through the experiences of SDT15 and SDT14, who describe using print statements as a strategic tool to dissect program flow and troubleshoot logical discrepancies. SDT15 advocates for their use in clarifying program execution and addressing logical errors, saying, “we should maybe use some print statements to understand the flow, especially for these logical errors”. This recommendation highlights how print statements can shed light on the execution path of a program, revealing where it deviates from expected logic. Similarly, SDT14 emphasises the strategic placement of print statements for diagnostic purposes, noting, “I’m going to insert some print statements at strategic points in our code. This will help us track the values of our variables and understand where our logic is failing”. Such tactics allow programmers to chart their program’s execution comprehensively, enhancing the understanding of variable behaviour and pinpointing the root causes of logical issues. These insights collectively affirm the indispensable role of print statement debugging in enhancing code clarity and resolving complex programming challenges.

In the debugging sessions, IDEs were a key factor, as evidenced by the participants’ reliance on their advanced features for efficient problem-solving. The IDEs, with functionalities like syntax highlighting, error highlighting, auto-indentation, and breakpoints, played a central role in identifying and resolving syntax errors, understanding execution flow, and performing in-depth variable analysis. SDT1’s comment, “I’m also keeping an eye on the IDE’s syntax highlighting feature. It’s really

helping to spot these kinds of errors much quicker”, highlights the effectiveness of syntax highlighting in speeding up error detection. SDT2 also appreciates the IDE’s auto-indent feature, saying, “while you’re fixing that, I’ll take advantage of the IDE’s auto-indent feature”, acknowledging its assistance in maintaining code structure. The use of IDEs also extends to deeper code analysis, as indicated by SDT18, who says, “Set a breakpoint and step through the code to catch any subtle errors”, and SDT22’s remark, “I’ve taken the helm now. Let’s harness the IDE debugger for a deeper analysis”. This emphasis on advanced debugging tools and user-friendly interfaces, as noted by SDT25, SDT29, and SDT30, showcases their importance in tracking elusive errors and helping beginners navigate complex code. SDT25 mentions using a code comparison tool, SDT29 talks about setting breakpoints for pinpointing deviations, and SDT30 appreciates the user-friendly interface of Visual Studio, all underlining the significant impact of IDEs in enhancing the debugging process.

Furthermore, as demonstrated by SDT3 and SDT4, the slicing technique in debugging effectively simplified and enhanced the efficiency of handling complex scripts. SDT3’s strategy, “let’s isolate the block of code responsible for calculating gross pay. If we comment out the rest and test this section alone, we might find the source of our logical errors more efficiently”, exemplifies a targeted slicing method, isolating specific functionalities like gross pay calculation for more streamlined error detection. Meanwhile, SDT4 accentuate the foundational importance of input validation with “I think the issue might be in how we’re handling the input validation. Let’s temporarily remove other functionalities and just run the input section to see if it’s working as expected”, emphasising the need to verify basic operations to prevent cascading errors.

Apart from that, SDT3's approach to dissecting complex logic, "Let's break down the tax calculation logic and test each condition separately. This way, we can determine exactly which part of the logic is causing the error", highlights the effectiveness of a granular analysis in debugging, especially for uncovering intricate logical errors by testing individual conditions independently.

The Rubber Duck Debugging strategy, as demonstrated by SDT13, SDT8, and SDT7, highlights the importance of verbalising and methodically reviewing code to uncover overlooked errors. SDT13's approach, "Okay, SDT14, let me talk you through the logic of this tax calculation part as if I'm explaining it from scratch. Sometimes, saying it out loud helps me catch something I might have missed", exemplifies this technique by articulating the logic behind the tax calculation as if to a novice or a rubber duck, facilitating the discovery of minor aspects. Also, SDT8's request, "While you go over the string concatenation, I'll act as if I'm hearing this for the first time. Explain it to me step by step; it might help us spot where the syntax is off", encourages a detailed breakdown of the process, advancing a meticulous reconsideration, crucial for revealing hidden syntax errors. SDT7 further reinforces this approach by deciding to narrate each step in fixing a runtime error, believing that "Walking through it verbally often makes me see things in a different light, like having a fresh pair of eyes on the problem", thereby acknowledging the effectiveness of Rubber Duck Debugging in gaining new perspectives and revealing hidden flaws.

The significance of code review in ensuring code quality and functionality is highlighted through the experiences and suggestions of several apprentices, including SDT6, SDT7,

SDT14, SDT13, SDT17, and SDT18. SDT6's observation, "during our code review, we noticed the function was not returning the correct value", emphasises the role of code review in identifying discrepancies in code functionality. This critical evaluation is essential for ensuring that the code behaves as intended. SDT7's proposal, "taking back control now. I think we should review the entire script again to check for any errors we might have missed", demonstrates the thoroughness required in debugging, focusing on the overall structure and coherence of the code.

Similarly, SDT14's call for a "comprehensive review of the script's logic to catch any remaining errors we might have overlooked", points to the importance of detailed analysis, particularly for elusive logical errors. SDT13's satisfaction, "I think we've done a thorough job on the script. All functions appear to be working as intended, and the code is much more readable now", reflects the dual goal of code reviews, such as, enhancing functionality and readability for future maintenance and development. SDT17's suggestion, "let's take a moment for a quick code review. We should scan for any similar syntax errors, ensuring our code is structurally sound", and SDT18's meticulous check, "Scanning through the script... All other conditional statements seem fine. No more missing colons in this section", both highlight the need for ongoing vigilance and attention to detail in coding, especially for syntax and structural integrity, to prevent minor errors from escalating.

Moreover, the utility of pattern matching as a debugging strategy is exemplified in the insights shared by SDT8, SDT10, SDT13, and SDT15. SDT8's detection of a 'TypeError', as noted in "that's a TypeError. Seems like a variable is not of the expected type. Maybe

something to do with input conversion?” showcases the identification of a common programming issue related to variable types and suggests a practical solution involving data type conversion. This reflects an acute understanding of type-related errors crucial for robust coding. SDT10, in “That’s a quick fix. Just add the parentheses around the print statement”, demonstrates a rapid identification of a syntax error, common in Python 3, highlighting the significance of language-specific knowledge for efficient debugging. SDT13’s remark, “this error looks similar to one we encountered before. Let’s apply the same fix”, underlines the role of experience and pattern recognition in coding, using past issues to guide current problem-solving. Similarly, SDT15’s observation, “we’ve seen this pattern of mistakes; let’s check if it’s the same issue here”, emphasises the importance of recognising and learning from recurring issues, facilitating quicker diagnosis and proactive error prevention. These insights illustrate how awareness of common errors and patterns can enhance debugging efficiency and effectiveness.

In addressing programming challenges, SDT19 and SDT20 reveal the application of variable tracing and code review in their debugging processes. SDT20’s intention to use variable tracing for monitoring tax calculations, as stated in “I’m thinking of using variable tracing to monitor the tax calculations closely”, illustrates a strategic approach to understanding and rectifying complex computational tasks. Meanwhile, SDT19 identifies an infinite loop error in “we have an infinite loop error. We need to check the loop condition and make sure it’s set up correctly”, emphasising the need to scrutinise loop conditions to resolve such issues. SDT20 further pinpoints the cause of this error in “We’re hitting an infinite loop due to the missing colon in the ‘for’ loop”, showcasing the significance of attention to syntactical details in programming. The collaborative dynamic

is highlighted in SDT19's reminder about their pair programming schedule in "Let's review that section of the code. Also, remember, it's almost time for us to switch roles as part of our pair programming arrangement", suggesting a structured and team-oriented approach to problem-solving. Finally, SDT20's method of tracing from the error message back to the problematic function call, as mentioned in "I traced back from the error message to the problematic function call", demonstrates a systematic technique for identifying and addressing the root causes of programming errors. These insights jointly highlight the importance of thorough analysis, attention to detail, and collaboration in effective debugging and code development.

Nonetheless, the debugging process in software development is characterised by the innovative combination of multiple strategies, as illustrated by the experiences of several participants. SDT18 and SDT20 demonstrate the synergy of print statement debugging with IDE debuggers. SDT18 states, "I combined print statements with the debugger to track variable changes". This technique reinforces the value of blending traditional print debugging with advanced IDE tools to achieve a more comprehensive grasp of variable dynamics. Similarly, SDT20's method, "using prints alongside the step-through debugger helped isolate the issue", showcases the effectiveness of this composite strategy in isolating and resolving specific problems in the code.

Further blending of techniques is seen in the approaches of other participants. SDT23, who combined rubber duck debugging with code review, remarks, "explaining each line to you during review helped identify the misplaced loop". This highlights how articulating the code line-by-line can enhance clarity and lead to the discovery of errors. The fusion

of trial and error with pattern matching is exemplified by SDT22 and SDT24. SDT22's observation, "after several attempts, I recognised a pattern similar to an earlier bug", along with SDT24's approach, "we used trial and error, then matched the pattern to a previous solution", demonstrates the effectiveness of iterative testing in recognising and applying solutions to recurrent problems. Tinkering in conjunction with tracing is adopted by SDT25 and SDT27. SDT25's method, "I tinkered with the code while tracing the execution path", and SDT27's technique, "modifying and tracing the function helped us understand the underlying issue", both highlight the value of hands-on manipulation and careful tracking for a deeper understanding of coding issues. Lastly, SDT26 and SDT28 showcase the integration of slicing with print statement debugging. SDT26 explains, "I sliced the function and added print statements in each section", while SDT28 describes, "breaking down the script and using prints in each block was enlightening". These methods illuminate how dissecting code combined with strategic print statements can illuminate complex issues. These varied combinations of strategies reflect the wide-ranging and adaptive nature of debugging in software development, emphasising the need for flexibility and creativity in resolving coding challenges.

5.1.3 Theme 3: Error Spectrum

The 'Error Spectrum' was another prominent theme in the debugging transcripts, vividly portrayed through the participants' experiences. As seen in Table 18, three major error types are prominent in the video analysis and the observation notes; these are diverse error types, ranging from simple syntax slip-ups and syntax errors to complex logical oversights, logical errors and execution hurdles associated with runtime errors, emphasising the complex nature of debugging.

Table 18: Error Spectrum Subthemes in Dyadic Debugging Sessions

Subthemes	Description
Syntax Errors	Participants encountered various syntax errors, including missing colons, indentation mistakes, and typographical errors, illustrating how minor code slip-ups can disrupt execution.
Logical Errors	Logical errors were prominent, such as incorrect tax calculations, flawed loop logic, and floating-point precision issues, underscoring the need for rigorous logic validation.
Runtime Errors	Issues like unconverted string inputs, null pointer exceptions, and function parameter mismatches demonstrated how improper data handling can cause program crashes.
Ambiguous Errors	Some errors, such as incorrect operators, string concatenation issues, and data type mismatches, blurred the lines between syntax, logic, and runtime errors, highlighting the complexity of debugging.

Participants identified a variety of syntax errors, reinforcing the critical nature of precise coding. SDT1 observed, “looks like we’ve hit our first syntax error, it’s missing a colon at the end of the function definition”, pinpointing a common yet crucial mistake. Echoing this attention to detail, SDT4 found “an indentation error, missing colon, in our if-else block”, bringing to the fore how such oversights can disrupt code logic. The simplicity of syntax errors was further illustrated by SDT9, who stated, “this ‘def’ misspelled, a typo in the function declaration, is causing trouble”, drawing attention to how minor typographical errors can lead to significant problems. SDT11 added to this theme by identifying “a missing parenthesis, missing colon, in our print statement”, a small error with potentially large consequences. SDT17 addressed a less obvious syntax issue, noting “there’s a syntax error, wrong comparison operator, in our if-else statement”, which could lead to logical errors in the program. SDT18 addressed a compound issue by suggesting, “we have encountered an undefined variable error, Infinite loop due to missing colon in ‘for’ loop, in our script”, illustrating how syntax errors can cause major

runtime problems. Each instance suggests the importance of meticulous syntax in programming, where even minor errors can have significant impacts.

Participants also encountered several logical errors that challenged the integrity of their code. SDT3 identified an issue with the settings, stating, “we’re dealing with a logical error. The tax rate is incorrectly set here”, pointing out a fundamental mistake in the application’s logic. Similarly, SDT7 discovered a flaw in the main function, “we’ve got a name comparison issue here, a logical error, it’s wrong tax calculation logic”, highlighting a critical oversight in the program’s core functionality. SDT8 faced a more complex issue, “our loop logic is flawed, resulting in an infinite loop error, attempt to use an undefined variable”, illustrating the cascading effects of logical errors on program flow. SDT21 dealt with a minor but consequential problem, “we’ve got a problem with scope here, a typical logical error in variable handling”, demonstrating how mismanagement of variable scope can disrupt a program’s operation. SDT26 dealt with a numerical precision challenge, “we’ve got a floating-point precision error in our calculations”, shedding light on the intricacies of handling numerical data. Each of these instances emphasises the necessity for rigorous logical scrutiny in software development, where overlooked details can lead to significant operational flaws.

Participants also encountered various runtime errors that hindered their progress. SDT5 identified a fundamental conversion issue, stating, “the script threw a runtime error; it’s not converting string input to number before mathematical operations because of unconverted string inputs”, pointing out a critical oversight in data handling. SDT14 faced a null pointer exception, “this section’s throwing a null pointer exception, definitely a

runtime issue”, highlighting a common but serious error in accessing uninitialised memory or objects. In a similar vein, SDT19 tackled a type conversion problem, “this segment is throwing a type conversion error, needs fixing”, affirming the importance of ensuring data types are correctly managed. SDT27 encountered a function call issue, “there’s a mismatch in function arguments, causing a parameter error”, illuminating the complexities and possible drawbacks in function parameter management. Each incident reflects the complexity of runtime errors in software development, where incorrect handling of data types, memory, and function parameters can lead to significant issues in program execution.

However, certain errors defy straightforward classification, as highlighted by participants who encountered ambiguous issues. SDT2 pointed out a common error in operator usage, stating, “Ah, Wrong operator for multiplication. We’ve used ‘x’ instead of ‘*’, a classic multiplication operator error”. This error’s nature could swing between a syntax or logical error, depending on the context and language used. Similarly, SDT6 discovered a less apparent issue in string operations, “just spotted mismatch in string concatenation, a string concatenation mistake in our return statement”. This could either be a syntax error affecting code structure or a logical error where the code’s syntax is correct but fails to execute as intended. As a final point, SDT12 faced a data type mismatch, “we’ve got a data type mismatch error, something’s not adding up right”, an issue that could manifest as a logical or runtime error, depending on its effect on the program’s functionality. These situations expose the fine-grained characteristics of coding errors, where the line between different error types can be blurred, reflecting the complex and layered challenges in software development.

5.1.4 Theme 4: Cognitive Load Management

Cognitive Load Management emerged as a pivotal theme, capturing the participants' strategic efforts to distribute mental effort effectively. As seen in Table 19, this theme encompasses various subthemes, including verbalising thought processes, role-switching, and structured debugging approaches, all of which played a crucial role in mitigating cognitive strain and enhancing collaboration during debugging sessions.

Table 19: Cognitive Load Management Subthemes in Dyadic Debugging Sessions

Subthemes	Description
Task Segmentation and Role Division	Participants organised debugging by listing errors, breaking down problems, and alternating tasks, effectively balancing workload to boost efficiency and reduce overwhelm.
Managing Distributed Cognitive Load	The participants collaboratively managed cognitive load through structured debugging, record-keeping, code reviews, and alternating coding and reviewing for accuracy and efficiency.
Collaboration and Team Dynamics	Participants leveraged teamwork strategies like structured time management, role flexibility, and reflective pauses to sustain productivity and balance cognitive load during debugging.
Task Execution and Process Improvement	Pre-emptive planning, role swapping, iterative improvement, and prioritising critical errors helped optimise workflow efficiency and cognitive resource distribution.

The dyads' practices, such as task segmentation and role division, played a crucial role. For example, in a focused approach to managing distributed cognitive load sharing, participants from DYAD4 and DYAD5 shared strategies that emphasise systematic processing and division of labour. SDT7 suggests a methodical first step by "listing out all the errors first, then we'll address them systematically", setting the stage for an organised problem-solving. Complementing this, SDT8 proposes a division of focus, where one handles "runtime errors" and the other tackles "syntax errors", demonstrating a tailored approach to distribute cognitive demands according to individual strengths. From DYAD5, SDT9 introduces a prioritisation strategy, focusing on "the errors that seem most critical",

which ensures that efforts are concentrated where they are most needed. SDT10 further refines this approach by “Breaking down complex problems into smaller tasks”, enabling a more manageable and less overwhelming process of troubleshooting. These strategies illustrate jointly an approach to workload distribution, ensuring that cognitive resources are optimally allocated to enhance efficiency and accuracy in problem-solving.

In addressing the management of distributed cognitive load, with a focus on structured debugging, SDT11 and SDT12 present a detailed strategy that affirms the importance of a structured and systematic approach to debugging. SDT11’s suggestion to “tackle the errors one at a time to avoid getting overwhelmed” introduces a methodical way of breaking down the complexity, aiming to minimise cognitive overload by focusing on individual issues sequentially. Complementing this, SDT12’s commitment to “note down each error and our approach in resolving it” offers a record-keeping practice that ensures transparency and aids in tracking progress. Further, SDT11 advocates for periodic code reviews “to catch any missed errors”, highlighting the proactive measures taken to ensure thoroughness and accuracy in their work. SDT12’s strategy to “alternate between coding and reviewing” proposes a dynamic workflow that facilitates error detection and maintains a balance between creation and analysis, leading to a more efficient debugging process. These approaches, in a way, point to a collaborative effort towards distributed cognitive load management, focusing on precision, accountability, and a strategic division of tasks to enhance problem-solving effectiveness.

In the context of collaboration and team dynamics, the dyads portray a strategic approach to managing distributed cognitive load through various techniques to enhance teamwork efficiency during the debugging process. DYAD1’s SDT1 suggests taking a

moment to recap progress, addressing the risk of becoming overwhelmed, thereby emphasising the importance of reflective pauses to maintain clarity and focus. DYAD7 introduces structured time management and role flexibility, with SDT14 advocating for the use of timers during debugging phases for better time allocation, and SDT13 proposed role switching to gain fresh perspectives, showcasing methods to keep the cognitive load balanced and ensure sustained productivity. DYAD8, through SDT15 and SDT16, highlighted the value of collaborative problem-solving and leveraging individual strengths, suggesting working together on complex parts and combining syntax and logical analysis skills to form a complementary team dynamic. These strategies, in a way, illustrate a thoughtful approach to workload distribution, focusing on maintaining momentum, leveraging diverse skills, and periodically reassessing team strategy to optimise performance and mitigate cognitive overload.

In the realm of task execution and process improvement, participants from DYAD10, DYAD13, and DYAD15 offered insightful strategies for managing distributed cognitive load sharing effectively. Beginning with DYAD10, SDT19 advocated for a planned approach before coding, paired with SDT20's suggestion for role swapping to maintain fresh perspectives, drawing attention to the value of pre-emptive planning and flexibility in role allocation as methods to distribute cognitive load efficiently. SDT19 further emphasises the need for wise management of cognitive resources, aligning with the overarching theme of sustainable workload distribution. Moving to DYAD13, SDT26 and SDT25 discussed iterative improvement and continuous code refinement as mechanisms for gradual learning and error reduction, highlighting an ongoing commitment to evolution and quality enhancement. DYAD15's contributions, with SDT29 and SDT30, stressed

prioritising critical errors, efficient resource use, focusing on impactful errors, strategic task allocation, and periodic reassessment of priorities. These strategies also indicate a sophisticated approach to workload management, where planning, adaptability, focused efforts, and strategic reassessments converge to optimise distributed cognitive load sharing, thus fostering a more effective and efficient debugging and development process.

5.1.5 Theme 5: Challenges Faced

Challenges Faced by the dyad emerged as a significant theme highlighting the multifaceted difficulties encountered. As seen in Table 20, this theme comprises various subthemes, including technical proficiency and error resolution, cognitive and workflow management, collaboration and communication dynamics, and tool and resource utilisation, all of which influenced the dyads' abilities to navigate debugging tasks effectively.

Table 20: Challenges Faced Subthemes in Dyadic Debugging Sessions

Subthemes	Description
Technical Proficiency and Error Resolution	Participants faced difficulties with unfamiliar programming languages, recurring errors, and logical complexities, highlighting the challenge of mastering debugging techniques.
Cognitive and Workflow Management	Frequent context switching, role transitions, and overwhelming workloads contributed to cognitive strain, necessitating structured strategies for maintaining focus and efficiency.
Collaboration and Communication Dynamics	Misalignment in problem-solving approaches, difficulties in articulating thoughts, and ineffective communication hindered smooth collaboration and debugging progress.
Tool and Resource Utilisation	Struggles with non-intuitive debugging tools, unfamiliar development environments, and poorly documented code exacerbated the challenges of efficient error resolution.

For instance, while navigating through the collaborative debugging task, participants encountered various technical challenges, vividly captured through their personal

reflections. The journey begins with SDT1's confusion, "I'm lost with this syntax. It's nothing like what I've worked with before", and SDT6's acknowledgement of unfamiliar territory, "I'm not very familiar with this programming language, which makes debugging challenging". These statements lay the groundwork for understanding the difficulties faced due to unfamiliar coding environments and languages. The recurrence of errors, as expressed by SDT5, "this error keeps recurring. It feels like we're missing something fundamental", further illustrates the struggle to grasp core issues within the code.

The complexity of logical errors becomes apparent through SDT2's observation, "these logical errors are trickier than I thought. It's hard to get the logic right", while SDT16's frustration, "every fix seems to introduce a new problem. It's frustrating", encapsulates the cyclical nature of debugging. SDT17's remark, "the complexity of this code is beyond what I've handled before", and SDT9's challenge in spotting "small syntax errors" denote the daunting task of navigating complex code. Adding to the depth of challenges, SDT13 admits, "some of these errors are beyond my current knowledge base", highlighting the learning curve involved. Similarly, SDT26's insight, "the logic behind these functions is not what I expected. It's confusing", and SDT27's self-doubt, "I keep second-guessing myself. Am I fixing this the right way?" reflect the cognitive and emotional hurdles in debugging. These experiences shared by each of the apprentices portray a varied landscape of the debugging process, marked by technical, cognitive, and emotional challenges as participants grapple with unfamiliar syntax, logical complexities, and the thorough insight necessary for proficiently managing and fixing errors.

Also, navigating the labyrinth of collaborative debugging, participants revealed an array of challenges around cognitive and workflow management, each illuminating different aspects of the ordeal. Starting with SDT3's revelation, "constantly switching between different parts of the code is really disorienting", the narrative unveils the cognitive turmoil triggered by incessant shifts in attention. This sense of disorientation resonates with SDT12's admission, "the constant role-switching is making it hard to maintain a train of thought", underlining the struggle to stay focused amidst ongoing transitions. The confession by SDT4, "I'm struggling to keep up with the pace. This is more complex than I expected", and SDT21's assertion, "sometimes I feel overwhelmed by the sheer volume of issues to address", bring to light the overwhelming complexity and breadth of debugging activities. This quest for clarity amid chaos is echoed in SDT7's frustration, "the more we fix, the more issues seem to arise. It's like a never-ending cycle", capturing the cyclical nature of their task.

The discourse then expands to include efficiency and strategic planning, or rather the lack thereof, with SDT8 and SDT22 voicing concerns over the monumental task of error management and the challenge of prioritisation. SDT10 adds another layer to the struggle, noting, "understanding this existing codebase is tough. It's not well-documented", pinpointing the difficulties posed by insufficient documentation. The dialogue shifts to strategy and methodology, with SDT18's remark, "keeping track of all the changes and errors is quite a task", spotlighting the logistical challenges of monitoring progress. Also, doubts about their chosen approach are succinctly expressed by SDT19, "I'm not sure of the obstacles", and the task's emotional impact emerges, with SDT29 stating, "It's challenging to remain focused with so many different types of errors", and

SDT30 sharing, “I’m feeling the pressure with the amount of work we need to get through”, reflecting the stress and pressure inherent in the debugging process. SDT25’s introspection, “balancing between fixing errors and understanding the code is tough”, indicates the delicate act of navigating between correction and comprehension.

These narratives weave together a story filled with cognitive, logistical, and emotional complexities, offering an in-depth look at the collaborative debugging journey. From SDT3’s insights on the disorientation caused by frequent context switches to SDT25’s struggles with balancing error correction and code understanding, participants’ candid quotes highlight the composite nature of debugging in a team setting. This comprehensive depiction sheds light on the participants’ technical and emotional battles and reiterates the need for systematic approaches and efficient strategies to navigate the intricate process of debugging collaboratively.

In the intricate process of collaboratively debugging Python code, the dynamics of collaboration and communication among participants illuminate the pluralistic challenges encountered. Starting with SDT11’s candid admission, “I’m finding it hard to articulate my thoughts clearly to my partner”, we look at the complexities involved in conveying detailed technical concepts within a team. This difficulty in communication is not isolated, as SDT14 reveals, “aligning my coding approach with my partner’s suggestions is proving difficult”, reinforcing the hurdles in meshing diverse problem-solving strategies. The sentiment of misalignment in collaboration is further echoed by SDT20, who states, “I feel like we’re not communicating effectively. It’s impacting our progress”, highlighting the direct impact of communication barriers on the efficiency of debugging efforts. These

admissions collectively paint a picture of a collaborative environment where the challenges extend beyond the technical aspects of debugging to include the critical, yet often overlooked, elements of clear communication and effective teamwork. Through these reflections, the narrative shifts from individual coding struggles to a broader examination of how collaborative dynamics influence the outcome of joint debugging tasks.

To add to that, the context of collaboratively debugging Python code using tools and resources emerges as a significant hurdle for participants, shedding light on various aspects of the debugging process. Beginning with SDT15's frustration, "I'm struggling with the debugging tools. They're not very intuitive", we uncover the initial layer of complexity that non-intuitive tools add to the debugging process. This struggle centres around the tools and how their design can impede the progress of those unfamiliar with their intricacies. Moving forward, SDT23's experience, "I'm not used to this development environment, so it's slowing me down", further complicates the situation. This statement reveals the adjustment challenges faced when navigating unfamiliar development environments, highlighting how such unfamiliarity can directly slow down the debugging process. Transitioning to a new environment requires learning its functionalities and adapting one's debugging strategy to fit its constraints.

Furthermore, the issue of poorly documented code is brought to the fore by SDT24, who points out, "the lack of comments in the code is making it hard to understand the intent". This highlights another dimension of the challenge, for example, the difficulty of deciphering code without adequate documentation. Understanding the original

programmer's intent becomes a task in itself, adding another layer of complexity to the debugging effort. These insights stress the broader challenges of tool and resource utilisation in collaborative debugging. They reflect on the technical difficulties posed by unfamiliar or poorly designed tools and environments and the importance of clear documentation in facilitating a smoother debugging process. Through these participant experiences, we understand the additional obstacles that tools and resources can present in the collaborative debugging of Python code.

In conclusion, Tables 21 and 22 summarise the observations from the debugging session notes and video analysis. It encapsulates the situation during and at the conclusion of the debugging sessions, with a focus on the five themes previously identified. It illustrates the number of bugs each pair was unable to resolve, the debugging strategies and tactics employed, the use of technology, how they manage the sharing of cognitive load, and the challenges encountered while collaboratively debugging the Python code.

Table 21: Outline of the debugging sessions' core findings

Pair / Participant	Error Spectrum	Debugging Strategies & Tactics	Technology Utilisation	Cognitive Load Management	Challenges Faced
DYAD1 SDT1 & SDT2	5 Syntax Errors, 2 Logical Errors and 1 Runtime Errors	Print Statement Debugging, IDE Debuggers, Code Review, Tinkering, Tracing	IDE Debuggers, Visual Studio. Microsoft Team, Python, Live Share, GitHub	Collaboration and Team Dynamics	<ul style="list-style-type: none"> • Technical Proficiency and Error Resolution
DYAD2 SDT3 & SDT4	0 Syntax Errors, 2 Logical Errors and 3 Runtime Errors	Print Statement Debugging, Tinkering, IDE Debugger, Slicing, Code Review	IDE Debuggers, Visual Studio. Microsoft Team, Python, Live Share, GitHub	Task Execution and Process Improvement	<ul style="list-style-type: none"> • Cognitive and Workflow Management
DYAD3 SDT5 & SDT6	6 Syntax Errors, 6 Logical Errors and 2 Runtime Errors	Tinkering, Trial & Error, Print Statement Debugging, Divide & Conquer	Visual Studio. Microsoft Team, Python, Live Share, GitHub	Task Execution and Process Improvement	<ul style="list-style-type: none"> • Technical Proficiency and Error Resolution
DYAD4 SDT7 & SDT8	2 Syntax Errors, 2 Logical Errors and 1 Runtime Errors	Rubber Duck Debugging, Tinkering, Print Statement Debugging, Code Review	Visual Studio. Microsoft Team, Python, Live Share, GitHub	Workflow Management and Strategy	<ul style="list-style-type: none"> • Cognitive and Workflow Management

Table 22: Outline of the debugging sessions' core findings (Continuation)

Pair / Participant	Error Spectrum	Debugging Strategies & Tactics	Technology Utilisation	Cognitive Load Management	Challenges Faced
DYAD5 SDT9 & SDT10	2 Syntax Errors, 5 Logical Errors and 2 Runtime Errors	Trial & Error, Tinkering, Tracing, Print Statement Debugging	Visual Studio. Microsoft Team, Python, Live Share, GitHub	Workflow Management and Strategy	<ul style="list-style-type: none"> • Technical Proficiency and Error Resolution • Cognitive and Workflow Management
DYAD6 SDT11 & SDT12	7 Syntax Errors, 6 Logical Errors and 2 Runtime Errors	Trial & Error, Code Review, Tinkering, Print Statement Debugging	Visual Studio. Microsoft Team, Python, Live Share, GitHub	Task Segmentation and Collaboration	<ul style="list-style-type: none"> • Cognitive and Workflow Management • Collaboration and Communication Dynamics
DYAD7 SDT13 & SDT14	3 Syntax Errors, 2 Logical Errors and 2 Runtime Errors	Rubber Duck Debugging, Tinkering, IDE Debuggers, Print Statement Debugging	IDE Debuggers, Visual Studio. Microsoft Team, Python, Live Share, GitHub	Collaboration and Team Dynamics	<ul style="list-style-type: none"> • Technical Proficiency and Error Resolution • Collaboration and Communication Dynamics
DYAD8 SDT15 & SDT16	0 Syntax Errors, 4 Logical Errors and 2 Runtime Errors	Print Statement Debugging, Tinkering, Pattern Matching, Code Review	Visual Studio. Microsoft Team, Python, Live Share, GitHub	Collaboration and Team Dynamics	<ul style="list-style-type: none"> • Technical Proficiency and Error Resolution • Tool and Resource Utilisation.
DYAD9 SDT17 & SDT18	1 Syntax Errors, 3 Logical Errors and 2 Runtime Errors	Code Review, IDE Debuggers, Tinkering, Rubber Duck Debugging, Pattern Matching	IDE Debuggers, Visual Studio. Microsoft Team, Python, Live Share, Github	Collaboration and Team Dynamics	<ul style="list-style-type: none"> • Technical Proficiency and Error Resolution • Cognitive and Workflow Management
DYAD10 SDT19 & SDT20	2 Syntax Errors, 1 Logical Errors and 1 Runtime Errors	Tracing, Tinkering, Print Statement Debugging, Code Review	Visual Studio. Microsoft Team, Python, Live Share, Github	Task Execution and Process Improvement	<ul style="list-style-type: none"> • Cognitive and Workflow Management • Collaboration and Communication Dynamics
DYAD11 SDT21 & SDT22	4 Syntax Errors, 3 Logical Errors and 1 Runtime Errors	Print Statement Debugging, Tinkering, IDE Debuggers, Code Review	IDE Debuggers, Visual Studio. Microsoft Team, Python, Live Share, Github	Collaboration and Team Dynamics	<ul style="list-style-type: none"> • Cognitive and Workflow Management
DYAD12 SDT23 & SDT24	0 Syntax Errors, 2 Logical Errors and 3 Runtime Errors	Trial & Error, IDE Debuggers, Tinkering, Print Statement Debugging	IDE Debuggers, Visual Studio. Microsoft Team, Python, Live Share, Github	Collaboration and Team Dynamics	<ul style="list-style-type: none"> • Tool and Resource Utilisation
DYAD13 SDT25 & SDT26	2 Syntax Errors, 3 Logical Errors and 1 Runtime Errors	Code Review, Tinkering, Trial & Error, Pattern Matching, Print Statement Debugging	Visual Studio. Microsoft Team, Python, Live Share, Github	Task Execution and Process Improvement	<ul style="list-style-type: none"> • Technical Proficiency and Error Resolution • Cognitive and Workflow Management
DYAD14 SDT27 & SDT28	2 Syntax Errors, 5 Logical Errors and 2 Runtime Errors	Tinkering, Print Statement Debugging, Code Review	Visual Studio. Microsoft Team, Python, Live Share, Github	Task Execution and Process Improvement	<ul style="list-style-type: none"> • Technical Proficiency and Error Resolution • Cognitive and Workflow Management
DYAD15 SDT29 & SDT30	3 Syntax Errors, 2 Logical Errors and 1 Runtime Errors	Print Statement Debugging, Tinkering, IDE Debuggers, Code Review	IDE Debuggers, Visual Studio. Microsoft Team, Python, Live Share, Github	Task Execution and Process Improvement	<ul style="list-style-type: none"> • Cognitive and Workflow Management

5.2 Python Code Analysis Findings

The investigation into debugging behaviour examined the performance of fifteen dyads working on Python code embedded with 20 bugs, consisting of 11 syntax, 6 logical, and 3 runtime errors, as detailed in Table 23. Although this table provided the counts, the specific error codes are listed in Tables 24, 25, and 26, where the errors are segmented into the dyads' debugging outcomes, such as bugs found, unfound, fixed, and unfixed. This segmentation reveals their proficiency across a spectrum from lower to higher. Their success in identifying and resolving errors across the three bug categories determined the classification into lower, moderate, and high proficiency levels.

Across the proficiency spectrum, DYAD6 and DYAD3 demonstrated foundational syntax handling by addressing basic errors such as missing colons and print statement issues, with DYAD6 resolving 4 out of 8 and DYAD3 half of 11 identified errors, yet both struggled with more complex challenges like string closures and loop completions. Advancing to moderate proficiency, DYAD1, DYAD11, and DYAD12 identified all errors but were partially stymied by intricate issues like incorrect 'if' sequencing and loop errors, with their resolution rates ranging from 6 to 7 out of 11. Higher up, DYAD7, DYAD9, and DYAD15 showcased greater skill, fixing common errors efficiently but encountering difficulties with specific problems like incorrect 'if' structures and loop errors, with their success rates nearing 8 to 9 out of 11. The top performers, Dyads 10, 13, 14, 2, 4, 5, and 8, identified and adeptly resolved the most challenging errors, including complex string and loop issues, with the latter group achieving a perfect resolution record. This progression from basic to exceptional proficiency amplifies the varied learning curves and

the need for targeted learning strategies to foster comprehensive Python programming skills.

Moreover, across DYADs 1 to 15, proficiency in addressing logical errors varied significantly, showcasing diverse levels of understanding in programming logic. DYAD3 and DYAD6, at the lower end, both identified 6 logical errors but failed to resolve any, indicating a fundamental need for improvement in understanding complex logic. Similarly, DYAD14 and DYAD5 struggled, each identifying 6 errors but resolving only one, highlighting difficulties with intricate issues like tax calculation logic and bonus logic. Slightly above, DYAD8 managed to fix 2 out of 6 errors, showing a modest improvement but still facing challenges with specific logical errors. DYAD9, DYAD11, and DYAD13 demonstrated moderate proficiency; DYAD9 resolved 2 out of 5 identified errors, DYAD11 fixed 3 out of 5, and DYAD13 corrected 3 out of 6, suggesting they have a foundational grasp of programming logic with room for growth. DYAD12's performance, resolving 4 out of 6 errors, aligns them with higher proficiency, akin to DYADs 1, 4, 7, and 15, each also resolving 4 out of 6 errors. This group effectively navigated a range of logical challenges, evidencing a robust understanding of programming logic, albeit with areas for further development. DYAD10 excelled by rectifying 5 out of 6 logical errors, demonstrating an advanced understanding of Python's logical constructs and superior problem-solving skills, and setting a benchmark for proficiency among their peers.

Furthermore, the investigation into runtime error resolution among fifteen dyads revealed a spectrum of debugging proficiencies. DYAD3 and DYAD6, unable to resolve any errors, demonstrated foundational proficiency, highlighting their nascent journey in

understanding Python's runtime environment. The moderate proficiency group, including DYAD1, DYAD2, DYAD7, DYAD8, DYAD11, DYAD12, and DYAD14, showed varying degrees of success; notably, DYAD1 and DYAD7 each fixed two out of three errors, indicating a developing but incomplete mastery over runtime challenges, while the others resolved at least one, revealing gaps in their debugging capabilities. DYAD4, DYAD9, and DYAD13, each resolving two out of three errors, were classified as proficient, showcasing a strong grasp on runtime error management and systematic problem-solving skills. Standing out for their high proficiency, DYAD5 and DYAD10 flawlessly fixed all three identified errors, indicating an advanced level of debugging expertise and setting a benchmark for their peers in navigating and rectifying runtime challenges efficiently. This stratification reinforced the varied levels of understanding and skill across the dyads, from foundational to high proficiency in dealing with runtime errors.

In conclusion, the analysis of syntax, logical, and runtime error handling among Dyads stressed the need for educational strategies tailored to individual and group proficiency levels in Python programming. It revealed that while some Dyads excelled in identifying and resolving errors, others faced challenges, signalling a diverse range of skill sets and problem-solving approaches. For example, DYADs 3 and 6, struggling with logical error resolution, and DYAD6's particular difficulty with runtime errors, illustrated the necessity for foundational training in Python's logic and runtime environments. Conversely, DYAD14's limited success in addressing complex logical errors and the moderate proficiency displayed by groups like DYAD1, DYAD2, DYAD7, and DYAD8 in runtime error resolution pointed towards the need for targeted learning focused on understanding intricate logic patterns and debugging skills. The stark contrast in error resolution

capabilities, particularly the adeptness of DYAD5 and DYAD10 in navigating runtime challenges, further highlighted the spectrum of competencies within the cohort. This calls for a dedicated emphasis on developing specific skills, such as loop mechanics, variable scope comprehension, and practical debugging techniques, to enhance overall coding proficiency. By pinpointing the distinct challenges each dyad encounters, educators can customise instruction to uplift every learner's understanding and application of Python's syntactical and logical constructs, fostering a deeper and more comprehensive grasp of programming fundamentals.

Table 23: Summary of bugs discovery, successful fixing and unsuccessful fixing

Dyad ID	Gender	Programming experience	Age bracket	No of bugs	Bugs Found & Fixed												
					Syntax				Logical				Runtime				
					Found	Not Found	Fixed	Not Fixed	Found	Not Found	Fixed	Not Fixed	Found	Not Found	Fixed	Not Fixed	
DYAD1	Female & Female	Low < 2 years	16 – 18 years	20	6	5	6	5	4	2	4	2	2	1	2	1	
DYAD2	Male & Female	Low < 2 years	16 – 18 years	20	11	0	11	0	5	1	3	3	2	1	1	2	
DYAD3	Male & Male	Low < 2 years	16 – 18 years	20	7	4	5	6	3	3	0	6	1	2	1	2	
DYAD4	Male & Male	Low < 2 years	16 – 18 years	20	11	0	9	2	6	0	4	2	2	1	2	1	
DYAD5	Male & Male	Low < 2 years	16 – 18 years	20	11	0	9	2	6	0	1	5	3	0	1	2	
DYAD6	Male & Male	Low < 2 years	16 – 18 years	20	8	3	4	7	4	2	0	6	1	2	1	2	
DYAD7	Male & Male	Low < 2 years	16 – 18 years	20	11	0	8	3	6	0	4	2	1	2	1	2	
DYAD8	Female & Female	Low < 2 years	16 – 18 years	20	11	0	11	0	6	0	2	4	0	1	1	2	
DYAD9	Male & Male	Low < 1 year	18 – 25 years	20	11	0	10	1	6	0	3	3	3	0	1	2	0
DYAD10	Male & Female	Low < 1 year	18 – 25 years	20	11	0	9	2	6	0	1	1	3	0	2	1	20
DYAD11	Male & Male	Low < 1 year	18 – 25 years	20	11	0	7	4	5	1	3	3	2	1	2	1	18
DYAD12	Male & Male	Low < 1 year	18 – 25 years	20	11	0	11	0	6	0	4	2	3	0	0	3	20
DYAD13	Male & Male	Low < 1 year	25 – 50 years	20	11	0	9	2	6	0	3	3	2	1	2	1	19
DYAD14	Male & Male	Low < 1 year	25 – 50 years	20	11	0	9	2	6	0	1	5	3	0	1	2	20
DYAD15	Male & Male	Low < 1 year	25 – 50 years	20	11	0	8	3	6	0	4	2	3	0	2	1	20

Table 24: Summary of specific syntax errors breakdown by discovery and resolution.

Syntax		Participant														
		Dyad1	Dyad2	Dyad3	Dyad4	Dyad5	Dyad6	Dyad7	Dyad8	Dyad9	Dyad10	Dyad11	Dyad12	Dyad13	Dyad14	Dyad15
Errors – Found	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01
	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02
	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03
	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04
	SE05	SE05	SE05	SE05	SE05	SE05	SE05	SE05	SE05	SE05	SE05	SE05	SE05	SE05	SE05	SE05
	SE06	SE06	SE06	SE06	SE06	SE06	SE06	SE06	SE06	SE06	SE06	SE06	SE06	SE06	SE06	SE06
	SE07	SE07	SE07	SE07	SE07	SE08	SE07	SE07	SE08	SE07	SE07	SE07	SE07	SE07	SE07	SE07
	SE08		SE08	SE08		SE09		SE08	SE08	SE08	SE08	SE08	SE08	SE08	SE08	SE08
	SE09		SE09	SE09		SE09		SE09	SE09	SE09	SE09	SE09	SE09	SE09	SE09	SE09
	SE10		SE10	SE10		SE10		SE10	SE10	SE10	SE10	SE10	SE10	SE10	SE10	SE10
	SE11		SE11	SE11		SE11		SE11	SE11	SE11	SE11	SE11	SE11	SE11	SE11	SE11
Errors – Not Located	SE07	None	SE08	None	None	SE07	None	None	None	None	None	None	None	None	None	None
	SE08		SE09			SE10										
	SE09		SE10			SE11										
	SE10		SE11													
	SE11															
Errors – Fixed	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01	SE01
	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE02	SE03
	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE03	SE04
	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE04	SE05
	SE05	SE05	SE05	SE05	SE05	SE06	SE05	SE05	SE06	SE05	SE05	SE05	SE05	SE05	SE05	SE06
	SE06	SE06	SE06	SE06	SE08	SE06	SE07	SE06	SE07	SE06	SE06	SE06	SE06	SE06	SE06	SE07
	SE07		SE07	SE09		SE07		SE08	SE07	SE08	SE09	SE09	SE07	SE08	SE08	SE08
	SE08		SE08	SE10		SE09		SE08	SE08	SE09	SE10	SE10	SE11	SE08	SE08	SE09
	SE09		SE09	SE11		SE11		SE10	SE10	SE11	SE11	SE11	SE11	SE09	SE10	SE10
	SE10		SE11													
	SE11															
Errors – Not Fixed	SE07	None	SE06	SE10	SE05	SE03	SE05	None	SE05	SE07	SE08	SE08	None	SE10	SE07	SE02
	SE08		SE07	SE11	SE07	SE05	SE07		SE05	SE07	SE09	SE09		SE11	SE08	SE10
	SE09		SE08			SE08			SE09							
	SE10		SE09			SE10			SE10							
	SE11		SE10			SE11			SE11							

Table 25: Summary of specific logical errors breakdown by discovery and resolution.

Table 26: Summary of specific runtime errors breakdown by discovery and resolution

Runtime		Participant															
		Dyad1	Dyad2	Dyad3	Dyad4	Dyad5	Dyad6	Dyad7	Dyad8	Dyad9	Dyad10	Dyad11	Dyad12	Dyad13	Dyad14	Dyad15	
	Errors – Found	RE01 RE02	RE01 RE02	RE01	RE01 RE02	RE01 RE02 RE03	RE01	RE01	RE01 RE02	RE01 RE02	RE01 RE02	RE01 RE02 RE03	RE01	RE01 RE02	RE01 RE02 RE03	RE01 RE02	RE01 RE02 RE03
	Errors – Not Located	RE03	RE03	RE02 RE03	RE03	None	RE02 RE03	RE02 RE03	None	None	RE03	None	RE03				
	Errors – Fixed	RE01 RE02	RE01	RE01	RE01 RE02	RE03	RE01	RE01	RE01	RE01 RE03	RE01 RE02	None	RE01 RE02	RE01	RE01	RE01 RE02	
	Errors – Not Fixed	RE03	RE02 RE03	RE02 RE03	RE03	RE01 RE02	RE02 RE03	RE02 RE03	RE02 RE03	RE02	RE03	RE01 RE02 RE03	RE03	RE02 RE03	RE03		

5.3 Interview Session Findings

The dyads' interview transcripts unveil and clarify some of the thoughts behind some actions taken during the debugging that were not entirely captured through the actions seen and from the think-aloud verbal protocol. A holistic examination of the dyads' interview sessions using Braun and Clark's thematic analysis (see Section 4.4.4) reveals three key themes, as outlined in Table 27. In particular, the spectrum of errors, the combination of technical and cognitive skills, and challenges arising from collaboration.

Table 27: Overview of key themes in interview sessions

Themes	Description
Theme 1: Error Spectrum	The data highlights participants' debugging progression, from syntax errors as a foundation to logical and runtime errors, which require deeper problem-solving and execution flow understanding.
Theme 2: Technical and Cognitive Skills	The data highlights participants' technical and cognitive skills, focusing on IDE tool usage, structured debugging strategies, and cognitive load management through collaboration and role distribution.
Theme 3: Challenges	The data highlights participants' challenges in remote debugging, including communication barriers, cognitive strain, and logistical constraints, requiring coordination, adaptability, and strategic problem-solving.

5.3.1 Theme 1: Error Spectrum

The exploration of the error spectrum in dyad interviews highlighted the range of programming challenges encountered during debugging. This theme is central to unravelling the complexities of debugging, offering insights into novice programmers' varied skill levels, problem-solving techniques, and learning progression. Far from being mere obstacles, these errors are valuable indicators for skill evaluation and development.

The analysis, as outlined in Table 28, categorises errors into syntax, logic, and runtime errors, each reflecting a unique challenge and requiring specific skills for resolution.

Table 28: Error Spectrum Subthemes in Interview Sessions

Subthemes	Description
Syntax Errors	Participants identified syntax errors as the most common and easiest to fix, often caused by typographical errors, missing elements, or structural inconsistencies, making them a fundamental first step in debugging.
Logical Errors	Debugging logical errors proved challenging as they required a deep understanding of both programming logic and the underlying problem domain, often leading to frustration and a steep learning curve.
Runtime Errors	Participants found runtime errors particularly difficult due to their reliance on understanding execution flow, with issues like infinite loops and data type mismatches highlighting gaps in programming experience.

Starting with the syntax error, participants uniformly acknowledged the primacy of syntax errors in their initial diagnostic efforts. SDT3's observation that "syntax errors were usually the first thing we looked for in debugging" signifies a common strategy among the cohort, reflecting a foundational approach to troubleshooting code. This sentiment was echoed by SDT19 and SDT17, who noted that these errors "were typically related to incorrect code structure" and "were usually due to overlooking Python's rules", respectively, highlighting common pitfalls in adhering to the language's syntax requirements. SDT8 and SDT11 further pointed out that such errors "were often about small typos or forgotten elements" and their identification was "crucial in the initial phase of debugging", indicating that these mistakes, while minor, were significant barriers to code execution. The ease of resolving these issues was a recurrent theme, with SDT12 and SDT15 describing syntax errors as "often the easiest to diagnose and fix" and "the most common and the easiest to fix", suggesting a contrast between their frequency and the simplicity of their resolution. SDT7's mention of "mismatch in string concatenation was a typical syntax error that we frequently encountered" adds specificity to the types

of syntax issues commonly faced, presenting a tangible example of the errors that participants navigated. Collectively, these reflections paint a picture of debugging as a process where identifying and correcting syntax errors forms the bedrock of resolving more complex issues, marked by a shared understanding of these errors' nature and their role in the debugging hierarchy.

Furthermore, as voiced by participants, the challenge of debugging logical errors in Python highlights a complex journey through the intricate landscape of programming logic and syntax, particularly for those with limited prior experience. Initially, SDT1's observation that "these errors required a deep understanding of Python's logic and syntax, which was challenging given our limited experience", captures a common sentiment that resonates across the group. Subsequently, this struggle is echoed by SDT4, who conceded, "however, I struggled with some of the more complex logical errors", emphasising the steep learning curve encountered. Furthermore, SDT10's recounting, "for instance, fixing the tax calculation logic in the script was tough, and I struggled with understanding the deeper logic required for tax deduction conditions", alongside SDT7's admission of frustration, "there were moments of frustration, especially when dealing with complex logical errors like bonus calculation that lacks context"; both underline the arduous task of navigating through errors that necessitate a profound dive into the code and its underlying business logic. Moreover, this perspective is solidified by SDT16, who reflected, "for me, these logical errors are challenging because they require a deep understanding the code and the underlying business logic".

Additionally, the task of articulating complex debugging processes, especially within collaborative or remote settings, was brought to light by SDT9 and SDT11. Particularly, SDT9 highlighted, “it was difficult to convey my reasoning and thought process solely through verbal explanations”, while SDT11 disclosed, “one significant obstacle we encountered during our debugging session was the complexity of managing and understanding the program’s logic from a remote location”. These revelations expose an additional layer of complexity introduced by remote collaboration on intricate debugging tasks. Moreover, SDT15’s assertion, “these types of errors require coding skills and a deep understanding of the problem domain”, coupled with SDT21’s observation, “for instance, when we encountered the logical error involving the misuse of the special variable ‘name’, it wasn’t just a matter of syntax but understanding the conceptual use of this Python construct”, highlight the crucial intersection of coding proficiency and domain-specific knowledge in surmounting logical errors.

Concluding the logical errors discourse, this collective reflection from participants sheds light on the technical hurdles faced when debugging logical errors. It also accentuates the significance of an all-encompassing understanding that extends beyond mere syntax to encompass the broader context of the problem. The shared experiences suggest a significant learning curve and illuminate the pivotal role of deep, conceptual comprehension in facilitating effective problem-solving within software development.

Likewise, as shared by participants, the journey through Python’s syntax and runtime errors also highlights a challenging yet enlightening path in programming. Initially, SDT2 opened the discussion with a reflection on their struggle, stating, “I did find myself

challenged by some of the syntax and more intricate runtime errors. For instance, the runtime error involving string-to-number conversion was a bit tricky for me initially”. This admission sets the stage for a broader conversation about the complexities involved in understanding and resolving programming errors. Subsequently, SDT4 connected with this sentiment, revealing, “we managed to fix some runtime errors; however, the infinite loop issue highlighted a gap in my skills”. Similarly, SDT8 aligned with this perspective, adding, “I also share SDT7’s sentiment about runtime errors; not catching the infinite loop was a learning point for me”, further emphasising the common challenges faced by the group.

Moreover, the conversation deepened as SDT21 shared their specific struggles, noting, “for me, the runtime errors were the most challenging during our collaboration, particularly Infinite loop due to missing colon in ‘for’ loop… runtime errors often require an understanding of the code and how the Python interpreter executes it”. This insight was supported by SDT10, who observed, “runtime errors needed us to think about the logic and structure of the program, which can be quite daunting”. Similarly, SDT11 highlighted a particular type of error, mentioning, “I found runtime errors particularly challenging, specifically the ‘not converting string input to number’ error. It was a bit perplexing as it involved understanding the data types and how Python handles input operations”.

The narrative further evolved with SDT4 and SDT5 discussing the need for a deeper analysis and the realisation of their beginner status through these errors. SDT5 candidly stated, “runtime errors such as the infinite loop. These areas, which I couldn’t fix, clearly

indicate my beginner status and lack of in-depth programming experience". SDT16, drawing from SDT15's experiences, identified additional challenges, including runtime errors such as the use of undefined variables and infinite loops caused by missing colons in 'for' loops, highlighting the difficulties posed by their limited experience.

In all, this collection of insights illuminates the technical hurdles encountered when addressing runtime errors and emphasises the valuable learning moments they provide. Through these shared experiences, the narrative captures the participants' journey of discovery and adaptation in confronting programming challenges. It highlights the essential role that a profound comprehension of Python's logic, syntax, and execution flow plays in overcoming these obstacles, marking a significant step in their developmental journey as programmers.

5.3.2 Theme 2: Technical and Cognitive Skills

The theme 'Technical and Cognitive Skills' highlights a participant's ability to employ various debugging tools and their aptitude for logical reasoning and problem-solving. As seen in Table 29, this theme is further divided into three key sub-themes: technology utilisation; debugging strategies and tactics; and cognitive load sharing. Moreover, a selection of extracts from the debugging sessions vividly exemplifies technology utilisation, debugging strategies and tactics, and cognitive load sharing as three sub-themes.

Table 29: Technical and Cognitive Subthemes in Interview Sessions

Subthemes	Description
Technology Utilisation	Participants leveraged IDE tools such as Visual Studio Live Share for real-time collaboration, syntax highlighting, debugging consoles, and version control, enhancing their abilities to debug remotely and efficiently.
Debugging Strategies & Tactics	A mix of structured methods like print statement debugging, IDE debuggers, rubber duck debugging, and divide-and-conquer approaches helped participants systematically identify and resolve coding errors.
Cognitive Load Sharing	Participants managed mental workload by adopting strategies such as the Driver-Navigator model, verbalising thought processes, role-switching, and leveraging individual strengths to maintain efficiency and prevent cognitive fatigue.

Starting with the first sub-theme, technology utilisation, participants across all fifteen dyads universally utilised IDE tools, notably Visual Studio and Microsoft Visual Studio Live Share, to navigate the challenges of distributed pair debugging in Python code. Commonalities across dyads included the use of real-time code collaboration features, particularly effective for overcoming geographical barriers and addressing errors. Despite employing similar strategies, each dyad exhibited slight divergencies in their use of technology. For instance, the exploration of debugging in remote settings, as shared by participants, reveals the invaluable role of Integrated Development Environment (IDE) tools and collaborative platforms in overcoming the challenges posed by physical distance. SDT17 observes that “despite the physical distance, the use of tools like Microsoft Studio live share facilitated real-time collaboration, making the process smoother than anticipated”. This suggests the effectiveness of live-sharing features in bridging gaps between team members. Similarly, SDT18 highlighted, “the IDE’s collaborative features, such as live code sharing and simultaneous editing, significantly eased the challenges of remote pair debugging”, pointing to the synergy between technology and teamwork.

SDT4's remark, "another aspect of the IDE that greatly aided our debugging process was the integrated version control system", alongside SDT2's detailed account - "In our session, we used Microsoft Teams and Visual Studio Live Share, which allowed us to share and edit code in real-time" illustrated the multifaceted benefits of these platforms in enhancing collaborative debugging efforts. However, SDT8 noted a potential downside, "another factor contributing to these challenges was our reliance on technology to bridge our geographical gap", suggesting that while technology is a facilitator, it also introduces dependencies.

The conversation then shifts to specific IDE features that streamline the debugging process. SDT6 and SDT10 mention a "feature of the IDE that significantly helped us was the syntax highlighting and code suggestion features", and "the IDE's features like code highlighting and error notifications were significant in streamlining our debugging process", respectively. These functionalities aid in error identification and enhance learning. Building on this, SDT16 and SDT14 reflected on the broader utility of these tools, "building on SDT15's points about syntax highlighting and auto-indentation, I found the IDE's error notifications and debugging console extremely beneficial", and "I found the features like code completion and intelligent suggestions particularly beneficial".

Further emphasising the role of IDEs in debugging, SDT1 and SDT8 discuss the use of debuggers and integrated consoles by saying "another aspect was the use of IDE debuggers. They allowed us to step through the code and inspect variables at each stage", and "moreover, the integrated console within the IDE was a boon for Print Statement

Debugging”, highlighting how these tools facilitate a deeper understanding of code execution. Echoing this sentiment, SDT14 states, “we focused more on leveraging the IDE Debugger and Print Statement Debugging... this hands-on, tool-centric approach, complemented by our continuous dialogue, made our debugging more efficient and educational”.

Concluding the insights, SDT12, SDT20, SDT30, and SDT24 collectively praised the IDE’s broader capabilities, “another aspect (of IDE) that I found incredibly helpful was the integrated console and output window”, “the IDE’s capabilities for instant feedback and error highlighting significantly boosted our debugging efficiency”, “another aspect of IDE tools that proved immensely helpful was the code analysis features”, and “the IDE’s debugging features were pivotal in our session... IDE) tools were a game-changer”, they articulated. These reflections vividly depict how IDE tools and collaborative technologies are not just facilitators but essential elements in the modern debugging process, transforming challenges into opportunities for enhanced learning and efficiency in programming.

This subtheme, Debugging Strategies and Tactics, also featured prominently in the interview session transcript. Participants from a collaborative coding session provided insightful reflections on their varied strategies, each enriching the conversation with their unique experiences. SDT6 initiated the dialogue with a nod to teamwork, stating, “we collectively decided to use Print Review Debugging for complex issues, where both of us would analyse the outputs and brainstorm potential solutions”. This collaborative spirit was mirrored by SDT2, who applied, “employed strategies like Print Statement Debugging

and Slicing more methodically”, showcasing a disciplined approach to unravelling code complexities.

Furthermore, the discussion took a turn towards the benefits of structured methods through SDT21’s revelation, “collaborating with SDT22 introduced me to the systematic use of Print Statement Debugging and IDE Debuggers... It helped me realise how structured methods could offer clearer insights into the code’s behaviour”. This structured approach was contrasted by SDT6 and SDT23’s initial reliance on a more heuristic method, with SDT6 admitting, “I leaned more towards the Trial and Error approach in our session. Additionally, I used Tinkering”, and SDT23 reflecting on their journey from trial and error to integration of print statement debugging, “I heavily relied on the Trial and Error method at first... that’s when we started integrating print statement debugging”.

Moreover, SDT7’s commentary clarified a shift from intuition to a more systematic analysis as exemplified by “we adopted a more structured approach, systematically going through the code, which is a shift from my usual more intuitive method”. This evolution towards structured analysis was further supported by SDT8 and SDT9, who each found a balance between tried-and-true methods and exploratory techniques, with SDT8 expressing a preference for print statement debugging and seeing the value in code review, “I tend to favour Print Statement Debugging as my go-to strategy... I also see the merit in Code Review”, and SDT9 combining trial-and-error with print statement debugging, “I primarily focused on the ‘Trial-and-Error’ strategy... sometimes, I used to ‘Print Statement Debug’”.

Additionally, the dialogue expanded with SDT13, SDT16, and SDT17 incorporating additional strategies such as Rubber Duck Debugging and Pattern Matching. SDT13 mentioned, “I heavily relied on Print Statement Debugging... Additionally, Rubber Duck Debugging”, while SDT16 found pattern matching and code review beneficial by stating “I also found ‘Pattern Matching’ quite useful during our session... Additionally, ‘Code Review’”. SDT17’s approach combined IDE Debuggers with the unique method of Rubber Duck Debugging by indication “I was drawn towards using IDE Debuggers and Rubber Duck Debugging”. Participants SDT19 and SDT27 highlighted the importance of tracing and tinkering, with SDT27 specifically stating, “I primarily focused on print statement debugging... Tinkering also played a significant role in my approach”, illustrating a hands-on and exploratory approach to debugging.

A divide-and-conquer strategy was mentioned by SDT5, illustrating an efficient distribution of effort, remarking “we adopted a divide-and-conquer strategy, where each of us focused on different types of errors”. This strategy was part of a broader narrative of collaboration and rhythm in debugging, as shared by SDT16 and further elaborated by SDT26, who spoke to the benefits of discussing code changes comprehensively by asserting “we started with a comprehensive code review... discussing each part of the code before making changes allowed us to understand the underlying logic better”.

The collective reflections culminate in a narrative that highlights the diverse strategies employed by participants and emphasises the evolution of debugging practices through collaboration and shared learning. From the reliance on traditional print statement

debugging to the adoption of more complex approaches like pattern matching and IDE debuggers, the participants' experiences demonstrate the dynamic and extensive nature of debugging within the coding process.

Also, during the interview sessions, participants from DYAD1 to DYAD15 shared varied strategies for effectively managing cognitive workload and distributing responsibilities, spotlighting the theme of "Cognitive Load Sharing". Through their experiences, the importance of clear communication, strategic use of tools, and the dynamic distribution of roles emerged as crucial factors in navigating the complexities of debugging tasks.

SDT2 from DYAD1 articulated the value of articulating thoughts and utilising Integrated Development Environment (IDE) features, stating, "Another method we used was verbalising our thought process... we utilised the features of our IDEs, like breakpoints and debuggers... This blend of clear communication, role swapping, and effective use of tools ensured that we managed our cognitive workload well throughout our debugging session". This approach highlighted the blend of verbalisation and technological support in mitigating cognitive strain. In DYAD2, SDT4 and SDT3 shared insights into their collaborative dynamics and mental strategies. SDT4 described their adoption of a 'Driver-Navigator' model by declaring, "We intuitively adopted a 'Driver-Navigator' model to distribute responsibilities... This division of roles allowed us to manage the cognitive workload effectively... It also meant that we could switch roles and keep the session dynamic, preventing fatigue and tunnel vision", showcasing the benefits of role flexibility and division. Adding to this, SDT3 emphasised the role of communication and breaks, declaring "Our use of the 'Think Aloud' protocol was crucial in managing our cognitive

workload... We also made sure to take short breaks to prevent cognitive overload, especially after resolving a particularly challenging error", underlining the necessity of vocalising thoughts and pacing the session to maintain cognitive health.

Similarly, SDT5 from DYAD3 highlighted a strategy tailored to individual strengths by saying that, "We intuitively adopted a strategy that distributed responsibilities based on our individual strengths and comfort zones... We also set up a system where we would alternate roles every 15 minutes... This method ensured that neither of us became too mentally fatigued", illustrating an approach focused on leveraging personal strengths and maintaining mental stamina through role rotation. In DYAD10, SDT19 and SDT20 presented a systematic method for dividing debugging tasks. SDT19 spoke of a strategic distribution of work by emphasising that, "We adopted a strategic approach to distribute responsibilities... This allowed me to focus deeply on specific sections, reducing the cognitive load", indicating a depth-focused strategy. Complementing this, SDT20 outlined their role in broader oversight by articulating that, "I focused more on 'Print Statement Debugging' and overseeing the broader logic of the program... We also scheduled regular intervals to swap roles and discuss our findings", highlighting the balance between micro-level debugging and macro-level program understanding.

These narratives collectively illuminate the significance of adaptability, clear communication, and strategic planning in debugging. By incorporating verbal protocols, technological tools, and structured role distribution, the participants demonstrate a polymorphic approach to cognitive load management, reflecting the collaborative nature of problem-solving in coding environments.

5.3.3 Theme 3: Challenges

The interview session also unveiled a variety of challenges that the participants had during the debugging session, categorised into three distinct subthemes, namely, Communication and Collaboration; Technical and Cognitive; and Environmental and Logistics. As seen in Table 30, each of these sub-themes encapsulates specific aspects of the difficulties faced, shedding light on the multifaceted nature of debugging.

Table 30: Challenges Subthemes in Interview Sessions

Subthemes	Description
Communication and Collaboration	Participants struggled with remote debugging due to challenges in conveying thoughts clearly, synchronising edits, and overcoming the absence of non-verbal cues, necessitating extra effort for clarity and coordination.
Technical & Cognitive	Debugging required managing complex errors, synchronising understanding, and handling cognitive strain, all of which were further complicated by technological limitations and geographical separation.
Environmental and Logistics	Geographical dispersion, reliance on digital tools, and unpredictable internet connectivity introduced additional challenges, making real-time collaboration and seamless communication more difficult.

The subtheme of ‘Communication and Collaboration’ was significantly emphasised by participants across various dyads as a notable challenge they encountered, particularly when remotely debugging Python code. SDT1 from DYAD1 shared, “while tools like Visual Studio Live Share helped bridge the physical distance, we had to work harder to ensure clear and precise communication... explaining our thought processes or understanding the other’s perspective took extra effort”. This sentiment spotlights the need for enhanced clarity in remote interactions, where digital tools cannot fully compensate for the absence of face-to-face communication.

Echoing this challenge, SDT26 from DYAD26 noted the complications arising from digital collaboration, stating, “while tools like Live Share were invaluable, there were moments

when simultaneous editing led to confusion... additionally, the inability to physically observe each other's non-verbal cues was a minor hurdle". The lack of non-verbal cues and the confusion caused by simultaneous edits suggest the difficulties of remote collaboration. Similarly, SDT6 from DYAD6 highlighted issues with concurrent code modifications by articulating that "when we both tried to edit or highlight the same piece of code... it occasionally led to confusion and required us to pause and clarify who was taking the lead". This points to the importance of clear role delineation in preventing misunderstandings during collaborative tasks.

SDT4 from DYAD4 discussed the impact of technical delays on collaboration, revealing, "there were moments when changes made by SDT3 took a few seconds to reflect on my screen and vice versa... this lag, although minor, disrupted the flow of our debugging process". The slight delay in synchronising edits can disrupt the debugging flow, emphasising the need for patience and understanding in remote setups. Geographical challenges were addressed by SDT5 from DYAD5, who mentioned, "being geographically dispersed meant we couldn't simply look over each other's shoulder to point out issues or discuss solutions... We had to be extra clear and concise in our verbal explanations". The physical distance necessitates a higher level of verbal clarity, highlighting the importance of effective communication in remote debugging.

Interpretative differences were a concern for SDT23 from DYAD23, who said, "there were instances where we had different interpretations of the error messages, particularly the logical errors like the tax calculation logic". This indicates the potential for varied understandings of problems and the need for a unified approach to debugging. SDT24

from DYAD24 discussed the discipline required for effective remote collaboration, stating, "it required us to be more disciplined in our approach... additionally, the limited experience we both had meant that more complex errors, such as those involving deeper logical or structural issues in the code, took us longer to resolve". The comment reflects on the need for a structured approach and the challenges posed by inexperience.

Finally, SDT25 from DYAD25 lamented the limitations of digital communication, saying, "another obstacle was the limited ability to physically point out specific code segments or errors". The inability to directly indicate issues highlights another layer of challenge in remote debugging. These insights offer a comprehensive view of the intricacies of remote collaborative debugging. Despite the benefits of digital collaboration tools, the absence of physical presence and the intricate nature of effective communication and role clarification become evident. The participants' experiences stress the necessity for clear communication, patience, and a disciplined approach to navigate the complexities of debugging collaboratively across distances.

In addressing the **Technical and Cognitive sub-theme**, apprentices grappled with the dual challenges of navigating complex programming errors and the cognitive demands these errors imposed, particularly in a remote setting. The narrative begins with SDT30's reflection on the hurdles of technical glitches, such as "when we were tackling the 'Infinite loop due to missing colon' issue... additionally, relying on technology meant we were at the mercy of our internet connections, which occasionally disrupted our flow". This candid admission focuses on the reliance on stable internet connections in remote debugging and how technical issues can hamper progress.

Echoing this sentiment, SDT6 delved into the complexities of managing a shared editing environment, stating, “another significant obstacle was managing the shared editing environment effectively... this aspect of remote collaboration demanded a high level of coordination and patience”. The necessity for enhanced coordination and patience is highlighted here, showcasing the intricate balance required in remote collaborative settings. The theme of coordination is further explored by SDT12, who mentioned, “another obstacle we faced was the limitation in real-time feedback and reaction... we found that our responses to each other’s suggestions were sometimes delayed”, pointing to the challenges of immediate communication in synchronising collaborative efforts.

The conversation shifts to the use of specific tools with SDT14’s expressing that, “Another obstacle was efficiently utilising the IDE Debugger in a remote setting... This limitation made it difficult to collaboratively explore different hypotheses about the bug”. This insight brings to light the challenges of leveraging debugging tools remotely, complicating the collaborative exploration of potential solutions.

SDT1 and SDT2 discussed the cognitive load involved in debugging, with SDT1 stating, “One of the main obstacles we encountered during our debugging session was dealing with complex logical errors... these errors required a deep understanding of Python’s logic and syntax, which was challenging given our limited experience”. This is complemented by SDT2’s observation, “another significant obstacle was maintaining a synchronised understanding of the code... managing the cognitive load was also a challenge”,

highlighting the cognitive strain in maintaining mutual comprehension of the code amidst these technical challenges.

The geographical divide adds another layer of complexity, with SDT23 and SDT27 noting the difficulties it introduced. SDT23 mentioned, “the geographical separation added another layer of complexity”, while SDT27 expanded on this, saying “one of the significant obstacles we faced was the time lag and communication barriers due to our geographic separation... The lack of immediate, direct interaction made it more challenging to collaboratively and swiftly navigate through these complex issues”. These reflections accent the compounded difficulties of geographic separation, including time lags and communication barriers that hinder swift, collaborative navigation through technical issues.

SDT28’s comment, “another obstacle was effectively managing and integrating our different approaches to debugging... balancing these approaches remotely required careful coordination and patience”, concludes the discussion, focusing on the challenge of integrating diverse debugging approaches. This summary encapsulates the apprentices’ experiences, highlighting the multifaceted nature of technical and cognitive challenges in remote debugging, where technical difficulties, cognitive demands, geographical separation, and the need for coordination converge, illustrating the complexities of collaborative problem-solving in programming.

Similarly, in exploring the ‘Environmental and Logistics Challenges’ encountered during collaborative Python debugging, participants vividly described the complexities

introduced by geographical dispersion and reliance on digital tools. SDT2 opens the narrative, emphasising the isolation felt in remote settings by asserting that, “in a remote setting, it’s easy to feel isolated with the problem at hand”. This sentiment sets the stage for a series of reflections on the limitations of remote collaboration.

SDT3 and SDT6 discuss the physical limitations of digital communication, noting the inability to use gestures or point at the screen, noting “being geographically dispersed, we couldn’t simply point at the screen or use physical gestures to express our ideas”, and the challenges even helpful tools like Visual Studio Live Share introduce, “while tools like Visual Studio Live Share were immensely helpful, they also presented challenges”. These insights highlight how digital tools, despite their benefits, fall short of replicating the intricacies of in-person interaction.

The narrative then shifts to the constraints of tool usage, with SDT14 and SDT24 expressing the limitations on control during debugging and the gap left by the absence of physical presence by reiterating that, “only one of us could actively control and navigate the debugger at any given time”, and “real-time collaboration tools are great, but they can’t completely bridge the gap created by not being physically present in the same room”. These comments identify the challenges in achieving seamless collaboration remotely.

SDT19, along with other participants, touches on the time and effort required to communicate and understand concepts across distances by mentioning that, “our geographical separation... as it limited our ability to quickly bounce ideas off each other

and required more time to explain and understand concepts". This observation was echoed in remarks about the added complexity and the need for over-communication and reiterating that, "the lack of physical presence meant we had to over-communicate to compensate for the lack of in-person interaction, which sometimes slowed down our debugging process" (SDT26), reflecting the intricate balance required to maintain effective communication and collaboration remotely.

Concluding the discussion, SDT30 brings attention to the environmental challenges of working in different settings and the unpredictability of internet connectivity, expressing that, "our different locations also meant we were working in different environments, which sometimes led to distractions or interruptions unique to our individual settings... relying on technology meant we were at the mercy of our internet connections, which occasionally disrupted our flow". This summary encapsulates the multifaceted Environmental and Logistics Challenges faced by apprentices in debugging Python code collaboratively across distances, underscoring the critical role of effective communication, the limitations of digital tools, and the personal adaptability required in remote work environments.

5.4 Focus Group Discussion Findings

The focus group discussion was undertaken with work-based mentors and trainers on the debugging practices of SDT apprentices. These mentors and trainers, who work closely with the apprentices, provided crucial perspectives to bolster the data already collected from the apprentices, thereby enriching the overall understanding of the study's objectives. The discussion was structured around seven key questions, each designed to

delve into different facets of the research areas, initially focusing on the mentors' observations that apprentices often start with casual reasoning and frequently rely on trial-and-error methods when tackling debugging tasks.

Notably, it is observed that as apprentices gain experience and confidence, they tend to gradually move towards more structured methods, such as the top-down approach. The mentors unanimously acknowledged the effectiveness of strategies such as pair programming, pattern matching, and IDE debuggers, noting their complexity and their long-term benefits. They emphasised the value of mentoring, particularly through code reviews and collaborative problem-solving, in enhancing apprentices' debugging skills. Similarly, they opine that a shift towards proactive strategies, moving from basic techniques like print statement debugging to more advanced methods such as static code analysis and rubber duck debugging, were crucial for a deeper understanding of the code. As the discussion progressed, the mentors explored factors influencing apprentices' choice of debugging strategies, including educational background, project complexity, tool familiarity, learning environment, and peer influence. The variability in strategy effectiveness was noted, depending on the nature of bugs, apprentice skill level, and project context, underscoring the mentor's vital role in guiding apprentices towards effective debugging techniques. This guidance is essential for equipping apprentices to tackle a wide range of technical challenges, ensuring their growth and proficiency in debugging practices.

As seen in Table 31, three principal themes were identified from the focus group discussion transcript, including the nature and management of debugging errors, the

influence of technology on debugging processes, and the strategies and challenges encountered in debugging. These themes provide insight into how apprentices approach debugging, adapt to technological tools, and navigate common challenges, shaping their overall learning experience.

Table 31: Overview of key themes in Focus Group Sessions

Themes	Description
Theme 1: Nature and Handling of Debugging Errors	Participants view debugging as an evolving process, where apprentices initially rely on casual reasoning and trial-and-error methods, gradually shifting towards systematic approaches, collaboration, and tool adoption, overcoming initial apprehension to refine their debugging strategies.
Theme 2: Technology's Role in Debugging Processes.	Participants view technology as a crucial yet challenging aspect of debugging, where initial struggles with advanced tools give way to deeper understanding through mentorship, adaptive learning, and balancing basic and advanced techniques, ultimately enhancing debugging efficiency.
Theme 3: Strategies and Challenges in Debugging	Participants recognise debugging as a progressive learning process, where initial casual reasoning and trial-and-error approaches evolve into structured problem-solving, collaboration, and pattern recognition, though challenges such as cognitive overload and code tracing difficulties persist.

5.4.1 Theme 1: Nature and Handling of Debugging Errors

The Nature and Handling of Debugging Errors theme encompasses the apprentices' approaches and attitudes towards identifying and rectifying bugs in programming code. Analysis of the WMT focus group discussions suggested that, initially, apprentices lean on "casual reasoning", as WMT7 insightfully notes, "one common observation is that apprentices often rely on casual reasoning at the beginning". This method, while a natural starting point, as further described by WMT1 as "apprentices often use their basic understanding of the code to guess where the bug might be", marks the hit-or-miss nature of early debugging attempts. The narrative evolves, with WMT7 adding, "they use their initial understanding of the code to hypothesise about potential bugs", illustrating

the apprentices' reliance on their foundational knowledge yet pointing towards the need for refinement.

Furthermore, the progression to more systematic approaches marks a pivotal development in the apprentices' debugging journey. WMT3 observed, "this trial-and-error strategy, while common, can be inefficient", heralding the shift towards the "top-down approach" which, despite its promise, presents challenges highlighted by WMT4, "But apprentices sometimes struggle to identify the right level to start breaking down the problem". The role of collaboration in skill enhancement was illuminated by WMT5, who suggested, "pairing apprentices with more experienced colleagues for code reviews can significantly enhance their ability to dissect problems more effectively", a sentiment echoed by WMT4 through advocating for problem isolation techniques.

Similarly, the narrative further explored the initial apprehension towards IDE debuggers, with WMT1 revealing, "apprentices initially find IDE debuggers intimidating", a sentiment shared by many novices. Yet, as WMT10 points out, "many apprentices are hesitant to use IDE debuggers initially"; the journey includes overcoming such fears to embrace effective debugging tools. The importance of adaptability and a tailored approach to debugging is emphasised by WMT9 and WMT10, illustrating that successful debugging strategies are contingent upon the bug's nature and the project's context.

5.4.2 Theme 2: Technology's Role in Debugging Processes

In exploring the Technology's Role in Debugging Processes theme, the WMT findings elucidate the journey of apprentices as they navigated through the complexities of

debugging tools and methodologies. WMT10 articulated the initial struggle many apprentices face with advanced debugging techniques, particularly IDE debuggers, noting, “many apprentices struggle with more advanced debugging techniques initially”. This challenge, however, is part of a crucial learning curve that, once surmounted, offers significant benefits, as highlighted by WMT11 who observed, “there’s a definite learning curve with debugging tools. However, apprentices who embrace these tools, especially pair programming, tend to develop a deeper understanding of the debugging process”. WMT2 reinforces this sentiment, pointing out the eventual appreciation for the efficiency of tools like IDE debuggers after overcoming the initial intimidation.

The conversation shifted towards the importance of balancing technology with basic techniques, where WMT6 encouraged a progression from print statement debugging to utilising IDE debuggers and breakpoints, suggesting a move towards more advanced, yet effective, debugging practices. This balance is influenced by various factors, including the specific programming language or technology stack and the type of feedback provided by the development environment, as mentioned by WMT2 and WMT3, who specify how certain environments can nudge apprentices towards particular strategies.

The discussions also explored how apprentices’ familiarities with tools shape their debugging approach. WMT3 and WMT11 discussed the impact of comfort levels with IDEs and other tools on strategy choice, emphasising the role of mentorship and peer influence in this learning process. WMT12 and WMT11 further explored the challenges apprentices face, such as the pressure to quickly fix bugs leading to rushed learning, and

the overlooked importance of replicating bugs before attempting to fix them, which is crucial for a thorough debugging process.

Through these insights, the WMT findings paint a comprehensive picture of apprentices' evolving relationship with debugging technologies. From initial hesitance to a more confident and effective use of advanced tools, the journey is marked by learning curves, mentor guidance, and the adaptive choice of strategies based on the bug's nature and project context. This account emphasises the difficulties apprentices encounter and highlights the significant impact of technological proficiency and mentorship in advancing debugging skills.

5.4.3 Theme 3: Strategies and Challenges in Debugging

The **Strategies and Challenges in Debugging** theme considered the various strategies apprentices employed in debugging and the challenges they encountered. The narrative begins with an observation by WMT1, who noted, "apprentices often start with casual reasoning when debugging. They try to make sense of the code based on their understanding". This initial strategy, however, quickly transitions as described by WMT2 by stating that, "they tend to shift quickly to a trial-and-error approach when casual reasoning doesn't yield immediate results". Despite this shift, a more analytical strategy is recognised by WMT3, who mentioned, "a few apprentices use the top-down approach effectively, breaking the problem into smaller, more manageable parts".

The importance of mentorship and collaboration in fostering debugging skills was pinpointed by WMT5, stating, "pairing them with more experienced colleagues for code

reviews can significantly enhance their ability to dissect problems more effectively". This collaborative approach was further elaborated by WMT2 through the observation that "the familiarity of print statements makes them a go-to strategy", indicating a preference for simple, tried-and-tested methods. WMT5 added depth to this discussion by highlighting a developmental milestone, explaining that, "Once apprentices are comfortable with isolating problems, they begin to develop a knack for tracing and gathering information". The narrative then delves into the cognitive strategies involved in debugging, with WMT3 observing, "when apprentices explain their thought process, whether through rubber duck debugging or to a peer, it often leads them to a solution more quickly". This articulation, as suggested, aids in problem-solving. The sentiment is echoed in the context of collaborative learning by WMT7, who noted, "apprentices who participate in code reviews develop a better eye for spotting bugs".

WMT4 brought attention to the pattern recognition strategy, asserting that "I've seen apprentices use pattern matching, especially when they encounter similar bugs they've dealt with before", which is indicative of learning from past experiences. This approach is strengthened by an additional observation from WMT5, who opined, "Once apprentices grasp isolating and slicing techniques, they begin to develop better strategies for tracing and gathering information", suggesting a progression in skillset. The discussion transitioned to the challenges faced by apprentices, with WMT12 mentioning, "apprentices also face challenges with tracing the execution of code", pointing out the difficulties in understanding code flow. This is complemented by insights into the inefficiencies of certain approaches and the benefits of structured problem-solving, as WMT3 stated, "this trial-and-error strategy, while common, can be inefficient. I've seen

a few apprentices use the top-down approach effectively, breaking the problem into smaller, more manageable parts”.

Furthermore, WMT4 highlighted a critical learning curve, noting that “the top-down approach indeed helps in maintaining a structured way of debugging. But I must mention that apprentices sometimes struggle to identify the right level to start breaking down the problem, which can be due to a lack of experience”. This sentiment is reinforced by the discussion on the evolution of debugging approaches through collaborative efforts, as noted by WMT2 and the articulation of thought processes leading to quicker resolutions, as stated by WMT3.

However, challenges such as cognitive overload, the need for a holistic understanding of the application, and the importance of abstract thinking are also addressed. WMT8 shared, “additionally, I’ve seen apprentices struggle with isolating the problem. They often fixate on a certain part of the code without considering the entire system, which can lead to missed bugs”. This was further elaborated by WMT7, who discussed the implications of fixing bugs without understanding their broader impact.

The insights culminated in the acknowledgment of the wide range of strategies employed by apprentices, from simple print statements to sophisticated pattern matching and static code analysis, as highlighted by various WMTs. This account, enriched with direct quotes and participant details, encapsulates the essence of the “Strategies and Challenges in Debugging” theme, offering an overview of the apprentices’ journey through debugging, stressed by the invaluable role of WMTs in guiding and shaping their learning experience.

5.5 Summary

This study has explored the debugging practices among software development technician apprentices, synthesising findings from practical debugging sessions, analysed Python codes, dyad interviews, and insights from Workbased Mentors and Trainers (WMTs). It provided a detailed understanding of the apprentices' experiences in debugging, highlighting their skill development and strategic progression. The study's initial phase revealed a spectrum of errors faced by apprentices, who demonstrated proficiency in resolving straightforward issues such as syntax and runtime errors yet grappled with more intricate logical errors. These challenges were highlighted in dyad interviews, which confirmed the initial findings and provided a detailed perspective on the apprentices' struggles and coping strategies.

Central to the investigation was the essential role of technology in facilitating debugging practices. Apprentices predominantly utilised IDEs and debuggers, which were instrumental in enhancing their debugging proficiency. Furthermore, collaborative platforms like Microsoft Teams and Visual Studio Code's Live Share enabled real-time collaboration and code sharing. However, navigating these technologies presented notable challenges, particularly with difficulties in balancing the mental demands of complex debugging tasks and ensuring effective communication in remote environments. Consequently, apprentices often engaged in pair programming and debugging, which proved instrumental in sharing cognitive responsibilities and fostering a collaborative approach to problem-solving.

The study observed a significant evolution in the debugging strategies employed by apprentices. Initially, they relied on more straightforward methods such as print statement debugging, tinkering, and trial-and-error. However, as they gained experience, they transitioned to more sophisticated techniques, including systematic bug isolation strategies like tracing, pattern matching, and methodical step-by-step execution within IDEs. This progression from rudimentary to advanced methods reinforced their growing proficiency and adaptability in debugging.

The research findings on challenges faced by apprentices predominantly stemmed from the debugging sessions and dyad interviews, further corroborated by insights from WMTs. Key challenges included navigating intricate codebases, deciphering misleading error messages, and tackling the inherent difficulties of remote debugging, such as latency issues and reliance on digital communication. These challenges were exacerbated by the apprentices' initial lack of experience, often leading to cognitive overload. The WMTs echoed these sentiments, underscoring the necessity for ongoing mentorship and a nurturing learning environment. Such support is crucial for addressing the technical aspects of debugging and assisting apprentices in adapting to the multifaceted challenges of software development. This approach ultimately aimed to enhance their problem-solving skills and collaborative competencies, preparing them for the complexities of professional software development.

In conclusion, the final table (Table 30) consolidates the overarching themes identified throughout Chapter 5. It offers a comprehensive view of the study's findings, tying together the dyads' practical debugging experiences, interview reflections, and focus

group discussions with mentors and trainers. Table 32 serves as a concluding reference, to easily navigate and recall the key elements of the chapter.

Table 32: Overarching themes across the study

Data Collection Method	Themes
5.1 Dyads Debugging Session Findings	The debugging session findings are summarised, highlighting the key emerging themes, such as technology's pivotal role and the dyads' diverse strategies. They are: 1) Technology Utilisation, 2) Debugging Strategies, 3) Error Spectrum, 4) Cognitive Load Management, and 5) Challenges Faced.
5.2 Python Code Analysis Findings	Error Types, Proficiency Levels, Specific Challenges, Technological Tools Used.
5.3 Interview Session Findings	1) Error Spectrum, 2) Technical and Cognitive Skills, and 3) Challenges in Collaboration.
5.4 Focus Group Discussion Findings	1) Nature and Handling of Debugging Errors, 2) Technology's Role in Debugging, 3) Strategies and Challenges in Debugging.

Chapter 6: Conclusion

6.0 Introduction

This final chapter synthesises key findings to answer the initial research questions, integrating these results into a cohesive narrative. This involves providing answers to the central questions, linking empirical data to theoretical frameworks and assessing the research's trustworthiness based on credibility, transferability, dependability, and confirmability. The chapter also discusses the study's potential impact on software development education, highlighting implications for educators and practitioners and acknowledging its limitations. It proposes future research directions to address these gaps and clarifies the study's contributions to academia and professional practice.

6.1 Evaluation of Dyad's Case Studies

The case studies of DYADs 1 to 15 offer an insightful examination into the world of novice programmers aged 16 to 50, hailing from diverse organisational backgrounds and engaging in the complex task of debugging Python scripts in a remote environment. This study illuminates the varied strategic approaches adopted by each dyad and reveals commonalities in their experiences and methodologies. As a whole, these observations provide a diverse perspective on the challenges and triumphs encountered in software development, particularly in the context of novice programmers.

Furthermore, the dyad case studies provide crucial insights into the debugging approaches of novice programmers. These novices, despite being early in their coding

journey, demonstrated adaptability and a willingness to experiment with various debugging techniques, such as print statement debugging, tinkering, trial and error, IDE debuggers, rubber duck debugging, tracing, slicing, code reviews and pattern matching. These diverse approaches align with the concepts of adaptive expertise (Bransford et al., 2000; Clarke et al., 2023; Hatano & Inagaki, 1986), self-regulated learning (Kumar et al., 2005; Ramírez Echeverry et al., 2018) and socially shared regulation of learning (Silva, 2020) in programming education. As outlined by Bransford and Schwartz (1999) and Zimmerman (2002), the principles stress the importance of applying knowledge flexibly and self-tailoring strategies for enhanced learning and problem-solving. Thus, this range of strategies adopted by novice programmers indicates a growing understanding of the complex nature of programming and debugging and an engagement with deeper learning processes, which are key traits for successful programmers essential in the dynamic field of software development.

Also, despite the hurdles of geographical separation, the dyads showcased effective remote collaboration, leveraging tools like Microsoft Teams and Visual Studio Live Share. This proficiency in collaboration, situated within the framework of distributed cognition, affirms the importance of shared cognitive responsibilities and collective problem-solving, echoing research on computer-supported collaborative learning by Salomon (1997) and Stahl et al. (2006). Thus, the ability to collaborate effectively, irrespective of physical distance, is particularly relevant in the current global landscape of software development, where teams are often dispersed across various locations.

Moreover, a prevalent challenge identified across the dyads was their struggle with complex logical and runtime errors despite their effectiveness in resolving syntax errors. This difficulty highlights a common barrier among novice programmers in grasping the more intricate aspects of programming logic and computational thinking, a concept central to Papert (1980) and Wing (2006) research. In addition, troubleshooting these advanced errors is critical in developing comprehensive programming expertise and is often a distinguishing factor between novice and experts (Alqadi & Maletic, 2017; Rigby et al., 2020; Yen et al., 2012).

Within the array of individual dyads, some pairs notably distinguished themselves through their distinctive approaches to problem-solving. DYAD1 and DYAD11, for example, displayed considerable skill in addressing syntax errors, but they also encountered obstacles when dealing with logical and runtime errors. Their proficiency in employing distributed cognition and collaborative tools showcased their teamwork capabilities and aligned with the recognised values of teamwork in software development, as emphasised in the works of Salomon (1997) and Johnson and Johnson (1987). In contrast, DYAD3 and DYAD12 adopted an approach that was more exploratory and hands-on, reflecting their developing problem-solving skills. This approach is aligned with Kolb (1984) experiential learning theory, highlighting the importance of active engagement in the learning process.

In addition, the strategies adopted by DYAD7 and DYAD13, which included rubber duck debugging along with more traditional methods, illustrated their innovative approach to problem-solving. This technique aids in externalising thought processes, a crucial

component of metacognition in learning, as Flavell (1979) discussed. Finally, the strategy of DYAD15 stood out due to their effective utilisation of various debugging techniques and a well-balanced distribution of cognitive load. This approach showcased the high level of collaboration and communication skills indispensable for modern software development teams, as Torgeir et al. (2012) highlighted.

In conclusion, the cases emphasised the importance of a multi-dimensional approach to debugging in software development, highlighting the varied problem-solving strategies employed by novice programmers. Their experiences illuminated the challenges inherent in addressing complex logical and runtime errors, pointing to areas for further learning and skill development. Despite the constraints of remote interaction, the effective collaboration observed across these studies emphasised the pivotal role of communication and teamwork in programming, resonating with contemporary perspectives on collaborative software engineering (Torgeir et al., 2012). These insights contribute to our understanding of novice programmers' learning journeys and offer lessons for software development educators and practitioners.

6.2 Research Questions

This research addresses the key question, "How do the paired Software Development Apprentices in geographically distributed locations work collaboratively to fix Python programming bugs using the technology-mediated medium?" In pursuit of answers, this research has explored five distinct yet interrelated research questions.

RQ1: Types of Bugs

What bugs are generated by the paired geographically distributed SDT apprentices working collaboratively to solve a given problem using Python?

In exploring the errors encountered by SDT apprentices while debugging Python code, a detailed analysis reveals three distinct types of errors namely, syntax, logical, and runtime, all of which are faced during collaborative problem-solving efforts. This outcome reinforced previous studies indicating that these are typical bugs encountered by those new to programming in Python, acknowledging that, as an interpreted language, Python is prone to both compile-time and runtime errors (Becker et al., 2019; Cherenkova et al., 2014; Helminen et al., 2013; Pritchard, 2015). This study, encompassing 20 pre-seeded errors across these categories, provided insight into the varying proficiency levels among 30 apprentices grouped into fifteen dyad teams. Initiating the analysis with syntax errors, it is widely acknowledged that these constitute the most basic and easily identifiable errors in programming (Ahadi et al., 2018), a perspective robustly supported by insights from SDT12 of DYAD6 and SDT15 of DYAD8. These participants notably emphasised that syntax errors were prevalent and relatively straightforward to diagnose and correct. This view aligns with Sebesta (2016) assertion that syntax errors, while elementary, are critical in gauging a programmer's understanding of a language's framework.

Furthermore, the study's scope revealed a diverse proficiency landscape among the dyads in this domain. Particularly challenging were complex loop structures (Lowe, 2019), including notably incomplete 'for' loops (Kohn, 2019; Luxton-Reilly & Petersen, 2017), and issues in conditional statements, such as 'else' used without a preceding 'if' (Lutz,

2013). The prevalence of these errors in specific dyads, notably DYADs 6, 1, 11, 12, and 15, indicates a fundamental gap in understanding Python's essential structure and flow control (Lowe, 2019). This revelation is critical, highlighting a significant divergence in the apprentices' skill levels and conceptual grasp.

Conversely, many dyads demonstrated proficiency in basic syntax, adeptly addressing errors like missing colons and incorrect operators. This disparity in skill levels is particularly revealing, suggesting that while some apprentices comfortably navigate Python's basic syntax, others face considerable challenges with more complex constructs.

This could result from three types of breakdowns due to a programmer's cognitive limitations in conjunction with the programming system or external environment, according to Ko and Myers (2005). Such a scenario calls attention to the imperative need for a balanced and comprehensive approach to syntax education within programming curricula. As Downey (2012) aptly notes that establishing a solid foundational knowledge of programming languages is essential for developing proficiency. This approach ensures that learners are equipped to handle basic syntax and are prepared to tackle more advanced and intricate programming challenges.

Moreover, Gomes and Mendes (2007) reinforce the importance of addressing these disparities in educational settings, advocating for tailored teaching strategies that cater to diverse learning needs. By adopting such strategies, educators can ensure that all apprentices, regardless of their initial proficiency levels, can comprehensively understand programming syntax (Sun et al., 2024). Taking this into account, the study illustrates the varying degrees of proficiency in syntax among apprentices, highlighting the need for

educational approaches that accommodate this diversity. Through a combination of foundational teaching and tailored strategies, it is possible to bridge the gaps in understanding and skill, ensuring a more uniform and thorough comprehension of programming languages among learners.

Transitioning to logical errors presented a spectrum of challenges and competencies among the dyads. DYADs 3 and 6, for instance, encountered substantial difficulties with complex logical issues, exemplified by incorrect tax calculation logic. This struggle with logical problem-solving extended beyond these groups, as evidenced by the challenges faced by DYAD14. Such instances revealed that logical errors in programming were not merely syntactical mishaps but often involved more profound conceptual misunderstandings (Alqadi & Maletic, 2017; Rigby et al., 2020). In addition, SDT21's experiences with the special variable 'name' in Python further illustrated this complexity in logical errors, as the error encountered was not just a syntactic oversight but a misapprehension of the variable's conceptual usage (Pea, 1986).

Similarly, SDT19's reflections on the improper use of 'name' highlight the requirement for a more comprehensive understanding of the interactions between various code segments. These insights align with findings by Miller et al. (2019) regarding the intricacies of variable usage in Python. Adding to the complexity, SDT7 speaks of frustrations experienced while handling logical errors like bonus calculations, which lacked contextual clarity, pointing to a need for more contextually rich problem-solving scenarios in programming education. Contrasting these struggles, DYADs 1, 4, 7, and 15 displayed a firm grasp of programming logic, adeptly resolving most identified logical

errors. Notably, DYAD10's exceptional proficiency in resolving complex logical challenges aligned with the observations made by Ettles et al. (2018), reinforcing the notion that such skills can be developed with appropriate training and practice. This variance in proficiency amplifies the necessity for programming education to cater to diverse levels of logical understanding. As Tan (2021) suggests, educational curricula should reinforce fundamental concepts for novices and present complex problem-solving scenarios to challenge more advanced learners.

Furthermore, the findings suggest that an emphasis on practical application, as advocated by Hazzan et al. (2020), could enhance learners' abilities to tackle logical problems effectively. By integrating real-world scenarios, educators can provide learners with the context necessary to understand and solve complex logical errors (Robins et al., 2003). Additionally, the need for differentiated instruction, as highlighted by Tomlinson and Imbeau (2023), becomes evident in addressing the varied proficiency levels observed.

On another note, examining runtime errors in Python programming, as experienced by several dyads, reveals an intricate landscape of challenges marked by a common struggle with infinite loops. Often attributed to syntax issues like missing colons in 'for' loops (Kohn & Manaris, 2020; Simon et al., 2007), these errors underscore a broader difficulty in comprehending loop mechanics, an essential aspect of programming (Sedgewick & Wayne, 2016). This persistent challenge indicates a more profound issue than mere syntactic oversight, suggesting a fundamental gap in understanding crucial programming concepts. In stark contrast to the difficulties with loops, most dyads demonstrated relative ease in handling basic data type operations, such as converting string inputs to

numbers. This disparity in handling different types of runtime errors illuminates a variation in understanding complex programming structures compared to simpler operations. This observation is aligned with the findings of Monat et al. (2020) and Fromherz et al. (2018), who emphasise the importance of a strong foundation in fundamental concepts like variable scope and declaration.

In conducting further investigations through interviews, it became apparent that apprentices often faced substantial challenges in addressing runtime errors, especially in remote collaboration. Participants, including SDT2 and SDT4, explored specific runtime errors, highlighting the need for a deep understanding of Python's logic and its intricate aspects (Winslow, 1996). Furthermore, these observations resonate with the findings of Soloway and Spohrer (1989), who pointed out the deficiencies in novices' understanding of various programming language constructs such as variables, loops, arrays, and recursion. Likewise, SDT8 and SDT30 highlighted the complexities in resolving infinite loop errors, emphasising the crucial necessity for precise detection and rectification.

These perspectives resonate with the cognitive and technical demands highlighted by SDT10 and SDT11, who accentuated the significance of conducting a thorough analysis of a program's structure and logic, particularly for those less experienced. Such an approach is supported by Wing (2006) argument on the importance of computational thinking in programming. In these contexts, the value of debugging tools was also mentioned by SDT16 and SDT29, who pinpointed the essential roles of debugging consoles and variable state inspection in tackling complex issues. The necessity of these tools in debugging is

reinforced by the work of Murphy et al. (2008), who examined the role of debugging in software development.

Furthermore, this narrative brings to the forefront the intricate nature of runtime errors (Zhang et al., 2023) in Python programming. The insights offered by the participants illuminate the vital roles of debugging tools and the cognitive and technical skills required for effectively managing these errors. This observation is in line with Papert (1980) theory of constructionism, which advocates for hands-on experience in learning complex concepts. Thus, the identified challenges unveil gaps in skills and present considerable opportunities for learning and development. Similarly, novices SDT5 and SDT19, in their struggles with runtime errors, highlighted areas needing further development and understanding, supporting Vygotsky and Cole (1978) theory of the Zone of Proximal Development in learning. The array of issues encountered, exemplified by SDT2's difficulties with string-to-number conversion and SDT4's acknowledgement of a skill gap manifested by an infinite loop issue, reveals the breadth of challenges. Moreover, SDT8 and SDT30's reflections on specific runtime errors, like the 'Infinite loop due to missing colon', and the importance of understanding Python's interpreter execution, further mark the complexities involved (Guzdial & Ericson, 2013).

In addition, the role of debugging tools, as emphasised by SDT16 and SDT29, along with the cognitive demands of grasping a program's logic and structure, as noted by SDT10 and SDT11, are critical for effectively managing runtime errors. These insights are in harmony with the findings of Pea (1986) on the cognitive technologies for learning programming. SDT4's and SDT12's comments on the importance of critically evaluating

the code's execution flow and variable scope further reinforce these points, aligning with the assertions of Soloway and Ehrlich (1984) on the mental models in programming. Similarly, the experiences of the apprentices dealing with runtime errors in Python programming reveal a challenging landscape. These challenges, indicative of skill gaps, offer substantial learning opportunities, emphasising the need for a comprehensive approach to programming education. This approach should encompass both basic syntax and the deeper intricacies of Python's structure and logic, as advocated in the pedagogical theory of Bruner (2009).

Summing up, analysing the dyads' performances concerning syntax, logical, and runtime errors, (Kohn, 2019) uncovers trends and educational implications that are significant in programming education. The variance observed in skill levels highlights that while basic syntax (So & Kim, 2018) is generally well-understood among apprentices, there remains a pronounced need for more focused education on complex syntactical structures. This need is further exemplified in handling logical errors, where a broader range of competency is evident. Some dyads demonstrated a strong grasp of programming logic, while others faced considerable challenges (Smith & Rixner, 2019). This variability accentuates the importance of personalised learning paths, particularly in logical problem-solving. Also, the apprentices faced a more consistent set of challenges regarding runtime errors, particularly in areas such as loop control and variable scope. This uniformity in struggling with specific runtime errors across different dyads suggests fundamental gaps in programming education that need to be addressed (Smith & Rixner, 2019). However, it is noteworthy that areas involving basic operations, like data type conversion, were generally handled with greater ease. This observation indicates a

relative comfort among apprentices with Python's fundamental concepts, a foundational aspect of programming literacy.

RQ2 – Debugging Strategies and Tactics

What bug locating strategies and tactics are deployed by the paired geographically distributed SDT apprentices while attempting to fix defects in the given Python code? How do they go about finding the bugs in the program code?

Upon scrutiny of the debugging sessions of the dyads, a richly varied mosaic of debugging strategies and tactics was uncovered. This variety, captured through the lens of the Distributed Pair Debugging Conceptual Framework (DisConFrame) - discussed in Chapter 3 - highlighted the complexities of the debugging process. Within this framework, the think-aloud verbal protocol emerged as a crucial element, enhancing comprehension of how individuals and pairs navigated through the complex realm of debugging. When viewed through the lens of this framework, dyads' approaches to debugging in distributed environments became somewhat more explicit. Their journey through the Python code, pursuing deliberately embedded bugs, resembled 'a strategic foray into a labyrinthine forest in search of elusive prey'. As detailed in Chapter 5, Table 5.1 illustrates this diversity, showcasing nine distinct and multifaceted debugging strategies and tactics.

A close examination of dyads' debugging strategies and tactics revealed that print statement debugging was widely adopted, with 14 out of 15 dyads using it. This affirms the ongoing significance of print statement debugging in their debugging processes (Alqadi & Maletic, 2017; Liu & Paquette, 2023). As noted by DYAD1 and DYAD6, print

statements offer real-time insights into code behaviour, in line with research by Fitzpatrick and Collins-Sussman (2015) and Spinellis (2016). Additionally, DYAD2 highlighted the benefits of print statement debugging for immediate feedback and error recognition, supported by studies conducted by Layman et al. (2013) and Li et al. (2018). However, the simplicity of print statement debugging can also be its limitation (Agrawal et al., 1993; Fitzgerald et al., 2008; Poole, 2005; Zeller & Hildebrandt, 2002), as it may not be effective for more complex debugging scenarios, where the intricacies of code behaviour require deeper analysis (David, 2002; Matloff & Salzman, 2008).

Additionally, DYAD3, DYAD10, DYAD11, DYAD14, and DYAD15 demonstrated a significant preference for tinkering, a hands-on, exploratory method that involves interactive experimentation with code, facilitating learning and problem-solving through direct, experiential engagement with programming (Murphy et al., 2008). This approach, notable in the dyad case studies, allowed programmers to modify and examine their code gradually, enhancing their comprehension of its effects (Beckwith et al., 2006; Vossoughi & Bevan, 2014). However, contrasting viewpoints from Liu et al. (2017) and Murphy et al. (2008) suggest that tinkering might restrict the development of a more profound understanding of the program and is always ineffective (Park et al., 2015). It has also proven effective in various dyads, notably correcting syntax errors.

SDT9 also found tinkering valuable for syntax errors, aligning with Vossoughi and Bevan (2014) findings on its benefits for basic error correction. Similarly, SDT6 and SDT7 demonstrated their practicality in understanding and testing code, resonating with Beckwith et al. (2006), who highlight the importance of direct code engagement.

Likewise, reflections from DYAD11, DYAD14, and DYAD15 members further affirmed the significance of tinkering in the debugging process by incrementally modifying the code and observing outcomes, enhancing their understanding of code functionality. However, Murphy et al. (2008) caution that tinkering might not suffice for complex errors, potentially limiting deeper skill development.

In the same vein, the 'Trial and Error' method, characterised by its experimental and hands-on approach (Gugerty & Olson, 1986), was notably used by one in three dyads, DYAD3, DYAD5, DYAD6, DYAD12, and DYAD13. However, the relevance of this method's prevalence was evident in direct quotes from debugging sessions and interviews, reflecting a commitment to discovery and resilience. SDT6, SDT7, and SDT9 illustrated its exploratory and hands-on nature. SDT6's approach reflected Kolb's Experiential Learning Theory (1984), emphasising learning through experience. SDT7's method aligned with Piaget's Constructivist Learning Theory (1954), underscoring learning through direct interaction with the code. Similarly, SDT9's focus on trial and error, providing immediate feedback, resonated with Vygotsky's Social Development Theory (1978), highlighting the role of social interaction in cognitive development. Overall, trial and error, essential for immediate problem-solving, is crucial for the cognitive development of novice programmers, supported by various established learning theories, underscoring its value in programming education.

Moreover, SDT30 found print statement debugging and tinkering effective, reflecting a strategic debugging approach involving hypothesis testing and observation. This method fosters deeper code engagement and intuitive understanding within the programming

environment. The integration of various debugging techniques, as seen in DYAD2, DYAD3, DYAD14, and DYAD15, highlighted the importance of diverse strategies in addressing the complex challenges of programming, combining immediate visual feedback with hands-on experimentation and collaborative review for a deeper, more collaborative learning experience (Winslow, 1996). Also, dyads engaged in iterative testing and code modification, demonstrating their proactive approach to problem-solving. SDT6, for instance, emphasised experimenting with various solutions, a sentiment echoed by SDT9 and extended by SDT12, who also valued 'Code Review'. SDT23 also highlighted the intuitive nature of this method.

On the other hand, using IDE debuggers represents a more sophisticated approach. IDE debuggers allow for a more interactive and detailed examination of the program's execution, offering capabilities such as breakpoints and variable inspections (LaToza & Myers, 2010). This method aligns with the evolving complexity of programming tasks and the need for more advanced debugging tools. In addition to traditional methods, Integrated Development Environment (IDE) features play a significant role in enhancing debugging efficiency and productivity, as observed in the experiences of DYAD4 and DYAD9 (Proksch et al., 2018). SDT2 acknowledged the holistic view provided by IDE Debuggers, aligning with Afzal and Goues (2018) findings on the comprehensive understanding facilitated by IDEs.

Similarly, SDT14 termed IDE Debuggers as a "game-changer" for controlled code inspection, echoing Kohn and Manaris (2020) insights on the benefits of step-by-step code examination and variable state inspection for novices. SDT22 highlighted IDE

Debuggers' role in identifying complex errors and setting breakpoints, a strategy Beller et al. (2018) supported for dissecting intricate code segments. SDT24 affirmed the importance of pausing code execution for precise examination, resonating with Petrillo et al. (2017) suggestion on interactive learning environments in programming. Finally, SDT29 focused on using IDE Debuggers to observe program behaviour at various stages (Beller et al., 2017), an approach in line with Papert (1980) constructionism theory, which advocates learning through interactive and real-time feedback tools. These examples collectively demonstrate the crucial role of IDE Debuggers in enhancing novice programmers' debugging strategy and overall programming understanding.

In contrast, techniques such as slicing and code review reflect a shift towards more contemporary and investigative debugging practices. Slicing, for instance, involves isolating specific portions of the code to understand their behaviour better and is particularly useful in large and complex codebases (Weiser, 1984). In a similar vein, code review, typically performed as a group effort, assists in detecting bugs and enhances code quality while fostering a shared understanding among developers (Bacchelli & Bird, 2013). Also, the evolution from basic techniques like trial and error to more advanced methods such as IDE Debuggers and code review in DYAD12 showed a developmental trajectory in debugging skills. This progression is crucial in building confidence and expertise, exemplified by DYAD13's pattern matching and code review use. Combining introspective methods like rubber duck debugging in DYAD7 with structured approaches like IDE Debuggers demonstrated the necessity of diverse problem-solving perspectives in addressing varied programming challenges.

Overall, the dyads' approaches to debugging illustrated an interplay between individual problem-solving techniques and collaborative efforts. The study revealed a mosaic of debugging strategies and tactics, each tailored to the apprentices' specific needs and skills' set. The combination of direct, immediate techniques and more exploratory, collaborative methods accentuates the complexities of debugging in programming. This blend of tactics facilitates effective problem-solving and contributes to a deeper understanding and proficiency in programming, preparing the apprentices for a wide range of programming challenges. Furthermore, it becomes apparent that collaboration is a fundamental aspect of their approach. Across the various teams, there is a pronounced reliance on cooperative techniques. This includes the collective use of Integrated Development Environments (IDEs), engaging in discussions during code reviews, and employing pair debugging methods such as rubber duck debugging, as observed in DYAD4 and DYAD7. These methods identify the apprentices' inclinations towards utilising teamwork as an effective tool to tackle the intricacies involved in debugging scenarios.

RQ3 – Cognitive Load Sharing

How do the paired geographically distributed SDT apprentices distribute cognitive load when resolving bugged code?

The approach to cognitive load management by geographically distributed SDT apprentices in resolving bugged code is multifaceted and well-aligned with key educational theories. Commencing with the foundational aspect of collaboration and role switching, the apprentices exhibited a dynamic interplay between the roles of 'driver' and

'navigator'. This approach, as exemplified by SDT1 and SDT8, finds backing in Plonka et al. (2011) and Williams and Kessler (2002). These research findings highlight the importance of collaboration and role switching in uniformly distributing cognitive load, thereby improving efficiency and focus, which are central to the approach of these dyads. This dynamic approach allows them to alternate between the 'driver', actively coding, and the 'navigator', providing guidance and oversight (Plonka et al., 2011).

Similarly, it can be argued that the 'driver-navigator' approach, practised by SDT1 and SDT8, is vital in managing cognitive load in collaborative programming. One member codes ('drives') while the other offers guidance ('navigates'), ensuring fair distribution of tasks as suggested by the dyads. Thus, regular role swaps, like every 15 minutes, keep both members equally engaged. This technique aids cognitive load management, aligning with Cognitive Load Theory, which posits limited information processing capacity and the effectiveness of collaborative strategies in distributing cognitive load (Sweller, 1988), though Tsai et al. (2015) suggest sharing workload does not significantly reduce germane cognitive load.

In the same vein, in the driver-navigator model of programming, dividing tasks between coding and reviewing can distribute cognitive demands, potentially lessening overload. This model traditionally sees drivers focus on coding and navigators on reviewing, each at different levels of abstraction, as noted by Beck (2000) and Williams et al. (2000). Contrarily, Bryant et al. (2008), Chong and Hurlbutt (2007), and Freudenberg et al. (2007) argue that both roles function at similar abstraction levels without distinct task division.

However, strict adherence to designated roles in this study suggests that traditional distinctions between driver and navigator may still hold significance.

Also, the driver-navigator programming model adheres to Hutchins (1995) distributed cognition concept, promoting shared cognitive processes among group members, enhancing understanding and problem-solving. Regular role switching, advocated every 15 minutes, encourages active engagement, a key element in collaborative learning (Johnson & Johnson, 1987). This approach ensures apprentices gain experience in coding and strategic aspects like problem-solving and code review, broadening their skill set. Additionally, it aligns with Vygotsky's social development theory (1978), highlighting the role of social interaction in cognitive development. Through this collaborative model, participants collectively construct knowledge, optimising cognitive resources and boosting learning outcomes via active engagement and social interaction.

Further, verbalising thought processes is critical in collaborative problem-solving within software development, particularly in debugging tasks. This method is evident in the interactions within the dyads, where articulate communication is a key factor in sharing and managing cognitive load. This approach suggests, in some cases, clear communication, which is pivotal in managing cognitive load among SDT apprentices. SDT2 exemplified this with the use of frequent, concise discussions for task division, aligning with Sweller (1988) cognitive load theory that reiterates reducing extraneous cognitive load enhances learning and problem-solving. Kirschner et al. (2006) further support this, advocating that well-structured collaborative tasks optimise learning by efficiently distributing cognitive load.

Additionally, DYAD2's adoption of the 'think aloud' method, where thought processes are openly discussed, resonates with Hmelo-Silver (2004) emphasis on articulating thoughts in collaborative problem-solving. By vocalising their reasoning and assumptions, team members can better track and understand each other's perspectives, leading to more cohesive and efficient problem-solving. This method ensures mutual understanding, aligning with Johnson and Johnson (1999) research, which highlights the role of effective communication in achieving shared goals within a team. Mayer and Moreno (2003) also acknowledge that such interactive communication reduces cognitive load, enhancing problem-solving efficiency (Paas et al., 2003). It can be argued that verbalising thought processes, as demonstrated in the dyad debugging sessions, is essential for managing cognitive load and fostering collaborative efficiency in software debugging. This approach aids in task articulation, ensures effective cognitive load distribution, and is supported by the principles of Vygotsky and Cole (1978) social development theory and Paas et al. (2003) findings on collaborative cognitive load management.

Similarly, the use of various tools and strategies emerges as crucial in addressing the distribution of cognitive load among paired geographically distributed SDT apprentices during debugging tasks. Thus, the deployment of IDEs, debuggers, and collaborative code editors plays a central role in this process. This approach resonates with Mayer and Moreno (2003) Cognitive Theory of Multimedia Learning, which highlights the efficacy of multimedia tools in reducing cognitive overload by facilitating more efficient information processing. Additionally, Sweller (1988) Cognitive Load Theory suggests that such tools

are instrumental in alleviating individual cognitive burdens, particularly in complex tasks like debugging, thus contributing to a more effective debugging process.

Furthermore, the division of specialisation within teams, as exemplified by apprentices in DYAD9, is a significant method for managing cognitive load. This strategy, backed by Paas et al. (2003), highlights the effectiveness of distributed cognitive load in collaborative learning environments. By assigning tasks based on individual strengths and areas of expertise, apprentices can optimise their cognitive resources. This concept is further supported by Kirschner et al. (2006), who emphasise the role of well-structured collaborative tasks in enhancing learning outcomes by efficiently distributing cognitive load among team members.

In addition to these strategies, balancing workload and effective time management, as highlighted by SDT21 and SDT29, is vital in averting cognitive overload. This approach is in line with the findings of Dillenbourg et al. (2009) on collaborative learning, underscoring the significance of workload distribution in collaborative settings. Such strategies ensure that apprentices direct their cognitive efforts toward the most impactful issues, optimising the overall debugging process and contributing to the team's success.

In summary, the SDT apprentices' strategies in managing cognitive load during debugging sessions demonstrated an alignment with the cognitive theories and adaptive problem-solving approaches in software development. These methods, comprising role-switching, verbalising thought processes, tool utilisation, specialisation, and workload management,

reflected a managed approach to cognitive load management, enhancing both individual and collective efficiency in software debugging tasks.

RQ4 - Leveraging IDE

RQ4: How does leveraging Integrated Development Environment (IDE) tools enhance the capabilities of distributed pair debugging and mitigate the challenges encountered in debugging programs?

The integration of IDE tools in distributed pair debugging of Python code is a complex interplay of benefits and potential pitfalls. This analysis, enriched by the experiences of apprentices across various dyads and supported by academic literature (Goldman et al., 2011; Potluri et al., 2022; Saträtzemi et al., 2023), offers an understanding of the role of IDE tools. Also, through an examination of their experiences and the insights gleaned from the debugging and the interview sessions, the impact of IDE tools on their collaborative debugging process becomes apparent as it serves more than just facilitators of code, but is crucial in addressing the challenges inherent in debugging. A crucial benefit of IDE tools, particularly Visual Studio Live Share, as highlighted by SDT1 and SDT17, is their facilitation of real-time collaboration. This aligns with Hutchins (1995) distributed cognition theory, which posits that cognitive processes are spread across individuals and their tools, enhancing problem-solving abilities. Johnson and Johnson (1987) and Salomon (1997) recognition of the importance of collaborative tools in software development further validates this point.

Similarly, this aspect of real-time collaboration is essential in overcoming physical distance, a point reinforced by the experiences of SDT3, SDT5, and SDT26, commending the efficacy of tools like Microsoft Teams and Visual Studio Live Share in fostering effective collaboration in enabling seamless communication and coordination despite geographical separation. However, the dependence on these specific tools invites critical scrutiny. The apprentices' reliance on these specific IDE tools raises concerns about the potential stagnation of essential debugging skills, as reliance on technology can lead to a lack of development in fundamental problem-solving abilities (Mayer, 2004). In the same vein, in situations where these specific tools are unavailable, this dependency could become a significant hurdle, potentially leading to a stagnation in the development of essential debugging skills.

Also, in the discourse on the use of IDE tools within various dyads, significant attention has been given to features like syntax highlighting, error notifications, and integrated consoles. These functionalities have been praised by various dyads for their efficiency in identifying and resolving syntax errors (Cheng et al., 2003; Goldman et al., 2011). This finding is consistent with research by Fontana and Petrillo (2021) and Petrillo et al. (2019). Participants, including SDT2 and SDT10, have particularly commended the capacity of these tools to streamline the debugging process (Kurniawan et al., 2015). They noted the utility of IDE features in simplifying code review and error detection. However, there is a risk that over-reliance on automated features could impede deeper learning of code principles, leading to a scenario where apprentices are proficient with IDEs but lack the skills to debug without these aids, a concern highlighted in the context of technology-assisted learning (Grover & Pea, 2013). This essentially implies that while these tools are

undoubtedly helpful for debugging, an over-reliance on them might impede a deeper grasp of essential coding principles, potentially leading to a skill gap. Apprentices might become proficient in using IDEs for debugging but could struggle without these aids, hindering the development of more fundamental and adaptable coding skills.

Moreover, advanced features like static code analysis and code coverage assessment in IDEs further enhance debugging efficiency as suggested by SDT19 and SDT22, add an extra layer of effectiveness to the debugging process as they facilitate the management of code changes and error correction and enable a deeper analysis of the code, allowing apprentices to address more complex issues beyond basic syntax errors. These techniques, aligning with Beller et al. (2018) research, enable participants to address issues beyond basic syntax errors, highlighting the comprehensive nature of IDE tools in the debugging process. These tools go beyond identifying syntactical errors and highlight more fine-grained aspects of code quality and performance. They allow SDT apprentices to adopt a proactive approach to debugging, anticipating potential issues before they become problematic. This can also be linked to the role of IDEs in reducing cognitive load, as per Sweller (1988) theory, which is significant in the debugging process. By automating routine aspects of coding, IDEs allow apprentices to focus on complex tasks. While these tools facilitate a proactive debugging strategy, helping apprentices to foresee and prevent potential issues, they also pose the risk of creating a dependency that could limit the development of essential programming skills and a deep understanding of code principles (Miller & White, 2021). This echoes the broader concerns related to the integration of educational technology and its impact on cognitive development in

programming education. Thus, this juxtaposition highlights the need for a balanced approach to IDE tool usage in programming education.

In the same vein, the integration of version control systems within IDEs is another aspect that aids in distributed debugging. The participants also recognised the importance of version control systems and code completion features in IDEs. SDT4 and SDT29 pointed out how these functionalities reduce cognitive burdens and improve debugging efficiency. This aspect is crucial in the distributed pair debugging context, as it allows for more efficient management of code changes and error correction. This suggests that version control systems help manage changes and coordinate tasks among team members and provide a safety net that encourages experimentation, a key component in creative problem-solving in software development. These systems are indispensable in efficiently managing code changes, particularly in a distributed setting. However, this raises the critical question of whether apprentices fully grasp the underlying principles of version control. While IDEs simplify this process, it is imperative for apprentices to develop a comprehensive understanding of version control mechanisms, a necessity opined by Loeliger and McCullough (2012). This knowledge is essential for managing code changes effectively, even when IDEs are not in use or in different coding environments, ensuring a well-rounded skill set in software development.

In summary, the analysis of participants' experiences and quotations, supported by relevant academic references, points out the integral role of IDE tools in enhancing the effectiveness of distributed pair debugging of Python code. These tools facilitate real-time collaboration and expedite error identification and resolution, reduce cognitive load,

and offer advanced functionalities for a more comprehensive debugging experience (Du Preez Ockert, 2019; Kölling et al., 2019). Also, the participants' affirmative feedback on using IDE tools in overcoming challenges, particularly in remote settings, reaffirms the pivotal role of technology in enabling seamless collaborative coding experiences.

RQ5 - Collaborative Debugging Challenges

What challenges are experienced by paired geographically distributed SDT apprentices working collaboratively on debugging programming bugs, and why are they facing such challenges?

Geographically distributed SDT apprentices engaged in pair debugging face numerous hurdles, including the complexities of remote collaboration and programming intricacies. Their challenges, compounded by the limitations of digital communication tools and varying levels of coding knowledge, involve effectively conveying complex coding concepts over distances and managing the cognitive load of resolving technical issues remotely. These factors collectively impinge upon the efficiency of the debugging process, leading to a range of issues that will now be explored in detail.

Technical and Cognitive Challenges in Remote Debugging:

A primary challenge faced by dyads in remote debugging sessions is encapsulated in the realm of technical and cognitive difficulties. Various aspects contribute to these challenges, notably the apprentices' struggles with complex logical errors. For example, apprentices in DYAD1 contended with intricate logical errors such as the misuse of the special variable 'name' and bonus calculation lacking context (domain knowledge). These

instances serve to highlight the significant challenges within the remote debugging landscape. Apprentices, grappling with the subtleties of Python's logic and syntax, frequently encountered obstacles, largely due to their limited experience. SDT1 articulated this challenge, stating, "One of the main obstacles... was dealing with complex logical errors... which was challenging given our limited experience". This experience stands in stark contrast to that of DYAD8, where apprentices adeptly utilised IDE tools to navigate similar challenges, thereby illustrating the uneven distribution of technical proficiency and problem-solving approaches among the dyads. This discrepancy suggests an underlying issue within remote programming education, indicating that while IDE tools provide significant support, they cannot substitute for a fundamental understanding of programming concepts, a gap particularly pronounced in remote settings where immediate peer or mentor support is absent.

Another aspect under the technical and cognitive theme is the management of cognitive load. The experiences of DYAD3, struggling with cognitive overload, illuminate the complex nature of this challenge. SDT6 noted the difficulty in managing the shared editing environment, a task that becomes increasingly challenging in a remote context where isolation can exacerbate focus issues. Contrasting these experiences with those of DYAD11, who struggled with poorly documented codebases, reveals the range of technical challenges in remote debugging. Such experiences affirm the need for comprehensive programming training that transcends mere technical skill development to encompass strategies for effective cognitive load management and documentation comprehension.

Additionally, the use of IDE tools presents its own set of challenges. Apprentices in DYAD5 and DYAD8 faced significant hurdles in mastering these crucial tools for remote debugging. SDT14 highlighted the difficulties in efficiently utilising the IDE Debugger remotely, stating, “another obstacle was efficiently utilising the IDE Debugger in a remote setting... this limitation made it difficult to collaboratively explore different hypotheses about the bug”. This learning curve contrasts with the experiences of DYAD8, where apprentices demonstrated greater proficiency with these tools. Such differences in tool mastery stress the importance of tailored training in remote debugging education, focusing on the technical operation of these tools and their integration into the learning and debugging processes.

Navigating poorly documented codebases, a challenge faced by apprentices in DYAD4 and DYAD11, adds another layer of complexity to remote debugging. SDT21 highlighted the struggle with logical errors, exacerbated by limited experience and unclear documentation, “One significant obstacle we faced... was dealing with the logical errors, especially considering our limited experience”. This struggle differs from the cognitive challenges faced by DYAD4, emphasising the need for a focus on comprehensive documentation skills within programming education.

Similarly, managing complex workloads and intricate code structures, as evidenced in the experiences of DYAD13 and DYAD15, points to the multi-dimensional nature of remote debugging. These apprentices needed to balance solving complex programming tasks with effective time and mental resource management, a challenge distinct from the technical issues faced by DYAD5. This necessitates an educational approach that includes

elements of project management and personal organisation to ensure that apprentices are technically proficient and adept at handling the broader demands of software development projects.

In conclusion, exploring these aspects in detail revealed the layered complexity of technical and cognitive challenges in remote debugging sessions. The diverse struggles of apprentices across different dyads marked the need for a comprehensive and varied approach to programming education. This approach should address the technical aspects of coding and the cognitive, collaborative, and organisational skills essential for effective remote debugging. Balancing the development of technical competencies with the cultivation of communication, problem-solving, and project management skills is crucial for preparing apprentices for the diverse challenges of the contemporary software development environment.

Communication and Collaboration Challenges:

The exploration of communication and collaboration challenges faced by apprentices in various dyads during remote debugging sessions, even with the aid of visual tools like Microsoft Teams, unveiled a complex landscape of interaction hurdles. Despite the visual connectivity offered by such platforms, the geographical separation between apprentices persisted as a significant barrier, demanding an enhanced focus on both verbal and non-verbal communication skills. As SDT1 aptly put it, “While tools like Visual Studio Live Share helped bridge the physical distance, we had to work harder to ensure clear and precise communication”. This sentiment was echoed by SDT2, who noted the necessity of

constantly verbalising thoughts to maintain a shared understanding, highlighting the ongoing struggle to overcome the absence of physical presence in digital interactions.

Moreover, this challenge was not isolated to DYAD1. For instance, SDT5 spoke of the difficulties in not being able to physically point out issues or discuss solutions, a sentiment that SDT6 also shared, particularly during simultaneous code editing, which often led to confusion. These experiences backs up the limitations of visual connections in fully compensating for the lack of direct, physical interaction, especially in understanding and managing shared coding tasks. The visual component, while beneficial, could not completely bridge the gap in immediate, intuitive understanding and response that physical presence facilitates.

Additionally, the dyads confronted the challenge of aligning their coding strategies and interpretations, a task made more difficult by geographical separation. SDT12's observation that face-to-face brainstorming could have expedited the process during moments of confusion points to the complexities of remote collaboration. Visual contact, albeit helpful, did not wholly mitigate the challenges posed by the need for immediate and coherent strategy alignment. This issue was further compounded by differences in individual coding experiences and styles, as highlighted by SDT11, who found it challenging to align coding approaches with their partner, indicating a deeper need for structured and systematic problem-solving approaches in remote settings.

Furthermore, the apprentices' struggles extended to the realm of effective communication, particularly in conveying complex programming concepts and thoughts.

SDT20's reflection on the extra effort required to explain thought processes and perspectives marks the inherent limitations of remote communication tools in replicating the depth and distinctness of face-to-face interaction. Similarly, SDT19's expression of difficulty in articulating thoughts to their partner reflects a broader issue within remote collaboration, where the complexity of conveying intricate ideas through digital means can hamper progress and understanding.

In synthesising these accounts from different dyads, it becomes evident that despite the advantages of visual communication tools, apprentices faced a wide array of challenges in remote debugging. These challenges encompassed the need for effective verbal and non-verbal communication and the intricacies of managing shared coding environments and harmonising diverse coding strategies. The experiences highlight the critical need for a comprehensive approach in remote programming education that goes beyond the mere use of technological tools. Such an approach should focus on developing robust communication and teamwork skills, addressing the complexities of remote collaboration, and ensuring apprentices are well-prepared to navigate the multifaceted challenges of modern software development. This comprehensive approach is essential for fostering a collaborative, effective, and adaptable learning environment in the ever-evolving field of software development.

Environmental and Logistical Challenges:

The exploration of environmental and logistical challenges faced by SDT apprentices in remote debugging sessions, as reflected in their direct experiences, revealed intricate complexities and multifaceted nature of these hurdles.

Furthermore, geographical separation profoundly impacted the dynamics of communication and collaboration, as evidenced by the experiences of various dyads. This finding aligns with the suggestion put forward by Satratzemi et al. (2018) that distributed pair programming (DPP) is more demanding than traditional pair programming (PP). For instance, SDT1 remarked, “Additionally, being geographically dispersed posed its own set of challenges... we had to work harder to ensure clear and precise communication”. This sentiment was echoed by SDT2, who highlighted the necessity of constant verbalisation to maintain a synchronised understanding of the code. Similarly, SDT3 and SDT4 experienced delays in real-time collaboration due to remote setup, with SDT4 noting, “Due to our remote setup, we faced delays in real-time collaboration, even with the aid of live code-sharing tools”. These quotes underline the challenges posed by physical distance, necessitating enhanced communication strategies to bridge the gap.

In addition to these communication challenges, the need for effective coordination within shared digital spaces was another significant challenge. SDT5 expressed difficulties in not being able to physically point out issues, stating, “Being geographically dispersed meant we couldn’t simply look over each other’s shoulder to point out issues or discuss solutions”. This issue of managing shared coding environments effectively was also highlighted by SDT23, who mentioned, “The geographical separation added another layer of complexity. We relied heavily on digital communication tools”. These reflections point to the challenges in synchronising understanding and actions in a remote collaborative setting.

Compounding these issues, technological limitations and connectivity issues added another dimension to the challenges faced. SDT29 discussed the impact of internet connectivity on their workflow, noting that “Our different locations also meant we were working in different environments... relying on technology meant we were at the mercy of our internet connections, which occasionally disrupted our flow”. This highlights the need for reliable infrastructure and robust digital tools to facilitate seamless remote collaboration.

Moreover, the varied experiences across dyads illustrated the diverse nature of environmental and logistical challenges. While apprentices in DYAD8 and DYAD14 utilised IDE tools effectively, they faced specific challenges unique to their situations. For example, SDT15 noted, “Being geographically dispersed meant that we couldn’t quickly huddle and draw out our thoughts on a whiteboard or paper”. In contrast, SDT27 mentioned, “The challenges we faced... were largely due to the nature of remote communication... The lack of immediate, direct interaction made it more challenging to collaboratively and swiftly navigate through these complex issues”.

Consequently, these insights from apprentices across various dyads paint a picture of the environmental and logistical challenges encountered in remote debugging sessions. They emphasise the need for strategies that effectively bridge the geographical gap and address the unique demands of remote collaboration. This more comprehensive approach should focus on developing technical skills and enhancing communication, teamwork, and adaptability to diverse technological landscapes, preparing apprentices for the evolving challenges of modern software development.

However, in addressing the central question, “How do the paired Software Development Apprentices in geographically distributed locations work collaboratively to fix Python programming bugs using the technology-mediated medium?”, this study examines the five interrelated sub-questions that underpin the investigation. These sub-questions explore the nature of errors encountered, the debugging strategies and tactics used, the mechanisms for cognitive load distribution, the role of Integrated Development Environments (IDEs), and the challenges apprentices face in collaborative debugging. The findings reveal that apprentices adopt a highly structured yet adaptable approach to debugging, integrating problem-solving techniques, cognitive flexibility, and technology-supported collaboration. Despite the complexities and limitations associated with remote debugging, apprentices demonstrate resilience, adaptability, and an evolving mastery of debugging processes. Their ability to effectively communicate, structure their debugging efforts, and synchronise their workflows plays a crucial role in ensuring efficiency in distributed pair debugging.

Building on this, the study finds that apprentices encounter three primary categories of errors: syntax errors, logical errors, and runtime errors. Syntax errors occur when Python’s structural rules are violated, leading to issues such as missing colons, incorrect indentation, or misused operators. Since these errors produce immediate feedback from the interpreter, they are often straightforward to resolve. However, failing to address them efficiently can hinder progress and obscure deeper logical flaws. To resolve syntax errors, apprentices predominantly rely on print statement debugging, which allows them to observe variable states and track execution flow. While this method is highly effective

for identifying and correcting syntax issues, it becomes less useful when dealing with more complex errors that require deeper reasoning.

Extending beyond syntax issues, logical errors present a greater challenge as they do not produce explicit error messages but instead result in incorrect program behaviour. These errors frequently stem from misapplied conditional logic, flawed tax computations, or improperly structured loops, leading to unintended outcomes. Unlike syntax errors, which can often be corrected quickly, logical errors require a more systematic approach to debugging. Step-through debugging, where apprentices execute the code line by line while observing variable changes and function calls, proves particularly effective in diagnosing these errors. Furthermore, backtracking, where apprentices systematically review previous modifications to pinpoint when an error was introduced, plays a crucial role in isolating logical faults. However, this process can be mentally taxing, particularly in large programs with multiple interdependencies.

In addition to syntax and logical errors, runtime errors are the most unpredictable and complex to debug, as they only emerge during program execution. Examples include infinite loops, incorrect type conversions, and index errors, which can cause the program to behave erratically or even crash. Unlike syntax errors, which apprentices can address through static code analysis, runtime errors often require more extensive debugging efforts. Resolving these issues demands a combination of approaches, including trial and error, slicing, and code review. When runtime errors prove particularly elusive, apprentices frequently resort to rubber duck debugging, which involves verbalising their thought process to clarify their understanding. This technique often helps apprentices

identify overlooked logic flaws, reinforcing the importance of metacognition in debugging.

To manage these different types of errors, apprentices strategically employ ten distinct debugging methods, each offering unique advantages depending on the nature of the bug. As previously noted, print statement debugging remains the most frequently used approach due to its simplicity and immediate feedback, allowing apprentices to track variable states and execution flow. However, while print statements provide insight into syntax-related issues, they lack the precision needed to resolve deeper logical and runtime errors. To address these limitations, apprentices frequently use step-through debugging, facilitated by IDE debugging tools, to execute code incrementally, set breakpoints, and monitor changes in variable states in real-time. This method proves particularly useful in identifying subtle logical errors, yet its effectiveness is dependent on the apprentice's proficiency in using debugging tools.

Along with these structured methods, apprentices also engage in tinkering, where they experiment with incremental modifications to the code to observe how different changes impact execution. While this approach fosters exploratory learning and intuitive problem-solving, it lacks structure and can lead to inefficiencies if used indiscriminately. Similarly, trial and error, although valuable when the nature of the bug is unclear, can be time-consuming and unreliable if apprentices fail to document and analyse their attempts systematically. Therefore, while experimentation is an essential aspect of debugging, it must be balanced with structured techniques to ensure efficiency.

Furthermore, more methodical debugging approaches, such as backtracking, allow apprentices to trace errors back to their origin, making it easier to identify when and where a mistake was introduced. GitHub's version control system significantly enhances this process, as it enables apprentices to compare different iterations of their code and revert to previous versions when necessary. Additionally, code review plays a vital role in debugging, as apprentices critically evaluate each other's work, offering feedback, suggestions, and alternative solutions. This method enhances collaboration and debugging efficiency, as errors that might be overlooked by one apprentice can be identified by their partner, reinforcing the value of shared problem-solving.

Expanding on these strategies, pattern matching further contributes to debugging efficiency by enabling apprentices to identify recurring error types and apply solutions based on past experiences. This approach demonstrates a transition from trial-and-error methods to structured problem-solving, highlighting the apprentices' growing debugging expertise. Additionally, slicing, which involves isolating specific sections of the code for in-depth examination, significantly reduces cognitive overload, allowing apprentices to focus on smaller, more manageable code segments. When combined with other strategies, slicing ensures a systematic approach to debugging, preventing unnecessary effort spent on scanning the entire codebase.

Another effective technique is divide and conquer, which is especially useful for complex debugging tasks. Here, apprentices split the program into smaller sections, debugging individual parts independently before integrating their solutions. This approach improves

efficiency and minimises cognitive strain, ensuring that both apprentices remain engaged and contribute actively to the debugging process.

Since debugging is not only a technical task but also a cognitively demanding process, apprentices must employ effective cognitive load management strategies. The driver-navigator model remains the primary approach, with one apprentice writing or modifying code while the other provides real-time oversight and guidance. However, while this approach fosters structured collaboration, its effectiveness depends on regular role-switching, as prolonged navigation without hands-on coding can result in reduced engagement and passive participation. Additionally, verbalisation strategies, such as thinking aloud and articulating reasoning, play a crucial role in clarifying thought processes and ensuring mutual understanding. These strategies prevent misinterpretations and enhance collaborative problem-solving, particularly in remote environments where non-verbal cues are absent.

Given the geographical separation of apprentices, technology plays a crucial role in bridging the gap and enabling effective debugging. Visual Studio Live Share provides a shared environment for real-time collaboration, allowing apprentices to simultaneously edit, execute, and debug code, mimicking the experience of co-located pair programming. Similarly, GitHub's version control features support structured debugging workflows, ensuring that apprentices can track modifications, revert changes, and maintain a history of code updates. Microsoft Teams, Zoom, and Slack further facilitate verbal discussions and screen sharing, enabling apprentices to communicate effectively despite physical distance.

Despite these technological advantages, several challenges persist. Over-reliance on IDE debugging features can lead to superficial problem-solving approaches, where apprentices depend too much on automated tools instead of developing deeper analytical skills. Moreover, communication barriers in remote debugging introduce delays and inefficiencies, particularly when apprentices struggle to articulate complex programming issues without face-to-face interaction. Additionally, cognitive overload remains a significant challenge, as apprentices must juggle multiple cognitive demands simultaneously.

Ultimately, the study highlights that distributed debugging is not merely a technical task but a complex cognitive and collaborative process. Apprentices must balance structured debugging methodologies with adaptive learning, integrate technology effectively without over-reliance, and refine their independent problem-solving skills while engaging in collaborative debugging. Despite the inherent challenges of remote debugging, apprentices demonstrate progressive mastery of debugging techniques, showcasing the potential for effective software development training in distributed settings. The research underscores the importance of structured learning, technological facilitation, and cognitive load management in fostering efficient and scalable remote debugging practices.

6.3 Refined Conceptual Framework Linking Research Outcomes to Distributed Debugging Processes

Understanding how research outcomes align with the conceptual framework is crucial to comprehending the mechanisms underpinning distributed debugging processes. This refined framework integrates Information Foraging Theory (IFT) (Pirolli & Card, 1999) and Distributed Cognition (DC) (Hutchins, 1995) to provide a structured perspective on how software development apprentices collaborate to debug Python programming errors in technology-mediated environments. By examining how apprentices seek and process information, distribute cognitive effort, and leverage debugging tools, this framework offers a comprehensive lens through which debugging behaviours can be analysed and optimised.

At the centre of this framework is Distributed Pair Debugging, which encapsulates the interplay between cognitive, behavioural, and technological factors that shape debugging in remote settings. Debugging is not a solitary task but a process that requires structured information retrieval, strategic collaboration, and effective technological support. As apprentices engage in distributed debugging, they must balance information foraging with cognitive load distribution, ensuring that problem-solving remains efficient and structured. The research findings confirm that the success of debugging depends on how effectively apprentices integrate these elements, reinforcing the need for a systematic approach to collaborative problem resolution.

A key component of this framework is Information Foraging Theory (IFT), which explains how apprentices search for, evaluate, and apply debugging information. Debugging

requires programmers to navigate multiple information sources, including error messages, documentation, online forums, and past code iterations. The ability to identify useful information efficiently and distinguish between relevant and irrelevant data directly affects the speed and accuracy of debugging. The research findings indicate that apprentices who develop effective information-seeking behaviours are more successful in applying structured debugging techniques, as they can quickly access and interpret the necessary resources without unnecessary delays.

However, acquiring information is only one aspect of debugging. Distributed Cognition (DC) complements IFT by explaining how cognitive processes are shared between apprentices and the tools they use. Debugging in a distributed setting involves continuous coordination, shared cognitive effort, and strategic tool utilisation. The research highlights that cognition is not confined to individual minds but distributed across pairs and the technological ecosystem they operate within. Apprentices must not only externalise their thought processes through verbalisation and structured discussions but also use debugging tools effectively to distribute cognitive workload. This shared cognition ensures that problem-solving remains fluid and adaptive, preventing any single apprentice from being overwhelmed by the complexity of the debugging task.

The research also reveals that Debugging Strategies act as a bridge between information foraging and problem resolution. Apprentices employ a range of techniques, including print statement debugging, step-through debugging, backtracking, pattern matching, slicing, and trial-and-error approaches. While some strategies, such as print statement debugging and trial-and-error, are exploratory, others, such as step-through debugging

and backtracking, require structured reasoning and systematic problem-solving. The conceptual framework highlights that debugging strategies must align with the nature of the error, ensuring that apprentices apply the most effective method for each debugging scenario. Without a structured approach, debugging becomes inefficient, leading to prolonged problem resolution times and increased cognitive strain.

Equally significant is Cognitive Load Management, which determines how effectively apprentices sustain focus and manage the demands of debugging. Debugging can be mentally taxing, particularly when apprentices must juggle multiple problem-solving tasks while collaborating in real time. The research findings emphasise the importance of role-switching strategies, such as the driver-navigator model, in ensuring equitable participation and reduced cognitive fatigue. By alternating roles, apprentices maintain an active engagement in debugging while balancing cognitive effort, preventing one individual from bearing the entire cognitive burden. Furthermore, verbalisation techniques, such as think-aloud protocols, help externalise reasoning and reinforce shared understanding, ensuring that both apprentices remain aligned in their debugging efforts.

The role of Technology-Mediated Tools is another fundamental aspect of this framework, as tools serve as both cognitive extensions and collaboration enablers. The research findings confirm that IDE debugging features, version control systems, and real-time collaboration platforms enhance efficiency, structure debugging workflows, and improve coordination between apprentices. Visual Studio Live Share facilitates synchronous debugging, allowing apprentices to view and modify code simultaneously, which

replicates the experience of co-located debugging sessions. Likewise, GitHub's version control capabilities enable structured debugging by allowing apprentices to track changes, document problem-solving processes, and revert to previous working versions when necessary. These tools not only support problem-solving but also reduce cognitive load by automating certain debugging tasks, enabling apprentices to focus on logical problem-solving rather than manual syntax corrections.

Nevertheless, while technology is a powerful enabler, it must be used strategically rather than as a substitute for fundamental debugging skills. The research highlights that over-reliance on automated debugging features can lead to superficial problem-solving approaches, where apprentices depend on error-highlighting tools rather than developing a deep understanding of debugging principles. Therefore, the conceptual framework reinforces that technology should facilitate, rather than replace, structured debugging methodologies, ensuring that apprentices cultivate both technical proficiency and problem-solving expertise.

A further critical insight from the research is the role of Collaboration and Shared Understanding in debugging success. Since apprentices operate in geographically distributed settings, effective debugging relies on clear communication, structured discussions, and synchronised problem-solving efforts. The research highlights that successful debugging pairs engage in continuous dialogue, share mental models of debugging problems, and refine solutions through collaborative reasoning. However, when collaboration is poorly structured or lacks clear communication protocols, debugging efforts become fragmented and inefficient, leading to duplicated efforts,

misinterpretations, and unresolved issues. The conceptual framework underscores that collaborative debugging is most effective when supported by structured coordination strategies, active engagement, and clear documentation of debugging steps.

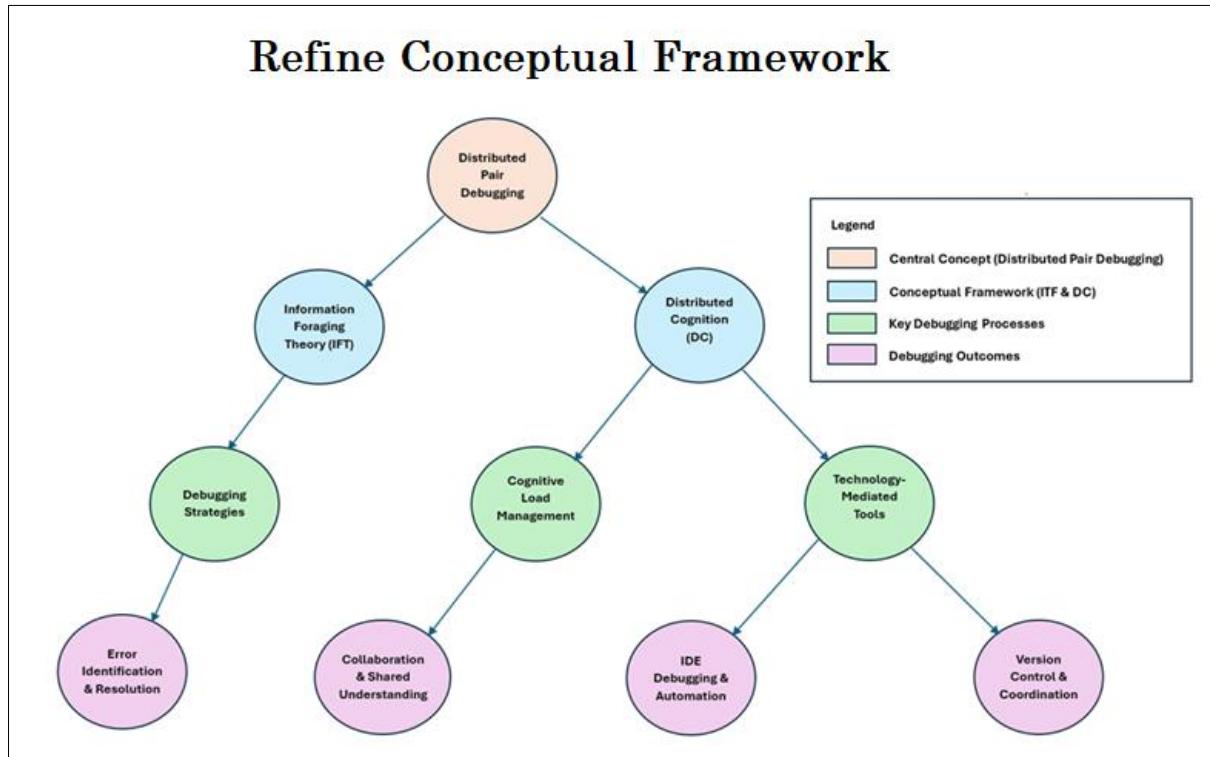


Figure 11: Refined Conceptual Framework Aligning Research Outcomes to Distributed Debugging Processes

Bringing these elements together, the refined conceptual framework (see Figure 11) presents a structured representation of how research outcomes align with the debugging process. It provides a multi-layered model that accounts for the interplay between theoretical constructs, debugging methodologies, cognitive processes, and technological interventions. Figure 11 visually encapsulates these relationships, illustrating how Distributed Pair Debugging is shaped by the integration of Information Foraging, Distributed Cognition, Debugging Strategies, Cognitive Load Management, and Technology-Mediated Tools.

This framework not only enhances understanding of how apprentices engage in distributed debugging but also provides valuable insights for software development education, training, and the design of debugging tools. By offering a structured approach to information navigation, collaborative cognition, and debugging strategy selection, the research contributes to a better-informed methodology for teaching and improving debugging practices in remote environments.

6.4 Novelty of this Work

The novelty of this study can be appreciated through three distinct focal points that contribute to the existing body of knowledge in a number of research areas. Notably, at the time of this research, there was a noticeable void in studies specifically targeting work-based learning environments in this sector, a gap that has persisted since 1973. This shortfall is particularly pronounced in investigations into the debugging practices of novice programmers, such as apprentices. While numerous studies have explored debugging strategies and tactics (Alaboudi & LaToza, 2023; Allwood & Bjorhag, 1990; Alqadi & Maletic, 2017; Fitzgerald et al., 2008; Fitzgerald et al., 2010; Gould, 1975; Gugerty & Olson, 1986; Jayathirtha et al., 2020; Katz & Anderson, 1987; Lee et al., 2014; Murphy et al., 2010; Murphy et al., 2008; Romero et al., 2007; Weiser, 1982; Yen et al., 2012), they predominantly focus on school and academic environments or on seasoned developers in realistic settings (Alaboudi & LaToza, 2023). Therefore, this study addresses a significant research gap by examining the debugging strategies and tactics of a previously unexplored group, the novice programmer apprentices in work-based learning environments.

Similarly, the distinctiveness of the study's participants contributes to its originality. Contrary to many research projects centred on specific demographics or professional groups, this study's participants represent a diverse array of learners employed across various sectors. Their variety in employment histories, employers, and age ranges enriches the research, offering a wider viewpoint on debugging practices due to the participants' extensive demographic range. This diversity in participant profiles suggests potential wider generalisability of the findings, although not the primary aim of this study, and reinforces the transferability of debugging skills across varied work-based settings.

Secondly, the study's distinctiveness is further highlighted by the lack of any prior empirical research on distributed pair debugging within the work-based learning sector. Although there are a limited number of studies on pair debugging, they do not specifically focus on debugging strategies (Jayathirtha et al., 2024; Murphy et al., 2010; Parkinson et al., 2024). The concept of pair debugging in distributed settings, particularly within educational contexts, had not been explored at the time of this study. While research on distributed pair programming has been recognised for its role in enhancing collaboration and problem-solving in software development (Baheti et al., 2002; Hafeez et al., 2023; Xu & Correia, 2023), the specific aspect of debugging within distributed pairs has remained largely unexamined. This study fills this void by delving into the unique challenges and strategies of debugging in a distributed environment, thereby enriching the understanding of collaborative debugging practices in software development.

Thirdly, a noteworthy contribution of this study is the creation of an innovative conceptual framework for distributed debugging (see Figure 4 and Section 3.4), which merges two theoretical frameworks, which are distributed cognition (Hutchins, 1995) and information foraging theories (Pirolli & Card, 1999). This new framework acts as a theoretical construct that structures the research findings and offers a resource for future research and practical applications. The development of this framework marks an advancement as it synthesises and systematises the knowledge gained from the study, enhancing its applicability across various contexts and settings (Bryman, 2016).

In conclusion, the novelty of this study is multi-faceted, encompassing its focus on work-based learning environments, the exploration of distributed pair debugging, the development of a conceptual framework, and the diversity of its participant profiles. These aspects collectively contribute to advancing knowledge in debugging practices and offer insights for both academia and the learning and development sector.

6.5 Contributions

My thesis makes notable contributions to computing education and the practical implementation of technology in education, potentially influencing the approaches of educators and practitioners in these fields.

One of the key empirical insights of my thesis lies in its ability to shed light on the debugging strategies and challenges within the work-based learning sector, leveraging a detailed compilation of experiences from 30 apprentices, further enriched by the perspectives of 12 mentors and trainers. This diverse cohort unveils an insight into

collaborative problem-solving in debugging within the distributed environment, setting a solid foundation for broader applicational insights. Additionally, the methodological and analytical depth employed in this study reinforces its potential wider generalisability, suggesting that the varied backgrounds of apprentices in terms of their ages, educational background, diverse employers and experiences of the participants may mirror the sector's complexity, thereby extending the relevance of these findings beyond the immediate study context. The research benefits significantly from the dual perspectives of apprentices and their mentors, offering a comprehensive view of the debugging process that highlights the critical role of guidance and support. This approach reinforces the study's potential broader applicability and signals its potential to inform educational practices and professional development across the sector. Similarly, including participants from a broad spectrum of backgrounds further strengthens the study's position as a resource for educators and policymakers alike, emphasising its capacity to address specific needs and challenges within work-based learning environments. Thus, this thesis stands as a testament to the value that diverse, collaborative insights offer in shaping educational strategies and policies.

Furthermore, my research extends its empirical contributions into the broader area of technology in education. In today's digital age, integrating technology into educational settings is both pervasive and potentially transformative. My thesis acknowledges students' challenges and presents innovative ways to harness technology to support and enhance an apprentice's learning. This practical aspect of my work holds particular relevance for educators, instructional designers and work-based mentors seeking evidence-based insights on effectively leveraging technology to support their learners. By

offering practical recommendations grounded in empirical research, my thesis serves as a guide for those navigating the ever-evolving landscape of the use of technology in debugging.

The technical contributions of my thesis provide an insight into the key technical aspects in computing education in the context of debugging strategies within distributed settings. Prior research, such as that of Katz and Anderson (1987), has explored debugging in solo and collocated environments, but there is a gap in understanding how novices integrate these strategies in distributed contexts. My study addresses this by examining the unique dynamics of distributed debugging, necessitating new analytical frameworks. It merges distributed cognition and information foraging theory to explore collaborative interactions and tool usage in error detection within code. This approach is significant as it extends beyond the well-documented novice debugging strategies of the 1980s, which were largely based on cognitive theories (McCauley et al., 2008). My research uncovers the specific strategies and challenges of debugging in distributed settings, providing insights for both novice and seasoned programmers. These findings add to the academic discourse and have practical implications, potentially transforming how debugging might be taught and practised, thereby potentially enhancing the proficiency and adaptability of programmers.

In summary, this thesis provides educators and practitioners with empirical insights and technical developments, aiding them in navigating the complexities of technology-enhanced debugging, which refers to the process of identifying and fixing errors (bugs) in software code using advanced technological tools and methods, and how best to support

the novice through the details provided. By delivering evidence-based recommendations and insights, the research is designed to advance the use of technology within these contexts. This, in turn, can enrich the learning experience for learners and support the cultivation of proficient and resilient professionals in computing.

6.6 Significance of the Study

This evaluative case study on the program debugging behaviour of paired SDT apprentices in a geographically distributed environment holds significant implications for both educational and industry contexts. This study addresses a notable gap in the literature by investigating debugging strategies in distributed, collaborative settings among novice programmers. The significance of this research can be elucidated through the following points.

Educational Enrichment and Curriculum Development: This study offers valuable insights into how novice programmers (SDT apprentices) approach debugging in remote, collaborative settings. By understanding their strategies, challenges, and successes, educators can tailor their curricula to better prepare apprentices for real-world software development challenges. Incorporating findings into apprenticeship programmes can enhance learning outcomes and equip apprentices with skills more aligned with industry demands. It allows educational institutions to bridge the gap between theoretical learning and practical application.

Industry Relevance and Software Quality Enhancement: The software development industry increasingly operates with geographically distributed teams (Herbsleb & Moitra,

2001), particularly in the wake of remote work trends accelerated by the COVID-19 pandemic. Understanding how debugging occurs in these settings can directly impact software quality (Beasley & Johnson, 2022), as effective debugging is critical for delivering robust and reliable software products. Insights from this study can guide software development teams in refining their collaborative debugging practices, resulting in more efficient and higher-quality code.

Remote Collaboration Strategies: Collaborative debugging among remote pairs introduces unique challenges related to communication, coordination, and information sharing. By investigating how dyad SDT apprentices tackle these challenges, the study contributes to understanding effective remote collaboration strategies. Such insights can inform the development of best practices for distributed software development teams, ensuring smoother communication and improved teamwork. The study results may provide insight into the type of error messages generated by SDT apprentices while debugging codes, their debugging strategies and how pairing novice programmers in different locations works. Also, it will help share good practices from other mentors about how best to support apprentices with low debugging skills.

Pedagogical Innovations and Tool Development: The study's findings encourage pedagogical innovations around teaching debugging techniques. Educators can leverage these insights to design more effective methods for teaching debugging skills to novice programmers, focusing on particularly relevant strategies in remote and collaborative settings. Additionally, the research can inform the development of tools and technologies tailored to support debugging in geographically distributed environments. This could lead

to the creation of debugging tools that facilitate remote collaboration and enhance efficiency.

Cognitive Processes and Distributed Cognition: Investigating how SDT apprentice dyads share the cognitive load while debugging unveils the intricacies of distributed cognition in collaborative software development (Hutchins, 1995). By understanding how individuals distribute tasks, make decisions, and solve problems together, the study contributes to the growing body of knowledge on cognitive processes in distributed teams. This understanding can lead to better collaboration frameworks and enhanced coordination mechanisms.

Enhancing Industry-Academia Collaboration: The findings of this study can foster stronger collaboration between educational institutions and the software development industry. The insights gained can be shared with industry partners to inform their practices and expectations of novice programmers. This collaboration ensures that industry needs can be met and educational programmes produce graduates who are well-equipped to contribute effectively to real-world development scenarios.

In conclusion, this evaluative case study has significance, as it could potentially impact both education and industry by offering insights into debugging practices and team synergy among geographically distributed SDT apprentices.

6.7 Trustworthiness of the Study

The foundational work of Lincoln and Guba (1985) is pivotal in establishing a framework for trustworthiness, comprising four essential criteria, including credibility, transferability, dependability, and confirmability.

Credibility, as posited by Lincoln and Guba (1985), pertains to the believability and truth value of the findings. In ensuring the credibility of the study involving apprentices and WMTs, I engaged in sustained observation, documenting and analysing the multi-faceted of practical debugging sessions and dyad interviews (see Appendix J: Sample transcript of the debugging session and Appendix K: Sample transcript of dyad's interview). This approach, endorsed by Shenton (2004), facilitated an immersion into the apprentices' experiences, offering a portrayal of their debugging skills that was as authentic as possible. Similarly, the study also combined the think-aloud verbal protocol during the debugging session (Ericsson, 2006) with retrospective post-debugging dyad interviews, akin to Murphy et al. (2008). This approach let quieter apprentices during the debugging session explain their actions and thoughts later, though it risked rationalised responses (Ericsson & Simon, 1993). The benefit of this method was that it potentially provided more in-depth insights into the dyads' strategies and misconceptions (Whalley et al., 2023). Furthermore, the integration of two debriefing sessions with WMTs further enriched this narrative, providing a multifaceted perspective on the apprentices' developmental trajectory. The study used methodological triangulation, as suggested by Carter et al. (2014), coalesced diverse data sources such as practical debugging sessions, Python code analysis, interviews with dyads, and WMTs' insights. This approach strengthened the study's reliability by corroborating and validating its findings.

Transferability, as elucidated by Lincoln and Guba (1985), addresses the applicability of findings in other contexts. To facilitate this, the study provided descriptions of the apprentices' environments, backgrounds, and experiences, as well as the diverse professional contexts of the WMTs. This approach, resonates with Geertz (1973) concept of thick description allowed for a good grasp of the apprentices' settings. Such detailing may equip other researchers with the necessary context to evaluate the potential applicability of these findings in analogous settings.

Dependability focuses on the consistency and stability of the findings over time, a crucial aspect affirmed by Lincoln and Guba (1985). The study embraced an iterative approach, continually revisiting and refining the data in light of emerging insights, a strategy supported by Morse (1994). As recommended by Rodgers and Cowles (1993), the maintenance of an exhaustive audit trail provided a transparent and comprehensive account of the research process, encompassing facets of data collection and analysis. This documentation of the study's methodology, encompassing both apprentices' and WMTs' contributions, underpins the dependability of the research, ensuring that the study's process is transparent, replicable, accountable, and consistent.

Confirmability, the fourth criterion in Lincoln and Guba's framework, relates to the degree to which the respondents shape the findings, not by researcher bias or predispositions. To achieve this, the researcher maintained a reflexive journal, a practice supported by Schwandt (2001), to record personal biases and reflections, thereby enhancing the objectivity of the research. This reflexive practice was crucial during the

analysis of the apprentices' debugging sessions, interviews, and the focus group discussions with WMTs. In these analyses, special care was taken to root interpretations in the data, utilising direct quotes and specific examples from the sessions. This practice effectively anchored the study's findings in the authentic experiences and perspectives of the participants, thereby bolstering the confirmability of the research.

6.8 Limitations of the Study

Looking back on the research conducted for this thesis, it is evident that while it constitutes an original contribution to the field, it also inevitably encompasses certain limitations and weaknesses.

First, the phenomenon of retrospective rationalisation, where participants reinterpret actions and thoughts after the fact, introduces potential discrepancies between actual and reported behaviours, as participants might align their narratives with perceived expectations or beliefs (Tufford & Newman, 2012).

In addition, the inherent complexity of qualitative data, sourced from diverse mediums like video recordings and interviews, presents considerable challenges in achieving thematic interpretation and analysis consistency (Saunders et al., 2023). Furthermore, this complexity is exacerbated by the emotional and psychological impacts on apprentices when observed, potentially influencing their responses and behaviours. Despite these challenges, qualitative research provides profound insights often beyond the reach of localised surveys, as suggested by Grant and Booth (2009).

Also, the study's focus on Python programming and Microsoft Visual Studio as the primary IDE presents a limitation in its scope, specifically in capturing the variety of challenges and strategies in diverse programming languages and environments. This is significant, as each programming language and IDE possesses unique intricacies that influence the debugging process (Murphy et al., 2006; Robins et al., 2003). Therefore, the study's findings might neither fully encompass the breadth of challenges faced in different software development contexts, nor account for the potential evolution of apprentices' debugging skills and strategies over a more extended period.

Furthermore, the rapid advancement of software development tools and practices compounds this limitation. The field's dynamic nature, with new languages, frameworks, and methodologies continually emerging, may render the study's findings, focused on specific technologies, less relevant in the long term (Rajlich & Bennett, 2000). This evolving landscape of software development suggests that the study's insights, while pertinent in the current context, might not maintain their applicability as new technologies and practices develop. Such limitations corroborate the importance of continuous research and adaptation in the field to stay abreast of these technological shifts.

Additionally, the study's reliance on digital communication platforms like Microsoft Teams introduces unique challenges. While facilitating remote collaboration, these platforms may lead to technical issues, reduced nuances in communication compared to face-to-face interactions, and disparities in digital literacy among participants. Such factors can significantly influence the dynamics of debugging sessions and interviews.

Consequently, the findings drawn from such digitally mediated interactions should be interpreted with caution, particularly when considering their applicability to different software development environments or settings where digital communication may not be as prevalent.

Moreover, the study's specific cultural and organisational focus raises questions about the generalisability of its findings to diverse contexts. As qualitative research often reflects the unique circumstances of its setting (Hsieh & Shannon, 2005), the experiences of apprentices and mentors in this study, conducted within a particular organisational culture, might not accurately represent those in different technological or organisational environments. This limitation is essential to consider when applying the study's insights to varied contexts, as they may not translate seamlessly across different organisational cultures or technological landscapes.

Additionally, the study's reliance on digital communication platforms like Microsoft Teams introduces unique challenges. While facilitating remote collaboration, these platforms may lead to technical issues, minimised subtleties in communication compared to face-to-face interactions, and disparities in digital literacy among participants. Such factors can significantly influence the dynamics of debugging sessions and interviews. Consequently, the findings drawn from such digitally mediated interactions should be interpreted with caution, particularly when considering their applicability to different software development environments or settings where digital communication may not be as prevalent.

On top of this, the process of validating findings with apprentices and mentors, though intended to enhance reliability, is not immune to confirmation bias. This bias occurs when individuals, including researchers and study participants, are more likely to agree with interpretations that align with their pre-existing beliefs or expectations (Nickerson, 1998). In this context, apprentices and mentors might unconsciously affirm findings that resonate with their experiences or perspectives, thereby reinforcing the researcher's initial interpretations. Although designed to strengthen the study's credibility, this feedback loop necessitates careful management to avoid reinforcing potentially skewed perspectives. Hence, while seeking validation from participants adds robustness, careful handling is required to mitigate the risks of confirmation bias, ensuring a more balanced and objective analysis of the data.

Another notable limitation of this study pertains to the applicability of the debugging tasks for younger apprentices, particularly in relation to the salary and tax issues embedded within them. These topics, while relevant to software development in business contexts, may not have been fully comprehensible or relevant to younger participants who lacked prior experience or understanding of such real-world concepts. Apprentices, especially those at the early stages of their careers, may not have had sufficient exposure to financial concepts like salary calculations and tax systems, which could have created a barrier to their engagement with the tasks. Studies indicate that for learning to be effective, tasks must align with the learners' cognitive developments and prior knowledge (Alexander, 2003). When tasks are overly complex or disconnected from participants' experiences, they may struggle to engage meaningfully, leading to sub-optimal learning outcomes (Kirschner et al., 2006).

This limitation highlights the importance of designing problem sets that are universally relevant to all learners, regardless of their age or background knowledge. By ensuring that the tasks used in future studies reflect scenarios that are relatable and within the comprehension of all apprentices, the study could enhance both the engagement and performance of participants. This approach is supported by educational theory, which suggests that contextualising learning materials to the learners' experience enhances cognitive engagement and motivation. For example, incorporating tasks that simulate everyday programming challenges rather than complex business scenarios could improve the accessibility of the debugging tasks for younger apprentices.

In conclusion, despite its limitations, the current research sets the stage for future exploration in broader contexts, over extended periods, and through varied methodologies. However, acknowledging these limitations enriches the study, positioning it as a contribution that offers foundational insights into apprentices' debugging practices and lays down pathways for comprehensive future research, echoing the call by Grant and Booth (2009) for robust and adaptive qualitative research methodologies.

6.9 Further Research and Recommendations

In addressing the current study's limitations, a critical analysis suggests that future research should adopt a more encompassing approach, integrating a broader and more diverse participant base across different industries and cultural backgrounds. This expansion is crucial for enhancing the generalisability of the findings, providing a more

representative understanding of debugging practices in varied organisational and cultural contexts. Incorporating a mixed-method approach could offer a crucial balance, allowing researchers to delve deeper while simultaneously capturing a broader perspective (Creswell & Clark, 2007; Tufford & Newman, 2012). Simultaneously, it is imperative to uphold methodological rigour and actively mitigate biases, particularly in qualitative research. This involves enhancing objectivity in thematic interpretation and being vigilant of the researcher's influence on the analysis. Such an approach would address potential confirmation biases, ensuring that validation processes, while robust, do not inadvertently reinforce skewed perspectives or pre-existing beliefs of participants (Hsieh & Shannon, 2005; Nickerson, 1998).

Moreover, considering the rapid evolution of software development tools and practices, future studies must adapt to include emerging technologies, languages, and frameworks. This adaptation is essential to ensure that research findings remain relevant and applicable within the fast-paced technological landscape of software development (Rajlich & Bennett, 2000). In tandem with this technological adaptability, an exploration into the impact of digital communication platforms like Microsoft Teams on research processes is warranted. As these platforms increasingly become integral to remote collaboration in research, understanding their effects on participant interaction, data collection, and analysis could yield vital insights. This investigation could unveil the dynamics of remote collaboration in research settings, highlighting how digital communication affects the intricacies of data gathering and participant interactions.

Through such a comprehensive and adaptable approach, future research in software development and apprenticeships can build on the foundational insights of the current study. It can provide richer, more deeper insights into debugging practices and apprenticeship learning, accounting for the evolving challenges and opportunities in the dynamic domain of technology-driven environments. This approach highlights the importance of continuous research adaptation and innovation in response to changing technological and methodological landscapes.

6.10 Conclusion

This thesis marks an advancement in the field of computing education, bringing to light the intricacies of work-based learning, specifically in the realm of software development apprenticeship and their debugging practices. Central to the thesis is exploring how apprentices navigate the intricate balance of cognitive, technical, and communicative aspects within debugging tasks. The study delves into the strategies apprentices employ to address complex syntax, logical and runtime errors and their use of IDEs like Microsoft Visual Studio, offering vital insights into their problem-solving processes and technical proficiency.

The research emphasises the critical role of communication and collaboration in debugging, especially within remote learning environments. The challenges posed by geographical dispersion highlight the need for innovative educational strategies and tools that effectively bridge communication gaps in remote learning scenarios. The research also calls attention to the necessity of keeping pace with the rapidly evolving field of

software development, urging continuous adaptation in teaching methodologies to align with technological advancements.

The limitations identified in the study, such as generalisability concerns and potential researcher bias, pave the way for future research opportunities. Exploring a broader range of contexts and employing diverse methodologies can enhance the scope of understanding in debugging practices within different environments. This approach would build upon the findings of this thesis and contribute to the broader body of knowledge in computing education.

In conclusion, this thesis stands as a critical contribution to computing education, providing insights into apprentices' debugging practices. It informs and has the potential to transform educational practices and tool development, fostering the growth of skilled professionals in the constantly evolving field of software development. The research establishes a foundational understanding for further investigation, demonstrating qualitative research's dynamic and impactful nature in technology education.

References

Adeliyi, A., Wermelinger, M., Kear, K., & Rosewell, J. (2021). *Investigating Remote Pair Programming In Part-Time Distance Education* 3rd Conference on United Kingdom and Ireland Computing Education Research, UKICER 2021, Online. <https://oro.open.ac.uk/79055/>

Adelson, B., & Soloway, E. (1985). The Role of Domain Experience in Software Design. *IEEE Transactions on Software Engineering*, SE-11(11), 1351-1360. <https://doi.org/10.1109/TSE.1985.231883>

Afzal, A., & Goues, C. L. (2018). *A study on the use of IDE features for debugging*. MSR '18: Proceedings of the 15th International Conference on Mining Software Repositories, Gothenburg, Sweden.

Agerfalk, P. J., Fitzgerald, B., Holmstrom, H., Lings, B., Lundell, B., & Conchui, E. Ó. (2005). *A framework for considering opportunities and threats in distributed software development*. Proceedings of the International Workshop on Distributed Software Development: DiSD 2005, Paris, France.

Agrawal, H., DeMillo, R. A., & Spafford, E. H. (1993). Debugging with dynamic slicing and backtracking. *Software: Practice and Experience*, 23(6), 589-616.

Ahadi, A., Lister, R., Lal, S., & Hellas, A. (2018). *Learning programming, syntax errors and institution-specific factors* ACE '18: Proceedings of the 20th Australasian Computing Education Conference, Brisbane, Queensland, Australia.

Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. *ACM SIGCSE Bulletin*, 37(3), 84-88.

Ahn, J., Sung, W., & Black, J. B. (2022). Unplugged debugging activities for developing young learners' debugging skills. *Journal of Research in Childhood Education*, 36(3), 421-437.

Akinola, S. (2014). An Empirical Comparative Analysis of Programming Effort, Bugs Incurrence and Code Quality between Solo & Pair Programmers. *Middle-East Journal of Scientific Research*, 21(12), 2231-2237.

Alaboudi, A., & LaToza, T. D. (2023). What constitutes debugging? An exploratory study of debugging episodes. *Empirical Software Engineering*, 28(5), 117. <https://doi.org/https://doi.org/10.48550/arXiv.2105.02162>

Alexander, P. A. (2003). The development of expertise: The journey from acclimation to proficiency. *Educational researcher*, 32(8), 10-14.

Allwood, C. M. (1986). Novices on the computer: a review of the literature. *International Journal of Man-Machine Studies*, 25(6), 633-658.

Allwood, C. M., & Bjorhag, C.-G. (1990). Novices' debugging when programming in Pascal. *International Journal of Man-Machine Studies*, 33(6), 707-724.

Alqadi, B. S., & Maletic, J. I. (2017). *An Empirical Study of Debugging Patterns Among Novices Programmers* Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, Washington, USA.

Anderson, J. R. (2015). *Cognitive Psychology and Its Implications* (8th ed.). Worth Publishers.

Arksey, H., & O'Malley, L. (2005). Scoping studies: towards a methodological framework. *International Journal of Social Research Methodology*, 8(1), 19-32.

Artman, H., & Wærn, Y. (1999). Distributed cognition in an emergency co-ordination center. *Cognition, Technology & Work*, 1(4), 237-246.

Bacchelli, A., & Bird, C. (2013, 18-26 May 2013). *Expectations, outcomes, and challenges of modern code review* 2013 35th International Conference on Software Engineering (ICSE), San Francisco, CA, USA.

Baheti, P., Gehringer, E., & Stotts, D. (2002). Exploring the Efficacy of Distributed Pair Programming. In D. Wells & L. Williams, *Extreme Programming and Agile Methods — XP/Agile Universe 2002 Conference on Extreme Programming and Agile Methods*, Berlin, Heidelberg.

Baker, R. S., Corbett, A. T., Koedinger, K. R., & Wagner, A. Z. (2004). *Off-task behavior in the cognitive tutor classroom: when students "game the system"* Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Vienna, Austria. <https://doi.org/10.1145/985692.985741>

Barbosa Rocha, H. J., Tedesco, P. C. D. A. R., & Costa, E. D. B. (2022). On the use of feedback in learning computer programming by novices: a systematic literature mapping. *Informatics in Education*, 22(2), 209–232. <https://doi.org/10.15388/infedu.2023.09>

Baxter, P., & Jack, S. (2008). Qualitative case study methodology: Study design and implementation for novice researchers. *The Qualitative Report*, 13(4), 544-559.

Beasley, Z. J., & Johnson, A. R. (2022). *The Impact of Remote Pair Programming in an Upper-Level CS Course*. ITiCSE 2022: Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education, Dublin, Ireland.

Beck, K. (2000). *Extreme programming explained: embrace change*. addison-wesley professional.

Becker, B. A., Denny, P., Pettit, R., Bouchard, D., Bouvier, D. J., Harrington, B., Kamil, A., Karkare, A., McDonald, C., & Osera, P.-M. (2019). Compiler error messages considered unhelpful: The landscape of text-based programming error message research. *Proceedings of the working group reports on innovation and technology in computer science education*, 177-210.

Becker, H. S. (1971). Sociological Work: Method and substance, Allen Lane. In: The Penguin Press.

Beckwith, L., Kissinger, C., Burnett, M., Wiedenbeck, S., Lawrance, J., Blackwell, A., & Cook, C. (2006). *Tinkering and gender in end-user programmers' debugging* Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Montréal, Québec, Canada. <https://doi.org/10.1145/1124772.1124808>

Beller, M., Spruit, N., Spinellis, D., & Zaidman, A. (2018). *On the dichotomy of debugging behavior among programmers* Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden. <https://doi.org/10.1145/3180155.3180175>

Beller, M., Spruit, N., & Zaidman, A. (2017). How developers debug. *PeerJ Preprints*, 5, e2743v2741. <https://doi.org/10.7287/peerj.preprints.2743v1>

Bennedsen, J., & Caspersen, M. E. (2007). Assessing process and product: a practical lab exam for an introductory programming course. *Innovation in Teaching and Learning in Information and Computer Sciences*, 6(4), 183-202.

Bipp, T., Lepper, A., & Schmedding, D. (2008). Pair programming in software development teams—An empirical study of its benefits. *Information and software technology*, 50(3), 231-240.

Blackwell, A., Robinson, P., Roast, C., & Green, T. (2002). *Cognitive models of programming-like activity* CHI'02 Extended Abstracts on Human Factors in Computing Systems, Minneapolis, Minnesota, USA.

Bogdan, R., & Biklen, S. (2007). *Qualitative Research for Education: An Introduction to Theory and Methods*. (5 ed.). Allyn & Bacon, Boston.

Bonar, J., & Soloway, E. (1983). *Uncovering principles of novice programming* Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Austin, Texas. <https://doi.org/10.1145/567067.567069>

Brannen, J. (2005). Mixing methods: The entry of qualitative and quantitative approaches into the research process. *International Journal of Social Research Methodology*, 8(3), 173-184.

Bransford, J. D., Brown, A. L., & Cocking, R. R. (2000). *How People Learn: Brain, Mind, Experience, and School*. National Academy Press.

Bransford, J. D., & Schwartz, D. L. (1999). Chapter 3: Rethinking transfer: A simple proposal with multiple implications. *Review of research in education*, 24(1), 61-100.

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2), 77-101.

Britten, N. (1995). Qualitative research: qualitative interviews in medical research. *Bmj*, 311(6999), 251-253.

Broome, M. E. (2000). Integrative literature reviews for the development of concepts. *Concept development in nursing: foundations, techniques and applications*. Philadelphia (USA): WB Saunders Company, 231-250.

Bruner, J. S. (2009). *The Process of Education*. Harvard University Press, Cambridge.

Bryant, S., Romero, P., & du Boulay, B. (2008). Pair programming and the mysterious role of the navigator. *International Journal of Human-Computer Studies*, 66(7), 519-529.

Bryman, A. (2003). *Quantity and Quality in Social Research* (Vol. 18). Routledge. <https://doi.org/https://doi.org/10.4324/9780203410028>

Bryman, A. (2016). *Social Research Methods* (5th ed.). Oxford University Press.

Cameron, R. (2011). Mixed methods research: The five Ps framework. *Electronic journal of business research methods*, 9(2), pp96-108-pp196-108.

Card, S. K., Moran, T. P., & Newell, A. (2018). *The Psychology of Human-Computer Interaction*. CRC Press.

Carnwell, R., & Daly, W. (2001). Strategies for the construction of a critical review of the literature. *Nurse education in practice*, 1(2), 57-63.

Carter, J. (2015). The Apprenticeship Agenda. *Impact Magazine*, 2-3.

Carter, N., Bryant-Lukosius, D., DiCenso, A., Blythe, J., & Neville, A. J. (2014). The Use of Triangulation in Qualitative Research. *Oncology Nursing Forum*, 41(5), 545-547.

Castillo-Montoya, M. (2016). Preparing for Interview Research: The Interview Protocol Refinement Framework. *The Qualitative Report*, 21(5), 811-831. <https://doi.org/10.46743/2160-3715/2016.2337>

Chalmers, P. A. (2003). The role of cognitive theory in human-computer interface. *Computers in Human Behavior*, 19(5), 593-607. [https://doi.org/10.1016/S0747-5632\(02\)00086-9](https://doi.org/10.1016/S0747-5632(02)00086-9)

Chen, M.-W., Wu, C.-C., & Lin, Y.-T. (2013). Novices' debugging behaviors in VB programming. Learning and Teaching in Computing and Engineering (LaTiCE), 2013,

Cheng, L.-T., de Souza, C. R., Hupfer, S., Patterson, J., & Ross, S. (2003). Building Collaboration into IDEs: Edit> Compile> Run> Debug> Collaborate? *Queue*, 1(9), 40-50.

Cherenkova, Y., Zingaro, D., & Petersen, A. (2014). Identifying challenging CS1 concepts in a large problem dataset. Proceedings of the 45th ACM technical symposium on Computer science education,

Chi, E. H., Pirolli, P., Chen, K., & Pitkow, J. (2001). Using information scent to model user information needs and actions and the Web. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems,

Chintakovid, T., Wiedenbeck, S., Burnett, M., & Grigoreanu, V. (2006). *Pair Collaboration in End-User Debugging*. Proceedings - IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2006, Brighton, UK.

Chong, J., & Hurlbutt, T. (2007). *The Social Dynamics of Pair Programming*. ICSE '07: Proceedings of the 29th international conference on Software Engineering, <https://doi.org/10.1109/ICSE.2007.87>

Chorfi, A., Hedjazi, D., Aouag, S., & Boubiche, D. (2020). Problem-based collaborative learning groupware to improve computer programming skills. *Behaviour & information technology*, 1-20.

Christensen, L. B., Johnson, B., & Turner, L. A. (2020). *Research methods, design, and analysis* (Thirteenth Edition ed.). Pearson Education, Inc.

Clarke, S. O., Ilgen, J. S., & Regehr, G. (2023). Fostering Adaptive Expertise Through Simulation. *Academic Medicine*, 98(9), 994-1001. <https://doi.org/10.1097/ACM.00000000000005257>

Cockburn, A., & Williams, L. (2000). The costs and benefits of pair programming. *Extreme programming examined*, 8, 223-247.

Cohen, L., Manion, L., & Morrison, K. (2007). Research methods in education. In: London: Routledge.

Coker, Z., Widder, D. G., Le Goues, C., Bogart, C., & Sunshine, J. (2019). *A qualitative study on framework debugging*. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), <https://doi.org/10.1109/ICSME.2019.00091>.

Colquitt, J. A. (2013). Crafting References in AMJ Submissions. In (Vol. 56, pp. 1221-1224): Academy of Management Journal.

Cooke, A., Smith, D., & Booth, A. (2012). Beyond PICO: the SPIDER tool for qualitative evidence synthesis. *Qualitative Health Research*, 22(10), 1435-1443.

Creamer, E. G. (2017). *An introduction to fully integrated mixed methods research*. sage publications.

Cresswell, J., & Plano Clark, V. L. (2011). Designing and conducting mixed method research. *Thousand Oaks, CA*.

Creswell, J. W. (2014). *Research design: Qualitative, quantitative, and mixed methods approaches* (4th ed. ed.). SAGE Publications.

Creswell, J. W., & Clark, V. L. P. (2007). Designing and conducting mixed methods research.

Creswell, J. W., & Poth, C. N. (2018). *Qualitative inquiry and research design: Choosing among five approaches* (Fourth ed.). Sage publications.

Crotty, M. (1998). *The foundations of social research: Meaning and perspective in the research process*. Sage.

da Silva Estacio, B. J., & Prikladnicki, R. (2015). Distributed pair programming: A systematic literature review. *Information and software technology*, 63, 1-10.

David, J. A. (2002). Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems. *American Management Association (AMACOM)*, 5.

de Raadt, M. (2007). A review of Australasian investigations into problem solving and the novice programmer. *Computer Science Education*, 17(3), 201-213.

Denny, P., Becker, B. A., Bosch, N., Prather, J., Reeves, B., & Whalley, J. (2022). Novice Reflections During the Transition to a New Programming Language.

Dewey, J. (1922). The middle works of John Dewey: Human nature and conduct (Vol. 14). In: Carbondale: Southern Illinois University Press.

Dey, A. K. (2001). Understanding and using context. *Personal and Ubiquitous Computing*, 5(1), 4-7.

DfE/BIS. (2013). *The Future of Apprenticeship in England: Next Steps from the Richard Review*. Department for Education and Department for Business, Innovation and Skills ...

Dillenbourg, P., Järvelä, S., & Fischer, F. (2009). The Evolution of Research on Computer-Supported Collaborative Learning. In N. Balacheff, S. Ludvigsen, T. de Jong, A. Lazonder, & S. Barnes (Eds.), *Technology-Enhanced Learning: Principles and Products* (pp. 3-19). Springer, Dordrecht. https://doi.org/10.1007/978-1-4020-9827-7_1

Downey, A. B. (2012). *Think Python: How to think like a computer scientist*. Green Tea Press.

Dreyfus, H., Dreyfus, S. E., & Athanasiou, T. (2000). *MIND OVER MACHINE: The Power of Human Intuition and Expertise in the Era of the Computer*. Simon and Schuster.

Dreyfus, H. L., & Dreyfus, S. E. (2005). Peripheral vision: Expertise in real world contexts. *Organization studies*, 26(5), 779-792.

Du Boulay, B. (1986). Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1), 57-73.

Du Preez Ockert, J. (2019). *Visual studio 2019 in depth*. BPb Publications.

Dyba, T., & Dingsoyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information and software technology*, 50(9-10), 833-859.

Eisenstadt, M. (1993). *Tales of Debugging From the Front Lines*. Empirical Studies of Programmers: Fifth Workshop, Palo Alto, California.

Endsley, M. R. (1995). Toward a theory of situation awareness in dynamic systems. *Human factors*, 37(1), 32-64.

Engeström, Y., Miettinen, R., Punamäki, R.-L., Minnis, M., & John-Steiner, V. P. (2001). Perspectives on activity theory. *Human development*, 44(5), 296-310.

Ericsson, K. A. (2006). Protocol analysis and expert thought: Concurrent verbalizations of thinking during experts' performance on representative tasks. *The Cambridge handbook of expertise and expert performance*, 223-241.

Ericsson, K. A., & Simon, H. A. (1984). Protocol analysis: Verbal reports as data. A Bradford book. In: The MIT Press: Cambridge, Massachusetts, London, England.

Ericsson, K. A., & Simon, H. A. (1993). *Protocol analysis*. MIT press Cambridge, MA.

Espinosa, J. A., Slaughter, S. A., Kraut, R. E., & Herbsleb, J. D. (2007). Team knowledge and coordination in geographically distributed software development. *Journal of management information systems*, 24(1), 135-169.

Ettles, A., Luxton-Reilly, A., & Denny, P. (2018). *Common logic errors made by novice programmers* Proceedings of the 20th Australasian Computing Education Conference, Brisbane, Queensland, Australia. <https://doi.org/10.1145/3160489.3160493>

Faja, S. (2014). Evaluating effectiveness of pair programming as a teaching tool in programming courses. *Information Systems Education Journal*, 12(6), 36.

Finlay, L. (2002). "Outing" the researcher: The provenance, process, and practice of reflexivity. *Qualitative Health Research*, 12(4), 531-545.

Fitzgerald, B., & Howcroft, D. (1998). Competing dichotomies in IS research and possible strategies for resolution. In *Proceedings of the International Conference on Information Systems (ICIS '98)*, 14, 155-164.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, 18(2), 93-116.

Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., & Zander, C. (2010). Debugging from the student perspective. *IEEE Transactions on Education*, 53(3), 390-396.

Fitzpatrick, B. W., & Collins-Sussman, B. (2015). *Debugging Teams: Better Productivity Through Collaboration.* "O'Reilly Media, Inc."

Flavell, J. H. (1979). Metacognition and cognitive monitoring: A new area of cognitive–developmental inquiry. *American Psychologist*, 34(10), 906.

Fleming, S. D., Scaffidi, C., Piorkowski, D., Burnett, M., Bellamy, R., Lawrence, J., & Kwan, I. (2013). An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(2), 14.

Flick, U. (2022). An introduction to qualitative research. *An introduction to qualitative research*, 1-100.

Flor, N. V., & Hutchins, E. L. (1991). *Analysing Distributed Cognition in Software Teams: A Case Study of Team Programming during Adaptive Software Maintenance* Empirical studies of programmers: Fourth workshop, New Brunswick, N.J.

Flyvbjerg, B. (2006). Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2), 219-245.

Fontana, E. A., & Petrillo, F. (2021). *Mapping breakpoint types: an exploratory study*. 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), Hainan Island, China.

Freudenberg, S., Romero, P., & du Boulay, B. (2007). "Talking the talk": Is intermediate-level conversation the key to the pair programming success story? Agile 2007 (AGILE 2007), Washington, DC, USA.

Fromherz, A., Ouadjaout, A., & Miné, A. (2018). *Static value analysis of Python programs by abstract interpretation* NASA Formal Methods: 10th International Symposium, NFM 2018,, Newport News, VA, US.

Fuller, A., & Unwin, L. (2010). Creating and Supporting Expansive Apprenticeships: a guide for employers, training providers and colleges of further education. In: London, LSIS [Online]. Available at <http://webarchive.nationalarchives.gov>

Fuller, A., & Unwin, L. (2013). Apprenticeship and the concept of occupation. *The Gatsby Charitable Foundation, London*.

Geertz, C. (1973). The interpretation of cultures. NY: Basic Books.

Glesne, C. (2016). *Becoming qualitative researchers: An introduction*. ERIC.

Glezou, K., & Grigoriadou, M. (2010). Engaging students of senior high school in simulation development. *Informatics in Education*, 9(1), 37-62.

Goldman, M., Little, G., & Miller, R. C. (2011). *Real-time collaborative coding in a web IDE* UIST '11: Proceedings of the 24th annual ACM symposium on User interface software and technology, Santa Barbara, California, USA. <https://doi.org/10.1145/2047196.2047215>

Gomes, A., & Mendes, A. J. (2007). *Learning to program-difficulties and solutions* International Conference on Engineering Education–ICEE, Coimbra, Portugal.

Gould, J. D. (1975). Some psychological evidence on how people debug computer programs. *International Journal of Man-Machine Studies*, 7(2), 151-182.

Gould, J. D., & Drongowski, P. (1974). An exploratory study of computer program debugging. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 16(3), 258-277.

Grandell, L., Peltomäki, M., & Salakoski, T. (2005). High school programming—a beyond-syntax analysis of novice programmers' difficulties. Proceedings of the Koli Calling 2005 Conference on Computer Science Education,

Grant, M. J., & Booth, A. (2009). A typology of reviews: an analysis of 14 review types and associated methodologies. *Health information & libraries journal*, 26(2), 91-108.

Gray, D. E. (2021). *Doing research in the real world* (5th ed.). SAGE Publications Ltd.

Greenbaum, T. L. (1998). *The handbook for focus group research*. Sage.

Greenhalgh, T., & Peacock, R. (2005). Effectiveness and efficiency of search methods in systematic reviews of complex evidence: audit of primary sources. *Bmj*, 331(7524), 1064-1065.

Grigoreanu, V., Burnett, M., Wiedenbeck, S., Cao, J., Rector, K., & Kwan, I. (2012). End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 19(1), 1-28.

Grix, J. (2004). The Foundation of Research, Great Britain. In: Plagave Macmillan Press.

Grover, S., & Pea, R. (2013). Computational thinking in K-12: A review of the state of the field. *Educational researcher*, 42(1), 38-43.

Grover, S., Pea, R., & Cooper, S. (2014). Promoting active learning & leveraging dashboards for curriculum assessment in an OpenEdX introductory CS course for middle school. Proceedings of the first ACM conference on Learning@ scale conference,

Guba, E. G., & Lincoln, Y. S. (1994). Competing paradigms in qualitative research. *Handbook of qualitative research*, 2(163-194), 105.

Gugerty, L., & Olson, G. (1986). Debugging by skilled and novice programmers. *ACM SIGCHI Bulletin*, 17(4), 171-174.

Guile, D., & Young, M. (1998). Apprenticeship as a conceptual basis for a social theory of learning. *Journal of Vocational Education & Training*, 50(2), 173-193.

Guzdial, M. (1994). Software-Realized Scaffolding to Facilitate Programming for Science Learning. *Interactive Learning Environments*, 4(1), 001--044. <https://doi.org/10.1080/1049482940040101>

Guzdial, M. (2015). *Learner-centered design of computing education: Research on computing for everyone*. Morgan & Claypool Publishers.

Guzdial, M. J., & Ericson, B. (2013). *Introduction to Computing and Programming in Python: International Edition*. Pearson Higher Ed.

Hafeez, M., Karki, A., Radwan, Y., Saha, A., & Zavaleta Bernuy, A. (2023). *Evaluating the Efficacy and Impacts of Remote Pair Programming for Introductory Computer Science Students* Proceedings of the 25th Western Canadian Conference on Computing Education, Vancouver, BC, Canada.

Hammersley, M., & Atkinson, P. (1995). *Ethnography: Principles in practice* (2nd ed. ed.). Routledge.

Hanks, B. (2008). Empirical evaluation of distributed pair programming. *International Journal of Human-Computer Studies*, 66(7), 530-544.

Hanks, B., Fitzgerald, S., McCauley, R., Murphy, L., & Zander, C. (2011). Pair programming in education: a literature review. *Computer Science Education*, 21(2), 135-173.

Hanks, B., McDowell, C., Draper, D., & Krnjajic, M. (2004). Program quality with pair programming in CS1. Proceedings of the 9th annual SIGCSE conference on Innovation and technology in computer science education,

Hannay, J. E., Dyba, T., Arisholm, E., & Sjoberg, D. I. (2009). The effectiveness of pair programming: A meta-analysis. *Information and software technology*, 51(7), 1110-1122.

Hart, C. (1998). Doing a literature review: Releasing the social science research imagination (SAGE) Reviewing the literature for a research project can seem a

daunting, even overwhelming task. *New researchers, in particular, wonder: Where do I start*, 30.

Hart, C. (2018). Doing a literature review: Releasing the research imagination. *SAGE Study Skills Series*, 352.

Hassan, M., & Zilles, C. (2022). On Students' Ability to Resolve their own Tracing Errors through Code Execution. SIGCSE 2022: Proceedings of the 53rd ACM Technical Symposium on Computer Science Education V. 1, March 3–5, 2022, Providence RI USA.

Hatano, G., & Inagaki, K. (1986). Two courses of expertise. In *Child development and education in Japan*. (pp. 262-272). W H Freeman/Times Books/ Henry Holt & Co.

Hazzan, O., Ragonis, N., Lapidot, T., Hazzan, O., Ragonis, N., & Lapidot, T. (2020). Problem-solving strategies. *Guide to Teaching Computer Science: An Activity-Based Approach*, 143-168.

Helminen, J., Ihantola, P., & Karavirta, V. (2013). Recording and analyzing in-browser programming sessions. Proceedings of the 13th Koli Calling International Conference on Computing Education Research,

Herbsleb, J. D., & Moitra, D. (2001). Global software development. *IEEE software*, 18(2), 16-20.

Heyes, J. (2013). Vocational training, employability and the post-2008 jobs crisis: Responses in the European Union. *Economic and industrial democracy*, 34(2), 291-311.

Hirsch, T., & Hofer, B. (2022). Using textual bug reports to predict the fault category of software bugs. *Array*, 100189.

Hmelo-Silver, C. E. (2004). Problem-based learning: What and how do students learn? *Educational Psychology Review*, 16, 235-266.

Hoeckel, K., & Schwartz, R. (2010). Learning for Jobs OECD Reviews of Vocational Education and Training. *Austria: Organisation for Economic Co-operation and Development (OECD)*.

Hollan, J., Hutchins, E., & Kirsh, D. (2000). Distributed cognition: toward a new foundation for human-computer interaction research. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(2), 174-196.

Hooper, E. J., & Thomas, R. A. (1990). Investigating the effects of a manipulative model of computer memory operations on the learning of programming. *Journal of Research on Computing in Education*, 22(4), 442-456.

Hopia, H., Latvala, E., & Liimatainen, L. (2016). Reviewing the methodology of an integrative review. *Scandinavian journal of caring sciences*, 30(4), 662-669.

Horwitz, S., Liblit, B., & Polishchuk, M. (2009). Better debugging via output tracing and callstack-sensitive slicing. *IEEE Transactions on Software Engineering*, 36(1), 7-19.

Hsieh, H.-F., & Shannon, S. E. (2005). Three approaches to qualitative content analysis. *Qualitative Health Research*, 15(9), 1277-1288.

Hughes, J., Walshe, A., Law, B., & Murphy, B. (2020). *Remote pair programming* 12th International Conference on Computer Supported Education - Volume 2: CSEDU, <https://doi.org/10.5220/0009582904760483>

Hutchins, E. (1995). *Cognition in the Wild*. MIT press.

IfATE. (2022). *Software development technician*. Retrieved April 5 from <https://www.instituteforapprenticeships.org/apprenticeship-standards/software-development-technician-v1-1>

IfATE. (2023). *Apprenticeship Standards*. Institute for Apprenticeships and Technical Education. <https://www.instituteforapprenticeships.org/apprenticeship-standards/?routes=digital&includeApprovedForDelivery=true>

IfATE. (2024). *Software development technician Apprentice Standard*. Institute for Apprenticeships and Technical Education. Retrieved 06/02/2024 from <https://www.instituteforapprenticeships.org/apprenticeship-standards/software-development-technician-v1-1>

Jayathirtha, G., Fields, D., & Kafai, Y. (2020). *Pair debugging of electronic textiles projects: Analyzing think-aloud protocols for high school students' strategies and practices while problem solving* The Interdisciplinarity of the Learning Sciences, 14th International Conference of the Learning Sciences (ICLS) 2020, Nashville, USA.

Jayathirtha, G., Fields, D., & Kafai, Y. (2024). Distributed debugging with electronic textiles: understanding high school student pairs' problem-solving strategies, practices, and perspectives on repairing physical computing projects. *Computer Science Education*, 1-35.

Jeffries, B., Lee, J. A., & Koprinska, I. (2022, July 8–13, 2022). 115 Ways Not to Say Hello, World! Syntax Errors Observed in a Large-Scale Online CS0 Python Course. Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1 (ITiCSE 2022), Dublin, Ireland.

Jeffries, R. (1982). *A comparison of the debugging behavior of expert and novice programmers* Proceedings of AERA annual meeting, New York, NY, USA. <https://doi.org/10.3102/0013189X011008022>

Jenkins, T. (2002). *On the difficulty of learning to program*. Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences, Loughborough, UK.

Jesson, J., Matheson, L., & Lacey, F. M. (2011). Doing your literature review: Traditional and systematic techniques. *Doing Your Literature Review*, 1-192.

Joanna Briggs Institute. (2017). *Checklist for systematic reviews and research syntheses*. https://joannabriggs.org/ebp/critical_appraisal_tools

Johnson, D. W., & Johnson, R. T. (1987). *Learning together and alone: Cooperative, competitive, and individualistic learning*. Prentice-Hall, Inc.

Johnson, D. W., & Johnson, R. T. (1999). Making cooperative learning work. *Theory into practice*, 38(2), 67-73.

Johnson, E. A. J. (1937). Predecessors of Adam Smith: The Growth of British Economic Thought. *Journal of the Royal Statistical Society*, 100(4), 678-680. <https://doi.org/10.2307/2980407>

Jones, S. R., Torres, V., & Arminio, J. (2013). *Negotiating the complexities of qualitative research in higher education: Fundamental elements and issues*. Routledge.

Jordan, B., & Henderson, A. (1995). Interaction analysis: Foundations and practice. *The journal of the learning sciences*, 4(1), 39-103.

Júnior, A. S., de Figueiredo, J. C. A., & Serey, D. (2019). Analysing the Impact of Programming Mistakes on Students' Programming Abilities. Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação-SBIE), Brazil.

Karvelas, I. (2019). *Investigating Novice Programmers' Interaction with Programming Environments* Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, Scotland, Uk. <https://doi.org/10.1145/3304221.3325596>

Katz, I. R., & Anderson, J. R. (1987). Debugging: An analysis of bug-location strategies. *Human-Computer Interaction, 3*(4), 351-399.

Kavitha, R., & Ahmed, M. I. (2015). Knowledge sharing through pair programming in learning environments: An empirical study. *Education and Information Technologies, 20*(2), 319-333.

Kendall, M., Murray, S. A., Carduff, E., Worth, A., Harris, F., Lloyd, A., Cavers, D., Grant, L., Boyd, K., & Sheikh, A. (2009). Use of multiperspective qualitative interviews to understand patients' and carers' beliefs, experiences, and needs. *Bmj, 339*, b4122.

Khalid, M. A. B., Farooq, A., & Mahmood, W. (2021). Communication Challenges for Distributed Teams. *International Journal of Engineering and Manufacturing (IJEM), 11*(1), 19-28.

Khan, M., & Manderson, L. (1992). Focus groups in tropical diseases research. *Health policy and planning, 7*(1), 56-66.

Kim, C., Vasconcelos, L., Belland, B. R., Umutlu, D., & Gleasman, C. (2022). Debugging behaviors of early childhood teacher candidates with or without scaffolding. *International Journal of Educational Technology in Higher Education, 19*(1), 1-26.

Kiron, D., Kane, G. C., Palmer, D., Phillips, A. N., & Buckley, N. (2016). Aligning the organization for its digital future. *MIT Sloan Management Review, 58*(1).

Kirschner, P. A., Sweller, J., & Clark, R. E. (2006). Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist, 41*(2), 75-86.

Klahr, D., & Carver, S. M. (1988). Cognitive objectives in a LOGO debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology, 20*(3), 362-404.

Ko, A. J., & Myers, B. A. (2005). A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages & Computing, 16*(1), 41-84.

Kohn, T. (2019). *The Error Behind The Message: Finding the Cause of Error Messages in Python* Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA. <https://doi.org/10.1145/3287324.3287381>

Kohn, T., & Manaris, B. (2020). Tell Me What's Wrong: A Python IDE with Error Messages. Proceedings of the 51st ACM Technical Symposium on Computer Science Education,

Kolb, D. (1984). Experiential Learning: Experience As The Source of Learning and Development. New Jersey: Prentice Hall, Inc, Englewood Cliffs.

Kölling, M., Brown, N. C., Hamza, H., & McCall, D. (2019). *Stride in BlueJ--computing for all in an educational IDE*. SIGCSE 2019 - Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, United States.

Kraus, S., Breier, M., Lim, W. M., Dabić, M., Kumar, S., Kanbach, D., Mukherjee, D., Corvello, V., Piñeiro-Chousa, J., & Liguori, E. (2022). Literature reviews as independent studies: guidelines for academic practice. *Review of Managerial Science, 16*(8), 2577-2595.

Kuhn, T. (1970). The structure of scientific revolutions 2nd edition. *The University of Chicago press, Chicago*.

Kumar, V., Winne, P., Hadwin, A., Nesbit, J., Jamieson-Noel, D., Calvert, T., & Samin, B. (2005). *Effects of self-regulated learning in programming* Fifth IEEE

International Conference on Advanced Learning Technologies (ICALT'05), Kaohsiung, Taiwan.

Kurniawan, A., Soesanto, C., & Wijaya, J. E. C. (2015). Coder: Real-time code editor application for collaborative programming. *Procedia Computer Science*, 59, 510-519.

Kurniawan, O., Lee, N. T. S., Sockalingam, N., & Pey, K. L. (2019). Game-Based versus gamified learning platform in helping university students learn programming. *ASCILITE Publications*, 159-168.

Kvale, S. (1996). Interviews: An introduction to qualitative research interviewing. In. Thousand Oaks: Ca: Sage.

Lacave, C., & Molina, A. I. (2021). The Impact of COVID-19 in Collaborative Programming. Understanding the Needs of Undergraduate Computer Science Students. *Electronics*, 10(14), 1728.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005). A study of the difficulties of novice programmers. *ACM SIGCSE Bulletin*, 37(3), 14-18.

LaToza, T. D., Arab, M., Loksa, D., & Ko, A. J. (2020). Explicit programming strategies. *Empirical Software Engineering*, 25, 2416-2449.

LaToza, T. D., & Myers, B. A. (2010). *Developers ask reachability questions* Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering- Volume 1, Cape Town, South Africa. <https://doi.org/10.1145/1806799.1806829>

Lau, W. W., & Yuen, A. H. (2009). Toward a framework of programming pedagogy. In *Encyclopedia of Information Science and Technology, Second Edition* (pp. 3772-3777). IGI Global.

Lave, J. (1995). *Teaching as learning in practice, Sylvia Scribner Award Lecture*, San Francisco American Educational Research Association, Annual Meeting, San Francisco, CA, USA.

Lave, J. (1996). Teaching, as learning, in practice. *Mind, Culture, and Activity*, 3(3), 149-164.

Lave, J., & Wenger, E. (1991). *Situated learning: Legitimate peripheral participation*. Cambridge university press.

Lawrance, J., Bellamy, R., Burnett, M., & Rector, K. (2008). *Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks*. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Florence, Italy. <https://doi.org/10.1145/1357054.1357261>

Layman, L., Diep, M., Nagappan, M., Singer, J., Deline, R., & Venolia, G. (2013). Debugging revisited: Toward understanding the debugging needs of contemporary software developers. 2013 ACM/IEEE international symposium on empirical software engineering and measurement, Baltimore, Maryland, USA.

Leavy, P. (2017). *Research design: Quantitative, qualitative, mixed methods, arts-based, and community-based participatory research approaches*. (First ed.). Guilford Press. <https://doi.org/10.1111/fcsr.12276>

Lee, M. J., Bahmani, F., Kwan, I., LaFerte, J., Charters, P., Horvath, A., Luor, F., Cao, J., Law, C., & Beswetherick, M. (2014). *Principles of a debugging-first puzzle game for computing education* 2014 IEEE symposium on visual languages and human-centric computing (VL/HCC), Melbourne, VIC, Australia.

Leedy, P. D., & Ormrod, J. E. (2021). *Practical research: Planning and design* (12th Edition ed.). Pearson Education.

Lewis, C. M., & Gregg, C. (2016). *How Do You Teach Debugging? Resources and Strategies for Better Student Debugging* SIGCSE '16: The 47th ACM Technical

Symposium on Computing Science Education, Memphis, Tennessee, USA. <https://doi.org/10.1145/2839509.2850473>

Li, X., Zhu, S., d'Amorim, M., & Orso, A. (2018). *Enlightened debugging*. Proceedings of the 40th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2018), Gothenburg, Sweden. <https://doi.org/10.1145/3180155.3180242>

Liamputpong, P. (2011). *Focus group methodology: Principle and practice*. Sage Publications.

Lincoln, Y. S., & Guba, E. G. (1985). *Naturalistic Inquiry* (Vol. 75). Sage.

Lincoln, Y. S., & Guba, E. G. (2000). The Only Generalization Is There Is No Generalization. In: Gomm, R., Hammersley, M. and Foster, P., Eds., *Case Study Method*, SAGE, London, 27-45.

Lincoln, Y. S., Lynham, S. A., & Guba, E. G. (2011). Paradigmatic controversies, contradictions, and emerging confluences, revisited. In N. K. Denzin & Y. S. Lincoln (Eds.), *The SAGE handbook of qualitative research* (4th ed., pp. 97–128). *The Sage handbook of qualitative research*, 4(2), 97-128.

Linn, M. C., & Dalbey, J. (1985). Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational psychologist*, 20(4), 191-206.

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., & Seppälä, O. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4), 119-150.

Liu, Q., & Paquette, L. (2023). *Using submission log data to investigate novice programmers' employment of debugging strategies* LAK23: 13th International Learning Analytics and Knowledge Conference, Arlington, TX, USA. <https://doi.org/10.1145/3576050.3576094>

Liu, Z., Zhi, R., Hicks, A., & Barnes, T. (2017). Understanding problem solving behavior of 6–8 graders in a debugging game. *Computer Science Education*, 27(1), 1-29.

Loeliger, J., & McCullough, M. (2012). *Version Control with Git: Powerful tools and techniques for collaborative software development*. " O'Reilly Media, Inc.".

Lowe, T. (2019). *Debugging: The key to unlocking the mind of a novice programmer?* 2019 IEEE Frontiers in Education Conference (FIE), Covington, KY, USA.

Lubbe, W., ten Ham-Baloyi, W., & Smit, K. (2020). The integrative literature review as a research method: A demonstration review of research on neurodevelopmental supportive care in preterm infants. *Journal of Neonatal Nursing*, 26(6), 308-315.

Lutz, M. (2013). *Learning python: Powerful object-oriented programming*. " O'Reilly Media, Inc.".

Luxton-Reilly, A. (2016). Learning to program is easy. ITiCSE '16: Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education, Arequipa, Peru.

Luxton-Reilly, A., & Petersen, A. (2017). *The Compound Nature of Novice Programming Assessments* Proceedings of the Nineteenth Australasian Computing Education Conference, Geelong, VIC, Australia. <https://doi.org/10.1145/3013499.3013500>

Lynch, J. W., Agarwal, J., & Imbrie, P. (2023). *Work in Progress: Engineering together - Applying remote collaborative technology to an in-person undergraduate engineering course* 2023 ASEE Annual Conference & Exposition, Baltimore , Maryland. <https://peer.asee.org/44240>

Maguire, M., & Delahunt, B. (2017). Doing a thematic analysis: A practical, step-by-step guide for learning and teaching scholars. *All Ireland Journal of Higher Education*, 9(3).

Malik, S. I., Mathew, R., Al-Sideiri, A., Jabbar, J., Al-Nuaimi, R., & Tawafak, R. M. (2022). Enhancing problem-solving skills of novice programmers in an introductory programming course. *Computer Applications in Engineering Education*, 30(1), 174-194.

Marchionini, G. (1995). *Information seeking in electronic environments*. Cambridge university press.

Martens, D. (2005). Research Methods in Education and Psychology: Integrating Diversity with Quantitative Approaches. In: Thousand Oaks: Sage.

Matloff, N. S., & Salzman, P. J. (2008). *The art of debugging with GDB, DDD, and Eclipse*. No Starch Press.

Maxwell, J. A. (2008). Designing a qualitative study. *The SAGE handbook of applied social research methods*, 2, 214-253.

Maxwell, J. A. (2012). *Qualitative research design: An interactive approach*. Sage publications.

Mayer, R. E. (1981). The psychology of how novices learn computer programming. *Acm Computing Surveys (Csur)*, 13(1), 121-141.

Mayer, R. E. (2004). Should there be a three-strikes rule against pure discovery learning? *American psychologist*, 59(1), 14.

Mayer, R. E., & Moreno, R. (2003). Nine ways to reduce cognitive load in multimedia learning. *Educational psychologist*, 38(1), 43-52.

McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: a review of the literature from an educational perspective. *Computer Science Education*, 18(2), 67-92.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B.-D., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *Working group reports from ITiCSE on Innovation and technology in computer science education* (pp. 125-180).

McDiarmid, G. W., & Zhao , Y. (2023). Time to Rethink: Educating for a Technology-Transformed World. *ECNU Review of Education*, 6(2), 189-214. <https://doi.org/10.1177/20965311221076493>

McDowell, C., Werner, L., Bullock, H. E., & Fernald, J. (2006). Pair programming improves student retention, confidence, and program quality. *Communications of the ACM*, 49(8), 90-95.

McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13(3), 307-325.

Mens, T., Cataldo, M., & Damian, D. (2019). The Social Developer: The Future of Software Development [Guest Editors' Introduction]. *IEEE software*, 36(1), 11-14.

Merriam, S., B. (2009). Qualitative research and case study appliances in education. In: San Francisco, CA: Jossey-Bass.

Merriam, S. B. (1998). *Qualitative Research and Case Study Applications in Education. Revised and Expanded from "Case Study Research in Education."*. ERIC.

Michaeli, T., & Romeike, R. (2019). *Current status and perspectives of debugging in the k12 classroom: A qualitative study* 2019 ieee global engineering education conference (educon),

Michaeli, T., & Romeike, R. (2020). *Investigating Students' Preexisting Debugging Traits: A Real World Escape Room Study* Proceedings of the 20th Koli Calling International Conference on Computing Education Research, Koli, Finland.

Miles, M. B., & Huberman, A. M. (1994). Qualitative data analysis: A sourcebook. *Beverly Hills: Sage Publications*.

Miller, B. N., Ranum, D. L., & Anderson, J. (2019). *Python programming in context*. Jones & Bartlett Learning.

Miller, C., Rodeghero, P., Storey, M.-A., Ford, D., & Zimmermann, T. (2021). "How Was Your Weekend?" *Software Development Teams Working From Home During COVID-19 2021* IEEE/ACM 43rd International Conference on Software Engineering (ICSE), Madrid, Spain. <https://doi.org/10.1109/ICSE43902.2021.00064>

Mirza-Davies, J. (2015). A short history of apprenticeships in England: From medieval craft guilds to the twenty-first century. *Retrieved*, 3(15), 2021.

Mittwede, S. K. (2012). Research Paradigms and Their Use and Importance in Theological Inquiry and Education. *Journal of Education and Christian Belief*, 16(1), 23-40. <https://doi.org/10.1177/205699711201600104>

Moher, D., Liberati, A., Tetzlaff, J., Altman, D. G., & Group, P. (2009). Preferred reporting items for systematic reviews and meta-analyses: the PRISMA statement. *Annals of internal medicine*, 151(4), 264-269.

Monat, R., Ouadjaout, A., & Miné, A. (2020). *Static type analysis by abstract interpretation of Python programs* 34th European Conference on Object-Oriented Programming (ECOOP 2020), Berlin, Germany (Virtual Conference). <https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2020.1>

Morse, J. M. (Ed.). (1994). *Designing funded qualitative research*. Sage Publications Inc.

Müller, L., Silveira, M. S., & de Souza, C. S. (2019). Source code comprehension and appropriation by novice programmers: understanding novice programmers' perception about source code reuse. *Journal on Interactive Systems*, 10(2), 96-109.

Murphy, G. C., Kersten, M., & Findlater, L. (2006). How are Java software developers using the Eclipse IDE? *IEEE software*, 23(4), 76-83.

Murphy, L., Fitzgerald, S., Hanks, B., & McCauley, R. (2010). *Pair debugging: a transactive discourse analysis* Proceedings of the Sixth international workshop on Computing education research, Aarhus, Denmark. <https://doi.org/10.1145/1839594.1839604>

Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: the good, the bad, and the quirky--a qualitative analysis of novices' strategies. *ACM SIGCSE Bulletin*, 40(1), 163-167.

Nania, J., Bonella, H., Restuccia, D., & Taska, B. (2019). No longer optional: Employer demand for digital skills. *Burning Glass Technologies*.

Nash, I., & Jones, S. (2013). Real Apprenticeships: Creating a Revolution in English Skills. Research by The Boston Consulting Group for the Sutton Trust. *Sutton Trust*.

National Research Council. (2013). *Education for life and work: Developing transferable knowledge and skills in the 21st century*. National Academies Press.

Neto, P. A. d. M. S., Mannan, U. A., de Almeida, E. S., Nagappan, N., Lo, D., Kochhar, P. S., Gao, C., & Ahmed, I. (2020). A deep dive on the impact of covid-19 in software development. *arXiv preprint arXiv:2008.07048*.

Neufeld, D. J., & Fang, Y. (2005). Individual, social and situational determinants of telecommuter productivity. *Information & management*, 42(7), 1037-1049.

Nickerson, R. S. (1998). Confirmation bias: A ubiquitous phenomenon in many guises. *Review of general psychology*, 2(2), 175-220.

Nosek, J. T. (1998). The case for collaborative programming. *Communications of the ACM*, 41(3), 105-108.

Oh, K., Almarode, J. T., & Tai, R. H. (2013). An exploration of think-aloud protocols linked with eye-gaze tracking: Are they talking about what they are looking at. *Procedia-social and behavioral sciences*, 93, 184-189.

Olson, G. M., & Olson, J. S. (2000). Distance matters. *Human-Computer Interaction*, 15(2-3), 139-178.

Oman, P. W., Cook, R., & Nanja, M. (1989). Effects of programming experience in debugging semantic errors. *Journal of Systems and software*, 9(3), 197-207.

Paas, F., Renkl, A., & Sweller, J. (2003). Cognitive load theory and instructional design: Recent developments. *Educational psychologist*, 38(1), 1-4.

Palumbo, D. B. (1990). Programming language/problem-solving research: A review of relevant issues. *Review of Educational research*, 60(1), 65-89.

Pane, J. F., & Myers, B. A. (1996). Usability issues in the design of novice programming systems.

Papadakis, S., & Orfanakis, V. (2018). Comparing novice programming environments for use in secondary education: App Inventor for Android vs. Alice. *International Journal of Technology Enhanced Learning*, 10(1-2), 44-72.

Papert, S. (1980). Mindstorms: children, computers, and powerful ideas Basic Books. Inc. New York, NY, 10, 1095592.

Park, T. H., Dorn, B., & Forte, A. (2015). An analysis of HTML and CSS syntax errors in a web development course. *ACM Transactions on Computing Education (TOCE)*, 15(1), 1-21.

Parkinson, M. M., Hermans, S., Gijbels, D., & Dinsmore, D. L. (2024). Exploring debugging processes and regulation strategies during collaborative coding tasks among elementary and secondary students. *Computer Science Education*, 1-28. <https://doi.org/10.1080/08993408.2024.2305026>

Patton, M. (2015). *Qualitative research & evaluation methods: Integrating theory and practice*. Sage publications.

Patton, M. Q. (1990). *Qualitative evaluation and research methods*. SAGE Publications, inc.

Pea, R. D. (1986). *Cognitive technologies for mathematics education*. Bank Street College of Education, Center for Children and Technology.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19(3), 295-341.

Perkins, D. N., & Martin, F. (1986). Fragile knowledge and neglected strategies in novice programmers. Papers presented at the first workshop on empirical studies of programmers,

Perscheid, M., Siegmund, B., Taeumel, M., & Hirschfeld, R. (2017). Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 25(1), 83-110.

Petrillo, F., Guéhéneuc, Y.-G., Pimenta, M., Freitas, C. D. S., & Khomh, F. (2019). Swarm debugging: The collective intelligence on interactive debugging. *Journal of Systems and software*, 153, 152-174.

Petrillo, F., Mandian, H., Yamashita, A., Khomh, F., & Guéhéneuc, Y.-G. (2017). *How do developers toggle breakpoints? observational studies* IEEE International Conference on Software Quality, Reliability and Security (QRS 2017), Prague, Czech Republic. <https://doi.org/10.1109/qrs.2017.39>

Phillips, H., Ivins, W., Prickett, T., Walters, J., & Strachan, R. (2021). Using contributing student pedagogy to enhance support for teamworking in computer science projects. In *Computing Education Practice 2021* (pp. 29-32).

Piaget, J. (1954). The construction of reality in the child (Vol. xiii). *New York: Basic Books.* <https://doi.org/10.1037/11168-11000>.

Pierre, C., & Jérémie, H. (2024). The effect of workplace vs school-based vocational education on youth unemployment: Evidence from France. *European Economic Review*, 162, 104637. <https://doi.org/https://doi.org/10.1016/j.eurocorev.2023.104637>

Piorkowski, D., Fleming, S., Scaffidi, C., Bogart, C., Burnett, M., John, B., Bellamy, R., & Swart, C. (2012). *Reactive information foraging: An empirical investigation of theory-based recommender systems for programmers* Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Austin, Texas, USA. <https://doi.org/10.1145/2207676.2208608>

Pirolli, P., & Card, S. (1999). Information foraging. *Psychological review*, 106(4), 643.

Pirolli, P., & Card, S. (2005). The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. *Proceedings of international conference on intelligence analysis*,

Plonka, L., Segal, J., Sharp, H., & Van Der Linden, J. (2011). *Collaboration in pair programming: driving and switching*. In: XP 2011: 12th International Conference on Agile Software Development, 10-13 May 2011, Madrid, Spain.

Poole, P. C. (2005). Debugging and testing. *Software Engineering: An Advanced Course*, 278-318.

Porritt, K., Gomersall, J., & Lockwood, C. (2014). Study selection and critical appraisal: the steps following the literature search in a systematic review. *Am J Nurs*, 114(6), 47-52.

Potluri, V., Pandey, M., Begel, A., Barnett, M., & Reitherman, S. (2022). *Codewalk: Facilitating shared awareness in mixed-ability collaborative software development* ASSETS '22: Proceedings of the 24th International ACM SIGACCESS Conference on Computers and Accessibility, Athens, Greece. <https://doi.org/10.1145/3517428.3544812>

Pritchard, D. (2015). *Frequency distribution of error messages*. Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools, Pittsburgh, PA, USA. <https://doi.org/10.1145/2846680.2846681>

Proksch, S., Amann, S., & Nadi, S. (2018). *Enriched event streams: a general dataset for empirical studies on in-IDE activities of software developers* Proceedings - 2018 ACM/IEEE 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden. <https://doi.org/10.1145/3196398.3196400>

Rajlich, V. T., & Bennett, K. H. (2000). A staged model for the software life cycle. *Computer*, 33(7), 66-71.

Ralph, P., Baltes, S., Adisaputri, G., Torkar, R., Kovalenko, V., Kalinowski, M., Novielli, N., Yoo, S., Devroey, X., & Tan, X. (2020). Pandemic programming. *Empirical Software Engineering*, 25(6), 4927-4961.

Ramírez Echeverry, J. J., Rosales-Castro, L. F., Restrepo-Calle, F., & González, F. A. (2018). Self-Regulated Learning in a Computer Programming Course. *IEEE Revista Iberoamericana de Tecnologias del Aprendizaje*, 13(2), 75-83. <https://doi.org/10.1109/RITA.2018.2831758>

Randolph, J. (2019). A guide to writing the dissertation literature review. *Practical assessment, research, and evaluation*, 14(1), 13.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., & Silverman, B. (2009). Scratch: programming for all. *Communications of the ACM*, 52(11), 60-67.

Richard, D. (2012). The Richard Review of Apprenticeships. London: Departement for Business. *Innovation and Skills*.

Ridder, H.-G. (2017). The theory contribution of case study research designs. *Business research*, 10, 281-305.

Rigby, L., Denny, P., & Luxton-Reilly, A. (2020). A Miss is as Good as a Mile: Off-By-One Errors and Arrays in an Introductory Programming Course. Proceedings of the Twenty-Second Australasian Computing Education Conference,

Robins, A., Haden, P., & Garner, S. (2006). Problem distributions in a CS1 course. Proceedings of the 8th Australasian Conference on Computing Education-Volume 52,

Robins, A., Rountree, J., & Rountree, N. (2003). Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2), 137-172.

Rodgers, B. L., & Cowles, K. V. (1993). The qualitative research audit trail: A complex collection of documentation. *Research in nursing & health*, 16(3), 219-226.

Rogers, Y. (1997). *A brief introduction to distributed cognition*. Interact Lab, University of Sussex.

Rogers, Y., & Ellis, J. (1994). Distributed cognition: an alternative framework for analysing and explaining collaborative working. *Journal of Information Technology*, 9(2), 119-128.

Rolfe, G. (2006). Validity, trustworthiness and rigour: quality and the idea of qualitative research. *Journal of advanced nursing*, 53(3), 304-310.

Romero, P., Du Boulay, B., Cox, R., Lutz, R., & Bryant, S. (2007). Debugging strategies and tactics in a multi-representation software environment. *International Journal of Human-Computer Studies*, 65(12), 992-1009.

Rubin, H. J., & Rubin, I. S. (2011). *Qualitative interviewing: The art of hearing data*. sage.

Russell, C. L. (2005). An overview of the integrative research review. *Progress in transplantation*, 15(1), 8-13.

Sajaniemi, J., & Kuittinen, M. (2008). From procedures to objects: A research agenda for the psychology of object-oriented programming education. *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments*.

Salas, E., Cooke, N. J., & Rosen, M. A. (2008). On teams, teamwork, and team performance: Discoveries and developments. *Human factors*, 50(3), 540-547.

Salas, E., Sims, D. E., & Burke, C. S. (2005). Is there a “big five” in teamwork? *Small group research*, 36(5), 555-599.

Salomon, G. (1997). *Distributed cognitions: Psychological and educational considerations*. Cambridge University Press.

Satratzemi, M., Stelios, X., & Tsompanoudi, D. (2023). Distributed pair programming in higher education: A systematic literature review. *Journal of Educational Computing Research*, 61(3), 546-577.

Satratzemi, M., Xinogalos, S., Tsompanoudi, D., & Karamitopoulos, L. (2018). Examining student performance and attitudes on distributed pair programming. *Scientific Programming*, 2018.

Saunders, M., Lewis, P., & Thornhill, A. (2019). *Research methods for business students* (Eighth ed.). Pearson education.

Saunders, M., Lewis, P., & Thornhill, A. (2023). *Research methods for business students* (9th ed.). Pearson.

Savage, S., & Piwek, P. (2019). Full report on challenges with learning to program and problem solve: an analysis of first year undergraduate Open University distance learning students' online discussions.

Sawyer, R. (2014). The Cambridge handbook of the learning sciences (Cambridge Handbooks in Psychology). Cambridge: Cambridge University Press. doi, 10, 317-330.

Schwandt, T. A. (2001). A postscript on thinking about dialogue. *Evaluation*, 7(2), 264-276.

Sebesta, R. W. (2016). *Concepts of Programming Languages, Global Edition*. (12th ed.). Pearson.

Sedgewick, R., & Wayne, K. (2016). *Computer science: An interdisciplinary approach*. Addison-Wesley Professional.

Shenton, A. K. (2004). Strategies for ensuring trustworthiness in qualitative research projects. *Education for information*, 22(2), 63-75.

Shneiderman, B. (1976). Exploratory experiments in programmer behavior. *International Journal of Computer & Information Sciences*, 5(2), 123-143.

Silva, L. S. (2020). *Investigating the Socially Shared Regulation of Learning in the Context of Programming Education* Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, Trondheim, Norway. <https://doi.org/10.1145/3341525.3394003>

Simon, B., Fitzgerald, S., McCauley, R., Haller, S., Hamer, J., Hanks, B., Helmick, M. T., Moström, J. E., Sheard, J., & Thomas, L. (2007). Debugging assistance for novices: a video repository. *ACM SIGCSE Bulletin*, 39(4), 137-151.

Sloane, K. D., & Linn, M. C. (1988). Instructional conditions in Pascal programming classes. In *Teaching and learning computer programming: Multiple research perspectives*. (pp. 207-235). Lawrence Erlbaum Associates, Inc.

Smite, D., Mikalsen, M., Moe, N. B., Stray, V., & Klotins, E. (2021). From Collaboration to Solitude and Back: Remote Pair Programming During COVID-19. In P. Gregory, C. Lassenius, X. Wang, & P. Kruchten, *Agile Processes in Software Engineering and Extreme Programming* International Conference on Agile Software Development, Cham.

Smith, R., & Rixner, S. (2019). The error landscape: Characterizing the mistakes of novice programmers. Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, USA.

So, M. H., & Kim, J. (2018). An analysis of the difficulties of elementary school students in python programming learning. *International Journal on Advanced Science, Engineering and Information Technology*, 8(4-2), 1507.

Sobral, S. R. (2020). Is pair programming in Higher Education a good strategy? *International Journal of Information and Education Technology*, 10(12).

Sokolic, D. (2022). Remote work and hybrid work organizations. *Economic and social development: Book of proceedings*, 202-213.

Soloway, E., Bonar, J., Woolf, B., Barth, P., Rubin, E., & Ehrlich, K. (1981). *Cognition and programming: Why your students write those crazy programs*. Proceedings of the National Educational Computing Conference, Texas, USA.

Soloway, E., & Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*(5), 595-609.

Soloway, E., & Spohrer, J. C. (Eds.). (1989). *Studying the Novice Programmer*. Lawrence Erlbaum.

Soloway, E., & Spohrer, J. C. (2013). *Studying the novice programmer*. Psychology Press.

Spinellis, D. (2016). *Effective Debugging: 66 Specific Ways to Debug Software and Systems*. Addison-Wesley Professional.

Spohrer, J. C., & Soloway, E. (1986). Novice mistakes: Are the folk wisdoms correct? *Communications of the ACM*, 29(7), 624-632.

Stahl, G., Koschmann, T. D., & Suthers, D. D. (2006). Computer-Supported Collaborative Learning: An Historical Perspective. *Cambridge Handbook of the Learning Sciences*, 409-426.

Steedman, H. (2012). Overview of apprenticeship systems and issues. *ILO contribution to the G20 task force on employment*, Geneva.

Sun, C., Yang, S., & Becker, B. (2024). Debugging in Computational Thinking: A Meta-analysis on the Effects of Interventions on Debugging Skills. *Journal of Educational Computing Research*, 07356331241227793.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2), 257-285.

Tan, O.-S. (2021). *Problem-based learning innovation: Using problems to power learning in the 21st century*. Gale Cengage Learning.

Taylor-Smith, E., Smith, S., Fabian, K., Berg, T., Meharg, D., & Varey, A. (2019). *Bridging the Digital Skills Gap: Are computing degree apprenticeships the answer?* ITiCSE '19 Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education, Aberdeen, UK. <https://doi.org/10.1145/3304221.3319744>

Teague, D., & Roe, P. (2007). Learning to program: Going pair-shaped. *Innovation in Teaching and Learning in Information and Computer Sciences*, 6(4), 4-22.

Tomlinson, C. A., & Imbeau, M. B. (2023). *Leading and managing a differentiated classroom*. ASCD.

Torgeir, D., Sridhar, N., VenuGopal, B., & Nils Brede, M. (2012). A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and software*, 85(6), 1213-1221. <https://doi.org/https://doi.org/10.1016/j.jss.2012.02.033>

Torraco, R. J. (2005). Writing integrative literature reviews: Guidelines and examples. *Human resource development review*, 4(3), 356-367.

Torraco, R. J. (2016). Writing integrative literature reviews: Using the past and present to explore the future. *Human resource development review*, 15(4), 404-428.

Tsai, C.-Y., Yang, Y.-F., & Chang, C.-K. (2015). *Cognitive Load Comparison of Traditional and Distributed Pair Programming on Visual Programming Language* 2015 International Conference of Educational Innovation through Technology (EITT 2015), Wuhan, China.

Tsan, J., Vandenberg, J., Fu, X., Wilkinson, J., Boulden, D., Boyer, K. E., Lynch, C., & Wiebe, E. (2019). *An investigation of conflicts between upper-elementary pair programmers* SIGCSE '19: Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA. <https://doi.org/10.1145/3287324.3293799>

Tsan, J., Weintrop, D., & Franklin, D. (2022). *An Analysis of Middle Grade Teachers' Debugging Pedagogical Content Knowledge* Proceedings of the 27th ACM Conference on on Innovation and Technology in Computer Science Education Vol. 1, Dublin, Ireland.

Tufford, L., & Newman, P. (2012). Bracketing in qualitative research. *Qualitative social work*, 11(1), 80-96.

Van Oers, B. (1998). From context to contextualizing. *Learning and instruction*, 8(6), 473-488.

Van Someren, M. W. (1990). What's wrong? Understanding beginners' problems with Prolog. *Instructional science*, 19, 257-282.

Veerasamy, A. K., D'Souza, D., & Laakso, M.-J. (2016). Identifying novice student programming misconceptions and errors from summative assessments. *Journal of Educational Technology Systems*, 45(1), 50-73.

Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, 23(5), 459-494.

Vossoughi, S., & Bevan, B. (2014). Making and tinkering: A review of the literature. *National Research Council Committee on Out of School Time STEM*, 67, 1-55.

Vourletsis, I., Politis, P., & Karasavvidis, I. (2021). The Effect of a Computational Thinking Instructional Intervention on Students' Debugging Proficiency Level and Strategy Use. *Research on E-Learning and ICT in Education: Technological, Pedagogical and Instructional Perspectives*, 15-34.

Vygotsky, L. S., & Cole, M. (1978). *Mind in society: Development of higher psychological processes*. Harvard university press.

Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM*, 25(7), 446-452.

Weiser, M. (1984). Program slicing. *IEEE Transactions on Software Engineering*(4), 352-357.

Welsh, E. (2002). Dealing with data: Using NVivo in the qualitative data analysis process. *Forum Qualitative Sozialforschung Forum: Qualitative Social Research*, 3(2). <https://doi.org/10.17169/fqs-3.2.865>

Wetton, R. (2021). Managing Virtual Teams: Creating a Virtual Community. In *Intercultural Management in Practice*. Emerald Publishing Limited.

Whalley, J., Settle, A., & Luxton-Reilly, A. (2021). *Novice Reflections on Debugging* Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, Virtual Event, USA. <https://doi.org/10.1145/3408877.3432374>

Whalley, J., Settle, A., & Luxton-Reilly, A. (2023). A Think-aloud Study of Novice Debugging. *ACM Trans. Comput. Educ.*, 23(2), 1. <https://doi.org/10.1145/3589004>

Whittemore, R., & Knafl, K. (2005). The integrative review: updated methodology. *Journal of advanced nursing*, 52(5), 546-553.

Williams, L., & Kessler, R. (2002). *Pair programming: Experience the difference* Extreme Programming and Agile Methods—XP/Agile Universe 2002: Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4–7, 2002 Proceedings 2, Chicago, IL, USA.

Williams, L., Kessler, R. R., Cunningham, W., & Jeffries, R. (2000). Strengthening the case for pair programming. *IEEE software*, 17(4), 19-25.

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33-35.

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. *ACM SIGCSE Bulletin*, 28(3), 17-22.

Wolter, S. C., & Ryan, P. (2011). Apprenticeship. In *Handbook of the Economics of Education* (Vol. 3, pp. 521-576). Elsevier.

Xu, F., & Correia, A.-P. (2023). Adopting distributed pair programming as an effective team learning activity: a systematic review. *Journal of Computing in Higher Education*, 1-30.

Yen, C.-Z., Wu, P.-H., & Lin, C.-F. (2012). *Analysis of experts' and novices' thinking process in program debugging*. Engaging Learners Through Emerging Technologies. ICT 2012. Communications in Computer and Information Science, vol 302, Hong Kong, China. https://doi.org/10.1007/978-3-642-31398-1_12

Yett, B., Hutchins, N., Snyder, C., Zhang, N., Mishra, S., & Biswas, G. (2020). *Evaluating student learning in a synchronous, collaborative programming environment*

through log-based analysis of projects International Conference on Artificial Intelligence in Education, **Tianjin, China**.

Yin, R. (2009). Case Study Research: Design and Methods, 4th edn Sage Publications. *Thousand Oaks*.

Yin, R., K. (2014). Case Study Research: Design and Methods. In (5th edition ed.). Thousand Oaks, CA: Sage publications.

Ying, K. M., Rodríguez, F. J., Dibble, A. L., & Boyer, K. E. (2021). Understanding Women's Remote Collaborative Programming Experiences: The Relationship between Dialogue Features and Reported Perceptions. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW3), 1-29.

Zeller, A., & Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2), 183-200.

Zhang, J., & Norman, D. A. (1994). Representations in distributed cognitive tasks. *Cognitive science*, 18(1), 87-122.

Zhang, Y., Paquette, L., Pinto, J. D., Liu, Q., & Fan, A. X. (2023). Combining latent profile analysis and programming traces to understand novices' differences in debugging. *Education and Information Technologies*, 28(4), 4673-4701.

Zhao, Q., Rabbah, R., Amarasinghe, S., Rudolph, L., & Wong, W.-F. (2008). How to do a million breakpoints: Efficient debugging using dynamic instrumentation. In L. Hendren, *Compiler Construction. CC 2008. Lecture Notes in Computer Science*, vol 4959. Springer, Berlin, Heidelberg.

Zimmerman, B. J. (2002). Becoming a self-regulated learner: An overview. *Theory into practice*, 41(2), 64-70.

Appendix A: Participants Information Sheet - Apprentices



Debugging Session & Interview Participant information sheet

Project Title: An evaluative case study of the program debugging behaviour of the paired Software Development Technician Apprentice in a geographically distributed environment.

For further information about how Lancaster University processes personal data for research purposes and your data rights please visit our webpage: www.lancaster.ac.uk/research/data-protection

I am a PhD candidate at Lancaster University. I would like to invite you to participate in a research study about the debugging behaviour of the paired Software Development Technician (SDT) Apprentice in a geographically distributed environment.

Please take time to read the following information carefully before you decide whether or not you wish to take part.

What is the study about?

This study aims to investigate debugging behaviours (and the influencing factors) of paired geographically distributed SDT apprentices working collaboratively on fixing written codes using technology-mediated agents. So, the compiler errors, the verbal and non-verbal interactions between pairs, how you build knowledge of the program's behaviour, technology agents' roles, the pattern of the debugging activities and eventually, resolution of issues (if applicable) will form the basis of this research.

Why have I been invited?

I have approached you because I am interested in understanding how SDTs in disparate locations work collaboratively on the same file, and at the same time go about locating and fixing bugs in a written programming code. I would be very grateful if you would agree to take part in this study.

What will I be asked to do if I take part?

If you decided to take part, this would involve the following:

- The researcher will organise two debugging sessions and one interview session with your assigned pair. It is reckoned that the two sessions of the debugging activities should not be longer than 2 hours altogether and the interview should not last more than 1 hour.
- You will be observed working with an assigned pair on remote debugging of a few Python programming codes with varied difficulties.
- The proceeding of the debugging activities in terms of code fixing, the conversations between the pair and video coverage of the entire proceedings will be made using the webcam of the laptop and stored in a password-protected Microsoft OneDrive.
- You will be expected to provide access to the modified programmatic code and the debugging and interview session recordings.
- The recordings will be reviewed by the researcher and to be used to determine the questions to follow-up during the interview session with a few parts of the recordings to be watched with you to further bolster the understanding of the phenomenon.
- Afterwards, you will be asked to participate in an interview session with your pair to answer questions on the debugging behaviours exhibited during the debugging sessions.

What are the possible benefits from taking part?

The study results may provide insight on the type of error messages generated by you as novice programmers while debugging codes, your debugging behaviour and how pairing of novice programmers in different locations work.

Do I have to take part?

No. It's completely up to you to decide whether or not you take part. Your participation is voluntary. If you decide not to take part in this study, this will not affect your studies and the way you are assessed on your course, or the relationship with the researcher or any staff within your area of employment.

What if I change my mind?

If you change your mind, you are free to withdraw at any time from your participation in this study. If you want to withdraw, please let me know, and I will extract any ideas or information (=data) you contributed to the study and destroy them. However, it is difficult and often impossible to take out data from one specific participant when this has already been anonymised or pooled together with other people's data. Therefore, you can only withdraw up to 2 weeks after taking part in the study.

What are the possible disadvantages and risks of taking part?

It is unlikely that there will be any major disadvantages to taking part apart from losing the time taken to participate in the debugging session and the interview session. In total, this is likely to take about three hours.

Will my data be identifiable?

After the observation and the interview, only I, the researcher conducting this study will have access to the ideas you share with me. I will keep all personal information about you (e.g. your name and other information about you that can identify you) confidential, that is I will not share it with others. I will remove any personal information from the written record of your contribution. All reasonable steps will be taken to protect your anonymity in this project.

How will we use the information you have shared with us and what will happen to the results of the research study?

I will use the information you have shared with me only in the following ways:

I will use it for research purposes only. This will include my PhD thesis and other publications, for example journal articles. I may also present the results of my study at academic or practitioner conferences. When writing up the findings from this study, I would like to reproduce some of the views and ideas you shared with me, but I will only use anonymised quotes (e.g. from my interview with you), so that although I will use your exact words, all reasonable steps will be taken to protect your anonymity in publications.

How my data will be stored

Your data will be stored in encrypted files (that is no-one other than me, the researcher, will be able to access them) and on password-protected computers. I will keep data that can identify you separately from non-personal information. The data will be subsequently destroyed after the thesis is completed.

What if I have a question or concern?

If you have any queries or if you are unhappy with anything that happens concerning your participation in the study, please contact myself on o.jolugbo@lancaster.ac.uk.

You can also contact my supervisor, Prof Don Passey on d.passey@lancaster.ac.uk, telephone number: [+44 \(0\) 1524 592314](tel:+441524592314).

Director of Studies, Doctoral Programme in e-Research and Technology Enhanced Learning
Department of Educational Research
Lancaster University
LA1 4YD

If you have any concerns or complaints that you wish to discuss with a person who is not directly involved in the research, you can also contact: Professor Paul Ashwin, Head of Department, Educational

Research, County South, Lancaster University, Lancaster, United Kingdom, LA1 4YD. Telephone: +44 (0) 1524 593572

This study has been reviewed and approved by the Faculty of Arts and Social Sciences and Lancaster Management School's Research Ethics Committee.

Appendix B: Participants Information Sheet - Work Based Mentors & Trainers



Workplace Mentor Participant information sheet

Project Title: An evaluative case study of the program debugging behaviour of the paired Software Development Technician Apprentice in a geographically distributed environment.

For further information about how Lancaster University processes personal data for research purposes and your data rights, please visit our webpage: www.lancaster.ac.uk/research/data-protection

I am a PhD candidate at Lancaster University, and I would like to invite you to take part in a research study about: Debugging behaviour of the paired Software Development Technician (SDT) Apprentice in a geographically distributed environment.

Please take time to read the following information carefully before you decide whether or not you wish to take part.

What is the study about?

This study aims to investigate debugging behaviours (and the influencing factors) of paired geographically distributed SDT apprentices working collaboratively on fixing written codes using technology-mediated agents. So, the compiler errors, the verbal and non-verbal interactions between pairs, how they build knowledge of the program's behaviour, technology agents' roles, the pattern of the debugging activities and eventually, resolution of issues (if applicable) will form the basis of this research.

Why have I been invited?

As a workplace mentor, I have approached you because I am interested in understanding how SDTs in disparate locations work collaboratively on the same file, and at the same time go about locating and fixing bugs in a written programming code. I would be very grateful if you would agree to take part in this study.

What will I be asked to do if I take part?

If you decided to take part, this would involve the following:

- The researcher will organise a focus group session using Microsoft Team with other workplace mentors that should not be more than 6 participants and lasting not more than two hours.
- You will be asked to participate in this focus group where your experience working with SDT apprentices will be discussed generally and more specifically, the type of bugs usually generated by them and your understanding of how SDT apprentices (or paired) go about debugging programming codes in terms of debugging strategies.
- Another area of discussion will be the emerging themes originating from the findings from the apprentice's debugging sessions and interview sessions, with the aim of seeking further clarifications or insights based on your own working knowledge of the phenomenon.
- The entire proceedings will be recorded using the laptop's webcam and stored in a password-protected Microsoft OneDrive.
- You will be asked to provide access to the recordings of the focus group session recordings.
- The recordings will be reviewed by the researcher, analysed and used to further bolster understanding of the phenomenon.

What are the possible benefits from taking part?

The study results may provide insight on the type of error messages generated by SDT apprentices while debugging codes, their debugging strategies and how pairing of novice programmers in different locations work. Also, it will help in sharing good practices from other mentors about how best to support apprentices with low debugging skills.

Do I have to take part?

No. It's completely up to you to decide whether or not you take part. Your participation is voluntary and has no bearing on your job and standing in your organisation.

What if I change my mind?

If you change your mind, you are free to withdraw at any time from your participation in this study. If you want to withdraw, please let me know, and I will extract any ideas or information (=data) you contributed to the study and destroy them. However, it is difficult and often impossible to take out data from one specific participant when this has already been anonymised or pooled together with other people's data. Your participation is voluntary, and you can withdraw from the study at any time before the focus group begins, but you will not be able to withdraw your contributions to the discussion once the recording has started. Also, if you are involved in a focus group and then withdraw, your data will remain part of the study. As your data will be part of the ongoing conversation, it is understandable that it cannot be destroyed. The researcher will try to disregard your views when analysing the focus group data, but this may not always be possible.

What are the possible disadvantages and risks of taking part?

It is unlikely that there will be any major disadvantages to taking part, apart from losing the time taken to participate in the debugging session and the interview session. In total, this is likely to take about two and a half hours. There is a possibility of a workplace mentor feeling obligated to participate in the discussion, but they are not under any compulsion to participate in the discussion.

Will my data be identifiable?

After the focus group session, only I, the researcher conducting this study, will have access to the ideas you share. I will keep all personal information about you (e.g. your name and other information about you that can identify you) confidential, that is, I will not share it with others. I will remove any personal information from the written record of your contribution. All reasonable steps will be taken to protect your anonymity in this project.

How will we use the information you have shared with us and what will happen to the results of the research study?

I will use the information you have shared with me only in the following ways:

I will use it for research purposes only. This will include my PhD thesis and other publications, for example journal articles. I may also present the results of my study at academic or practitioner conferences. When writing up the findings from this study, I would like to reproduce some of the views and ideas you shared with me, but I will only use anonymised quotes (e.g. from the focus group session), so that although I will use your exact words, all reasonable steps will be taken to protect your anonymity in publications.

How my data will be stored

Your data will be stored in encrypted files (that is no-one other than me, the researcher, will be able to access them) and on password-protected computers. I will keep data that can identify you separately from non-personal information. The data will be subsequently destroyed after the thesis is completed.

What if I have a question or concern?

If you have any queries or if you are unhappy with anything that happens concerning your participation in the study, please contact myself on o.jolugbo@lancaster.ac.uk.

You can also contact my supervisor, Prof Don Passey on d.passey@lancaster.ac.uk, telephone number: +44 (0) 1524 592314. Director of Studies, Doctoral Programme in e-Research and Technology Enhanced Learning Department of Educational Research Lancaster University LA1 4YD

If you have any concerns or complaints that you wish to discuss with a person who is not directly involved in the research, you can also contact Professor Paul Ashwin, Head of Department, Educational Research, County South, Lancaster University, Lancaster, United Kingdom, LA1 4YD. Telephone: +44 (0) 1524 593572

This study has been reviewed and approved by the Faculty of Arts and Social Sciences and Lancaster Management School's Research Ethics Committee.

Thank you for considering your participation in this project.

Appendix C: Participant Consent Form – Apprentices

CONSENT FORM



Project Title: An evaluative case study of the program debugging behaviour of the paired Software Development Technician Apprentice in a geographically distributed environment.

Name of Researcher: Olajide Jolugbo

Email: o.jolugbo@lancaster.ac.uk

Please tick each box

1. I confirm that I have read and understand the information sheet for the above study. I have had the opportunity to consider the information, ask questions and have had these answered satisfactorily.	<input type="checkbox"/>
2. I understand that my participation is voluntary and that I am free to withdraw at any time during my participation in this study and within 2 weeks after I took part in the study, without giving any reason. If I withdraw within 2 weeks of taking part in the study, my data will be removed.	<input type="checkbox"/>
3. If I am participating in the interview, I understand that any information disclosed within the interview remains confidential to the group, and I will not discuss the interview with or in front of anyone who was not involved unless I have the relevant person's express permission.	<input type="checkbox"/>
4. I understand that any information given by me may be used in future reports, academic articles, publications or presentations by the researcher/s, but my personal information will not be included, and all reasonable steps will be taken to protect the anonymity of the participants involved in this project.	<input type="checkbox"/>
5. I understand that my name/my organisation's name will not appear in any reports, articles, or presentation without my consent.	<input type="checkbox"/>
6. I agree to the video-recording of my pair debugging sessions and give consent for my program codes to be used for the purpose of this study.	<input type="checkbox"/>
7. I understand that any interviews will be audio-recorded and transcribed, and that data will be protected on encrypted devices and kept secure.	<input type="checkbox"/>
8. I understand that any interviews will be video-recorded and transcribed, and that data will be protected on encrypted devices and kept secure.	<input type="checkbox"/>
9. I understand that the images or recording will only be generated as a part of data collection, and will not be used for any other purpose. I also understand that the data/recordings will be destroyed after the completion of the thesis.	<input type="checkbox"/>
10. I agree to take part in the above study.	<input type="checkbox"/>

Name of Participant _____

Date _____

Signature _____

I confirm that the participant was given an opportunity to ask questions about the study, and all the questions asked by the participant have been answered correctly and to the best of my ability. I confirm that the individual has not been coerced into giving consent, and the consent has been given freely and voluntarily.

Signature of Researcher /person taking the consent _____ Date _____ Day/month/year

One copy of this form will be given to the participant and the original kept in the files of the researcher at Lancaster University

Appendix D: Participant Consent Form – Work-Based Mentors & Trainers

CONSENT FORM

Project Title: An evaluative case study of the program debugging behaviour of the paired Software Development Technician Apprentice in a geographically distributed environment.

Name of Researcher: Olajide Jolugbo

Email: o.jolugbo@lancaster.ac.uk

Please tick each box

I confirm that I have read and understand the information sheet for the above study. I have had the opportunity to consider the information, ask questions and have had these answered satisfactorily.	<input type="checkbox"/>
I understand that my participation is voluntary and that I am free to withdraw at any time during my participation in this study, without giving any reason. I understand that I can withdraw from the study at any time before the focus group begins but will not be able to withdraw my contributions to the discussion once recording has started. If I am involved in focus groups and then withdraw my data will remain part of the study. As my data will be part of the ongoing conversation, I understand that it cannot be destroyed. I understand that the researcher will try to disregard my views when analysing the focus group data, but I am aware that this will not always be possible.	<input type="checkbox"/>
I understand that any information disclosed within the focus group remains confidential to the group, and I will not discuss the focus group with or in front of anyone who was not involved unless I have the relevant person's express permission.	<input type="checkbox"/>
I understand that any information given by me may be used in future reports, academic articles, publications, or presentations by the researcher, but my personal information will not be included, and all reasonable steps will be taken to protect my anonymity in this project.	<input type="checkbox"/>
I understand that my name/my organisation's name will not appear in any reports, articles, or presentation without my consent.	<input type="checkbox"/>
I understand that the focus group will be audio-recorded and transcribed, and that data will be protected on encrypted devices and kept secure.	<input type="checkbox"/>
I understand that the focus group will be video-recorded and transcribed, and that data will be protected on encrypted devices and kept secure.	<input type="checkbox"/>
I understand that the images or recording will only be generated as a part of data collection, and will not be used for any other purpose. I also understand that the data/recordings will be destroyed after the completion of the thesis.	<input type="checkbox"/>
I agree to take part in the above study.	<input type="checkbox"/>

I confirm that the participant was given an opportunity to ask questions about the study, and all the questions asked by the participant have been answered correctly and to the best of my ability. I confirm that the individual has not been coerced into giving consent, and the consent has been given freely and voluntarily.

Signature of Researcher /person taking the consent _____ Date _____ Day/month/year

One copy of this form will be given to the participant and the original kept in the files of the researcher at Lancaster University

Appendix E: Ethics Approval

Educational
Research

Lancaster
University



11th June 2020

Dear Olajide Jolugbo

Thank you for submitting your ethics application and additional information for "An evaluative case study of the program debugging behaviour of the paired Software Development Technician Apprentice in a geographically distributed environment" The information you provided has been reviewed and I can confirm that approval has been granted for this project".

As principal investigator your responsibilities include:

- ensuring that (where applicable) all the necessary legal and regulatory requirements in order to conduct the research are met, and the necessary licenses and approvals have been obtained;
- reporting any ethics-related issues that occur during the course of the research or arising from the research (e.g. unforeseen ethical issues, complaints about the conduct of the research, adverse reactions such as extreme distress) to your Supervisor.
- submitting details of proposed substantive amendments to the protocol to your supervisor for approval.

Please do not hesitate to contact me if you require further information about this.

Kind regards

Alice Jesmont -

Alice Jesmont
TEL Programme Administrator

Appendix F: The Bugged Python Code

```
def calculate_payroll(hours_worked, hourly_rate)
    # SYNTAX ERROR (SE01): Missing colon

        gross_pay = hours_worked x hourly_rate
    # SYNTAX ERROR (SE02): Used 'x' instead of '*'

    # LOGICAL ERROR (LE01): Incorrect tax value
    tax_rate = 15

    # SYNTAX ERROR (SE03): Missing colon after 'if'
    if gross_pay > 6000
        tax = gross_pay * (tax_rate/100)
    else
        # LOGICAL ERROR (LE02): Wrong tax rate
        tax = gross_pay * 0.05

    net_pay = gross_pay - tax
    # LOGICAL ERROR (LE03): Shouldn't subtract tax if gross_pay is below a certain
    threshold

    return 'Total Pay: ', str(gross_pay) + ", Net Salary: " + str(net_pay)
    # SYNTAX ERROR (SE04): Mismatched string concatenation

# SYNTAX ERROR (SE05): 'def' typo
df main():
    hours = input("Input hours: ")
    rate = input("Input rate: $")

    # RUNTIME ERROR (RE01): Input is string and not converted to number
    payroll_info = calculate_payroll(hours, rate)

    # SYNTAX ERROR (SE06): print without parentheses
    print payroll_info

    # RUNTIME ERROR (RE02): Undefined variable 'rates'
    print(rates[0])

# LOGICAL ERROR (LE04): Improper use of '__name__'
if name = "__main__":
    # SYNTAX ERROR (SE07): Single '=' used instead of '==='
    main()

    # SYNTAX ERROR (SE08): Incorrectly closed string
    role = input("Enter employee's role:")

    # SYNTAX ERROR (SE09): Incorrect indentation
```

```
if role == "Manager":  
    # LOGICAL ERROR (LE05): Bonus amount doesn't make sense without context  
    bonus = 2000  
    print("Bonus: ", bonus)  
  
    # SYNTAX ERROR (SE10): Else without a prior if (due to the indentation error  
    above)  
    else:  
        print("No bonus")  
  
# LOGICAL ERROR (LE06): Redundant and incorrect code  
bonus = 100  
print("All employees get a bonus of: ", bonus)  
  
# SYNTAX ERROR (SE11): Incomplete 'for' loop  
for i in range(5)  
    print(i)  
    # RUNTIME ERROR (RE03): Infinite loop due to missing colon and indentation
```

Appendix G: Sample DYADs End of Session Codes

The Code after the Debugging Session - SDT23 & SDT24

The session concluded with the code still containing unresolved errors. The final code, with comments indicating both fixed and unfixed errors, demonstrated their efforts and learning process.

```
def calculate_payroll(hours_worked, hourly_rate):
    # ERROR FIXED (SE01): Missing colon at the end of function definition
    # ERROR FIXED (SE02): Incorrect operator, used 'x' instead of '*'
    gross_pay = hours_worked * hourly_rate

    # ERROR FIXED (LE01): Incorrect tax value, adjusted to 10%
    tax_rate = 10

    # ERROR FIXED (SE03): Missing colon after 'if'
    if gross_pay > 6000:
        # ERROR NOT FIXED (LE02): Incorrect logic for tax calculation
        tax = gross_pay * (tax_rate / 100)
    else:
        # ERROR NOT FIXED (LE03): Incorrect tax rate for lower bracket
        tax = gross_pay * 0.05

    net_pay = gross_pay - tax
    # ERROR FIXED (SE04): Corrected string concatenation
    return 'Total Pay: ' + str(gross_pay) + ', Net Salary: ' + str(net_pay)

# ERROR FIXED (SE05): 'def' typo
def main():
    hours = input("Input hours: ")
    rate = input("Input rate: $")

    # ERROR NOT FIXED (RE01): Input is string and not converted to number
    payroll_info = calculate_payroll(hours, rate)

    # ERROR FIXED (SE06): Added missing parentheses for print function
    print(payroll_info)

    # ERROR NOT FIXED (RE02): Undefined variable 'rates'
    print(rates[0])

    # ERROR FIXED (LE04): Corrected logic for '__name__' check
    # ERROR FIXED (SE07): Corrected comparison operator to '=='
    if __name__ == "__main__":
        main()
```

```
# ERROR NOT FIXED (SE08): Incorrectly closed string
role = input("Enter employee's role:")

# ERROR NOT FIXED (SE09): Incorrect indentation
if role == "Manager":
    # ERROR NOT FIXED (LE05): Bonus calculation lacks context
    bonus = 2000
    print("Bonus: ", bonus)

# ERROR NOT FIXED (SE10): Else without a prior if (due to the indentation error above)
else:
    print("No bonus")

# ERROR NOT FIXED (LE06): Redundant and incorrect bonus logic
bonus = 100
print("All employees get a bonus of: ", bonus)

# ERROR NOT FIXED (SE11): Incomplete 'for' loop
for i in range(5)
    # ERROR NOT FIXED (RE03): Infinite loop due to missing colon
    print(i)
```

Appendix H: DYAD Interview Protocols

1. How would you rate your programming skills and experience on a scale of 1 to 10, with 1 indicating a beginner level and 10 representing a highly skilled level?
2. What specific debugging strategies have you previously used in your programming tasks, and what led you to choose these methods?
3. What types of error messages do you find challenging while collaborating to solve and fix errors in the Python code?
4. What specific strategies do you employ in locating and rectifying bugs in the program, and why have you chosen these methods?
5. From your experience in the observed session, how did working together impact your approach to locating and fixing errors in the Python code, especially given your geographical distribution?
 - 5a Further to the answer provided to question 5, can you tell me in specific terms how working together impacted the strategy used or the way you approached the code debugging?
6. Can you describe the methods or strategies you used to distribute responsibilities and manage cognitive workload during the debugging process in your remote pairing?
7. Reflecting on the recorded hypothetical debugging session, how did using Integrated Development Environment (IDE) tools enhance your effectiveness and help mitigate the challenges you faced while debugging programs together in distributed pair debugging of Python code?
8. Using examples from the debugging session, what specific obstacles did you, as paired and geographically dispersed SDT apprentices, encounter while collaborating to resolve programming bugs?
9. Why do you think these particular challenges arose during your collaboration to fix bugs in the Python code, especially given your geographical separation?
10. How was your experience with the debugging session alongside your partner?

Appendix I: Focus Group Protocols

Thank you for participating in this focus group session. This session aims to tap into your wealth of experience in software development and working with the SDT apprentice debugging programming.

Introduction

1. Could we begin this discussion by exploring your experiences and observations on debugging programming codes while working with apprentices?

Discussion on the themes from the apprentice investigation and their personal experiences

2. What types of debugging strategies have you observed your apprentices using to identify and rectify bugs in programming code?
3. In your view, what are the likely contributing factors to apprentices adopting these specific debugging strategies?
4. How do you rate the effectiveness of these strategies in assisting apprentices to fix bugs efficiently, and what are the reasons for your assessment?
5. One recurring theme from the debugging sessions was the challenge surrounding mental models. Could you elaborate on the factors contributing to incorrect mental models among apprentices?

Conclusion

6. To conclude, could you share your insights on the known or perceived challenges that apprentices commonly face while debugging programming codes?
7. How do you think they can better be supported in improving your debugging practice?

Thank you for taking time out of your busy schedule to participate in this study.

Appendix J: Sample Transcript of the Debugging Session

Transcript of Debugging Session between SDT15 and SDT16

Session Start: 09:00 AM

Initial Run of the Script

SDT16 (Navigator): "Before we start fixing, let's run the script as is. We need to identify all the errors it throws up."

SDT15 (Driver): "Agreed. Executing the script now to catch the initial errors."

Script Execution Result: Error - SyntaxError on line 1: invalid syntax.

First 15 Minutes: Identifying and Correcting Syntax Errors

SDT15: "Looks like the first snag is a syntax error at the very beginning. Ah, we missed the colon after the function declaration. Such a small thing can cause a big issue."

SDT16: "Exactly, the colon is crucial in Python to indicate the start of the function block. Please add it at the end of the function declaration line."

SDT15 quickly adds the colon, fixing the syntax error *SE01.

SDT16: "Great, now let's rerun the script to check for the next batch of errors."

SDT15: "Hmm, now we have a TypeError. Oh, we used 'x' for multiplication on line 3. It should be an asterisk '*'."

SDT16: "That's a common mistake when switching from math notation to programming. Replace 'x' with '*'."

SDT15 promptly corrects the multiplication symbol *SE02 and re-runs the script.

SDT15: "Another syntax error, this time on the 'if' statement in line 7. We forgot the colon again."

SDT16: "The colon is crucial for if-else structures as well. Add it to signify the beginning of the if block."

SDT15 corrects the missing colon *SE03.

To trace the program's logic flow, they insert print statements and uncover inaccuracies in the tax calculation.

SDT15: "According to our task, the tax rate should vary between 10% and 25% based on the gross pay. But here, we've incorrectly used 15% and 5%."

SDT16: "We need to modify these values to align with the specified tax brackets. That will fix the logical errors in tax calculation."

SDT15 updates the tax rates, addressing *LE01 and *LE02.

30 Minutes: Switching Roles and Correcting Further Errors

As per their plan, *SDT16 takes over as the Driver, and *SDT15 becomes the Navigator.

SDT16: "I'll handle the string concatenation error in the return statement. We should concatenate using '+' instead of commas."

SDT15: "That's correct. Using plus signs will properly combine the strings and variables."

SDT16 rectifies the string concatenation issue *SE04.

SDT15: "The next issue is with the main function definition. It's mistakenly written as 'df'."

SDT16: "Oh, that's a typo. Changing 'df' to 'def' to correctly define the main function."

SDT16 fixes the function definition typo *SE05 and runs the script, leading to a runtime error.

SDT15: "The runtime error suggests an issue with data types. We're not converting the input strings to numbers, which is essential for arithmetic operations."

SDT16: "Right, I'll convert the input strings to integers to resolve this."

SDT16 amends the code to convert inputs to integers, addressing *RE01.

SDT15: "There's also a line with an undefined variable 'rates'. It seems out of place."

Appendix K: Sample Transcript of Dyad's Interview

4. What specific strategies do you employ in locating and rectifying bugs in the program, and why have you chosen these methods?

SDT7: "In our recent debugging session, I leaned heavily on Trial-and-Error as my primary strategy. It involves testing various solutions to see what works and what doesn't, which I find effective for immediate, hands-on learning. For example, when we encountered SE02, the wrong operator for multiplication, I quickly experimented with the correct operators to fix it. I prefer this approach because it gives me a direct sense of interaction with the code. However, I realise it's not always the most efficient, especially for more complex errors like the logical ones we faced. That's where I find Code Review really valuable. Reviewing SDT8's code alterations and discussing them helped me understand different perspectives and solutions."

SDT8: "I tend to favour Print Statement Debugging as my go-to strategy. It allows me to track how data changes throughout the program, which is crucial for understanding how errors, particularly runtime ones, manifest. For instance, when tackling RE01, not converting string input to number, using print statements helped us trace where the type mismatch occurred. I find this method systematic and informative, especially when dealing with intricate code. In addition to that, I also see the merit in Code Review, as SDT7 mentioned. It's a collaborative effort that offers insights that one might miss when working alone. The trial-and-error approach used by SDT7 also complemented our session, bringing a more dynamic and exploratory angle to our debugging process."

Critical Analysis of Responses:

SDT7's Analysis: SDT7's preference for Trial-and-Error reflects a hands-on, experiential learning style. This approach is effective for immediate problem-solving but may lack efficiency with complex issues.

Their appreciation for Code Review indicates an understanding of the value of collaborative learning and different perspectives in debugging.

SDT8's Analysis: SDT8's use of Print Statement Debugging demonstrates a systematic and analytical approach, allowing for a clear understanding of program flow and data states.

Their recognition of the benefits of Code Review highlights the importance of collaboration in their debugging strategy, complementing their individual analytical approach with collective insights.

Overall Assessment: Both SDT7 and SDT8 have employed strategies that suit their individual learning styles while complementing each other's approaches. SDT7's hands-on Trial-and-Error method provides immediate feedback and learning, while SDT8's systematic Print Statement Debugging offers detailed insights into the program's operation. The incorporation of Code Review by both participants enhances their debugging process, allowing for collaborative problem-solving and learning from each other's perspectives. This combination of strategies suggests a well-rounded approach to debugging, balancing individual exploration with collaborative analysis.

Appendix L: Sample of Included Studies for the Critical Analysis

	Table 1: Sample of a summary document for the critical analysis (CA) of selected studies.			
	Included studies	CA tool	Quality rating	Evidence level
1	Gould, J. D., & Drongowski, P. (1974). An exploratory study of computer program debugging. <i>Human Factors: The Journal of the Human Factors and Ergonomics Society</i> , 16(3), 258-277.	JBI	Outstanding	95%
2	Vessey, I. (1985). Expertise in debugging computer programs: A process analysis. <i>International Journal of Man-Machine Studies</i> , 23(5), 459-494	JBI	Outstanding	95%
3	Katz, I. R., & Anderson, J. R. (1987). Debugging: An analysis of bug-location strategies. <i>Human-Computer Interaction</i> , 3(4), 351-399.	JBI	Good	85%
4	Allwood, C. M., & Björhag, C.-G. (1990). Novices' debugging when programming in Pascal. <i>International Journal of Man-Machine Studies</i> , 33(6), 707-724.	JBI	Outstanding	90%
5	Ahmadzadeh, M., Elliman, D., & Higgins, C. (2005). An analysis of patterns of debugging among novice computer science students. <i>ACM SIGCSE Bulletin</i> , 37(3), 84-88.	JBI	Outstanding	90%
6	Chintakovid, T., Wiedenbeck, S., Burnett, M., & Grigoreanu, V. (2006). Pair Collaboration in End-User Debugging. <i>Proceedings - IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2006</i> , Brighton, UK.	JBI	Outstanding	95%
7	Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. <i>Computer Science Education</i> , 18(2), 93-116.	JBI	Outstanding	90%
8	Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., & Zander, C. (2008). Debugging: the good, the bad, and the quirky--a qualitative analysis of novices' strategies. <i>ACM SIGCSE Bulletin</i> , 40(1), 163-167.	JBI	Outstanding	90%
9	Fitzgerald, S., McCauley, R., Hanks, B., Murphy, L., Simon, B., & Zander, C. (2010). Debugging from the student perspective. <i>IEEE Transactions on Education</i> , 53(3), 390-396.	JBI	Outstanding	95%
10	Murphy, L., Fitzgerald, S., Hanks, B., & McCauley, R. (2010). Pair debugging: a transactive discourse analysis <i>Proceedings of the Sixth international workshop on Computing education research</i> , Aarhus, Denmark.	JBI	Outstanding	95%
11	Yen, C.-Z., Wu, P.-H., & Lin, C.-F. (2012). Analysis of experts' and novices' thinking process in program debugging. <i>Engaging Learners Through Emerging Technologies. ICT 2012. Communications in Computer and Information Science</i> , vol 302, Hong Kong, China.	JBI	Outstanding	95%
12	Akinola, S. (2014). An Empirical Comparative Analysis of Programming Effort, Bugs Incurrence and Code Quality between Solo & Pair Programmers. <i>Middle-East Journal of Scientific Research</i> , 21(12), 2231-2237.	JBI	Outstanding	100%
13	McCall, D., & Kölling, M. (2014). Meaningful categorisation of novice programmer errors. In <i>2014 IEEE Frontiers in Education Conference (FIE) Proceedings</i> (pp. 1-8). IEEE.	JBI	Outstanding	94%

14	Pritchard, D. (2015). Frequency distribution of error messages. Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools, Pittsburgh, PA, USA.	JBI	Outstanding	94%
15	Alqadi, B. S., & Maletic, J. I. (2017). An Empirical Study of Debugging Patterns Among Novices Programmers Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, Seattle, Washington, USA.	JBI	Outstanding	95%
16	Ettles, A., Luxton-Reilly, A., & Denny, P. (2018). Common logic errors made by novice programmers. Proceedings of the 20th Australasian Computing Education Conference, Brisbane, Queensland, Australia.	JBI	Good	80%
17	Júnior, A. S., de Figueiredo, J. C. A., & Serey, D. (2019). Analysing the Impact of Programming Mistakes on Students' Programming Abilities. Brazilian Symposium on Computers in Education, Brazil.	JBI	Outstanding	90%
18	Kohn, T. (2019). The Error Behind The Message: Finding the Cause of Error Messages in Python Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, MN, USA.	JBI	Outstanding	100%
19	Smith, R., & Rixner, S. (2019). The error landscape: Characterizing the mistakes of novice programmers. Proceedings of the 50th ACM Technical Symposium on Computer Science Education, Minneapolis, USA.	JBI	Outstanding	94%
20	Jayathirtha, G., Fields, D., & Kafai, Y. (2020). Pair debugging of electronic textiles projects: Analyzing think-aloud protocols for high school students' strategies and practices while problem solving The Interdisciplinarity of the Learning Sciences, 14th International Conference of the Learning Sciences (ICLS) 2020, Nashville, USA.	JBI	Outstanding	95%
21	Michaeli, T., & Romeike, R. (2020). Investigating Students' Preexisting Debugging Traits: A Real World Escape Room Study Proceedings of the 20th Koli Calling International Conference on Computing Education Research, Koli, Finland.	JBI	Outstanding	95%
22	Smite, D., Mikalsen, M., Moe, N. B., Stray, V., & Klotins, E. (2021). From Collaboration to Solitude and Back: Remote Pair Programming During COVID-19. In P. Gregory, C. Lassenius, X. Wang, & P. Kruchten, Agile Processes in Software Engineering and Extreme Programming International Conference on Agile Software Development, Cham.	JBI	Outstanding	95%
23	Whalley, J., Settle, A., & Luxton-Reilly, A. (2021a). Analysis of a Process for Introductory Debugging Proceedings of the 23rd Australasian Computing Education Conference, Australia.	JBI	Outstanding	95%
24	Whalley, J., Settle, A., & Luxton-Reilly, A. (2021b). Novice Reflections on Debugging Proceedings of the 52nd ACM Technical Symposium on Computer Science Education, Virtual Event, USA.	JBI	Outstanding	95%
25	Baabdullah, A., & Kim, C. (2022). Supporting Collaborative Debugging Processes. Proceedings of the 15th International Conference on Computer-Supported Collaborative Learning-CSCL 2022, pp. 557-558, Hiroshima, Japan.	JBI	Outstanding	95%
26	Jeffries, B., Lee, J. A., & Koprinska, I. (2022). 115 Ways Not to Say Hello, World! Syntax Errors Observed in a Large-Scale Online CS0 Python Course. Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education Vol. 1 (ITiCSE 2022), Dublin, Ireland.	JBI	Outstanding	100%
27	Kim, C., Vasconcelos, L., Belland, B. R., Umutlu, D., & Gleasman, C. (2022). Debugging behaviours of early childhood teacher candidates with or without scaffolding. International Journal of Educational Technology in Higher Education, 19(1), 26.	JBI	Outstanding	95%

28	Alaboudi, A., & LaToza, T. D. (2023). What constitutes debugging? An exploratory study of debugging episodes. <i>Empirical Software Engineering</i> , 28(5), 117.	JBI	Outstanding	95%
29	Liu, Q., & Paquette, L. (2023). Using submission log data to investigate novice programmers' employment of debugging strategies. <i>LAK23: 13th International Learning Analytics and Knowledge Conference</i> , Arlington, TX, USA.	JBI	Outstanding	95%
30	Whalley, J., Settle, A., & Luxton-Reilly, A. (2023). A Think-aloud Study of Novice Debugging. <i>ACM Trans. Comput. Educ.</i> , 23(2), 1.	JBI	Outstanding	95%
31	Zhang, Y., Paquette, L., Pinto, J. D., Liu, Q., & Fan, A. X. (2023). Combining latent profile analysis and programming traces to understand novices' differences in debugging. <i>Education and Information Technologies</i> , 28(4), 4673-4701.	JBI	Outstanding	95%
32	Brown, N. C., Mac, V., Weill-Tessier, P., & Kölling, M. (2024). Writing Between the Lines: How Novices Construct Java Programs <i>Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (SIGCSE 2024)</i> , USA.	JBI	Outstanding	90%
33	Jayathirtha, G., Fields, D., & Kafai, Y. (2024). Distributed debugging with electronic textiles: understanding high school student pairs' problem-solving strategies, practices, and perspectives on repairing physical computing projects. <i>Computer Science Education</i> , 1-35.	JBI	Outstanding	100%
34	Morales-Navarro, L., Fields, D. A., & Kafai, Y. B. (2024). Understanding growth mindset practices in an introductory physical computing classroom: high school students' engagement with debugging by design activities. <i>Computer Science Education</i> , 1-31.	JBI	Outstanding	90%
35	Parkinson, M. M., Hermans, S., Gijbels, D., & Dinsmore, D. L. (2024). Exploring debugging processes and regulation strategies during collaborative coding tasks among elementary and secondary students. <i>Computer Science Education</i> , 1-28.	JBI	Good	75%

Appendix M: Debugging Session Codebook

Appendix N: DYAD Interview Codebook

DPP

Codes\\Interview\\Stage 1 & 2 - Familiarisation & Coding

Name	Description	Files	References
Absence of Physical Presence for Quick Clarification	These represent struggles in not being able to point or visually show parts of code	1	11
Acknowledged Limitations	Learners showed self-awareness by noting their own lack of proficiency or understanding. These admissions underline honesty about gaps in capability.	1	3
Agreement on Challenge	Multiple participants expressed a shared view that certain logical errors posed significant difficulty. The consistency in their sentiments adds weight to the issue's complexity.	1	5
Analytical approach	Describes instances where a participant employed step-by-step reasoning or formal techniques. It highlights a structured way of unravelling complex logic.	1	1
Analytical Demand	Reflects how some debugging tasks required high-level reasoning and mental exertion. Participants perceived the activity as cognitively intensive.	1	1
Analytical Gaps	Denotes errors that occurred due to missed steps or incomplete reasoning processes. This suggests an underdeveloped analytical sequence.	1	2
Big picture review	Participants referred to stepping back and reassessing the entire codebase. This top-down perspective helped in recontextualising the issue.	1	1
Breakdown strategy	Refers to the act of deconstructing a problem into simpler parts to aid resolution. Learners discussed breaking logic into manageable pieces.	1	1
Calculation Confusion	Errors emerged from difficulties in creating or tracing formula-based logic. Mathematical thinking was the barrier.	1	1
Code Isolation Strategy	Participants isolated specific blocks or lines of code to test or observe behaviour. This strategy helped to narrow the problem area.	1	2
Code Review	Apprentices systematically examined and critiqued each other's code to identify issues,	1	12

Name	Description	Files	References
	clarify logic, and enhance collaborative problem-solving.		
Code Structure	For syntax issues relating to structural formatting or layout.	1	4
Code Visibility Advantage	Clarity in formatting, naming, or organisation made it easier to follow the logic. Participants attributed their success partly to how readable the code was.	1	1
Collaborative clarity	Understanding emerged more clearly through discussions with peers. Explaining logic to others often led to personal insight.	1	3
Collaborative Insight	New interpretations or corrections were achieved by engaging with someone else's viewpoint. The collaboration brought forth alternative solutions.	1	4
Complex Logic Breakdown	Learners attempted to untangle highly intricate or nested conditions. The difficulty lay not in syntax but in logical architecture.	1	4
Concept Misuse	Participants misapplied key Python concepts, leading to logic flaws. These misunderstandings pointed to a superficial grasp of coding constructs.	1	2
Conditional Misinterpretation	Learners misunderstood how conditionals executed. This misreading caused flawed logic paths.	1	1
Context understanding	Problem-solving success relied on grasping the wider function or scenario. The learner needed to understand not just 'what' but 'why'.	1	3
Contextual Misuse	A function or logic piece was applied in the wrong context. The logic was sound, but its placement was flawed.	1	1
Data Flow Understanding	Focused on tracking how information moved through variables and functions. This tracking helped diagnose where logic broke down.	1	1
Deep Dive Debugging	Marked by a thorough and prolonged engagement with the problem. Participants drilled deep into the logic layer rather than skimming.	1	1
Deep Logic	This reflects the intellectual depth required to trace and correct logic faults rooted in Python intricacies or conceptual frameworks. It signifies scenarios where surface-level knowledge was insufficient.	1	2
Difficulty Conveying Thought Process Remotely	These quotes are focused on how apprentices struggled to explain, align, or communicate their reasoning without face-to-face interaction	1	27

Name	Description	Files	References
Distraction in Individual Work Environments	These highlight challenges in focus due to remote, uncontrolled environments.	1	6
Divide and conquer	The issue was resolved by segmenting it into independent subproblems. Learners described resolving each part methodically.	2	3
Division of Tasks Based on Strengths	Apprentices strategically assigned responsibilities based on individual strengths or comfort zones to manage complexity and maintain focus.	1	22
Driver-Navigator Role Sharing	Apprentices adopted a structured pairing model where one coded while the other observed and guided, helping distribute cognitive demands.	1	18
Error Complexity	The nature of the logic error was itself intricate and multi-layered. These were not beginner mistakes but advanced logic misfires.	1	10
Execution Flow	For difficulty understanding the order of execution in Python.	1	5
Experience Builds Mastery	Learners acknowledged that repeated exposure helped them improve. Experience was credited as a major enabler of logical reasoning.	1	2
Explaining syntax fixes	For verbal explanation, negotiation, or clarification of syntax fixes during collaboration	1	15
Fixing Syntax Error	For comments about actively identifying, correcting, or guiding others through syntax issues.	1	9
Flow Confusion	The challenge stemmed from not understanding how code progressed during execution. This lack of clarity hampered logical deductions.	1	1
Found It Challenging	A general admission that the task was tough, without further detail. These expressions still signal cognitive overload.	1	7
General Complexity	Applied when logic problems were described as difficult but without specific explanation. It captures vague but valid struggle.	1	5
Growth Mindset	Participants expressed confidence that they could learn and improve with effort. This forward-thinking attitude supports resilience.	1	1
Growth Through Challenge	Struggle was reframed as an opportunity for learning. Participants reflected positively on the difficulty.	1	4
IDE Debugger Usage	Apprentices used the IDE debugger to step through code and inspect variable states, enabling precise identification and correction of logic errors.	1	35
Infinite Loop	For errors involving loops that do not terminate.	1	17

Name	Description	Files	References
Knowledge Gaps	Errors resulted from lacking the foundational knowledge needed to apply logic. This code tracks missing prerequisites.	1	6
Lack of Non-Verbal Feedback	These reflect how the absence of visual, gestural, or facial cues hindered effective communication and understanding during remote debugging	1	33
Limited Experience and Skill Gaps	Apprentices expressed difficulty navigating debugging tasks due to being new to programming, lacking foundational knowledge, or still developing confidence in applying core concepts.	1	15
Logic and Flow Challenges	Combined challenges in understanding both the logic and how it executed. These situations involved overlapping difficulties.	1	6
Logic Struggle	Captures moments of emotional or cognitive difficulty expressed by learners tackling logical bugs. Participants voiced frustration and mental fatigue in trying to make sense of such errors.	1	4
Logical Errors Challenging	Serves as a general label for statements identifying logic bugs as hard. It doesn't specify which part was problematic.	1	5
Logical Reasoning Gaps	Participants struggled with understanding or applying correct logic within the code, particularly when handling conditionals, calculations, or the flow of decision-making.	1	61
Methodical Problem Solving	The participant used a structured, procedural approach to identify the issue. This code praises disciplined debugging.	1	1
Misalignment in Understanding	These reflect how apprentices experienced confusion or divergent interpretations of logic or instructions during debugging sessions	1	33
Missing Colon	For specific mention of missing colons in syntax.	1	1
Missing Syntax	For syntax errors due to missing elements like colons, brackets, or forgotten components.	1	39
Misunderstood Logic Flow	Participants misunderstood how one part of the code affected another. These errors revealed disconnects in logic mapping.	1	1
Misuse of 'name'	Highlights confusion around Python's special ' <code>__name__</code> ' variable. This is a specific example of concept misunderstanding.	1	1
Navigator insight	In pair programming, the navigator offered a useful perspective. The insight usually shifted the course of debugging.	1	1
Needs More Practice	Participant acknowledged needing repetition or further exposure to improve. Practice was seen as key to mastering logic.	1	1

Name	Description	Files	References
Other IDE Features	Apprentices benefited from additional IDE tools like version control integration, intelligent suggestions, and code completion to streamline their workflow.	1	3
Pair support for syntax	For collaborative efforts in addressing syntax errors through shared roles or peer help.	1	13
Paired Strengths	Learners described how teammates complemented their skills. Their collective effort covered individual weaknesses.	1	1
Pattern Matching	Apprentices looked for recurring structures or familiar error patterns to quickly locate and fix bugs based on previous experience.	1	3
Pattern-based reasoning	Participants applied familiar logic patterns to solve new problems. This indicates transfer of learning.	1	1
Pattern-based syntax strategy	For use of recurring patterns, visual tracing, or structured methods in spotting syntax issues.	1	6
Peer Review Strength	Logic errors were identified through peer feedback. Review mechanisms improved accuracy.	1	1
Peer support	Emotional or technical encouragement came from fellow learners. It acted as a buffer during challenging moments.	1	1
Print Statement Debugging	Apprentices inserted print statements and monitored console outputs to trace program behaviour and identify bugs during execution.	1	32
Progress Despite Errors	Learners recognised forward movement even when mistakes occurred. This shows perseverance.	1	1
Real-Time Code Sharing and Synchronisation	Tools enabling simultaneous editing and shared visibility helped apprentices maintain alignment and coordinate debugging in real-time.	1	2
Real-Time Tool Support for Coordination	Collaborative tools like IDE features and remote sharing platforms were used to support synchronised thinking and reduce mental strain.	1	37
Remote Collaboration Limits	The online or distant setup introduced difficulties in understanding logic. Distance added barriers to debugging.	1	8
Role swapping	Team members changed roles mid-task to better tackle logic issues. The switch brought fresh perspective.	1	2
Rubber Duck Debugging	Apprentices explained their code aloud—to a partner or inanimate object—to clarify their thinking and uncover logic errors.	1	7
Rubber ducking	Participants verbalised logic step-by-step, often to a peer or non-technical object. This externalisation clarified their thinking.	1	2

Name	Description	Files	References
Runtime Contrast	Participants reflected on how runtime errors differed from logic ones. This comparative insight helped focus their approach.	1	2
Runtime Error Complexity	Several apprentices found runtime errors difficult to resolve because they often appeared after code execution and required understanding how the program behaved dynamically.	1	46
Runtime Overwhelm	When runtime feels particularly complex or challenging.	1	13
Runtime Print Tracking	For those using print statements to trace issues.	1	4
Runtime Strategy Lacking	When trial-and-error or lack of method was highlighted.	1	2
Runtime Type Confusion	When the issue involves converting string to number or similar.	1	6
Runtime Uncertainty	For quotes where learners are confident with syntax but unsure about runtime.	1	24
Slicing	Slicing refers to the strategy of breaking down or isolating specific segments of code, such as functions, conditions, or loops, to analyse them independently. This helps apprentices reduce complexity by focusing only on the relevant part of the code where the error is suspected, making it easier to locate and fix bugs collaboratively.	1	3
Solution experimentation	Debugging involved trying multiple possible solutions to test logic. Learners described trial as a deliberate tactic.	1	1
Strategy Limitations	Existing methods or plans failed to resolve the logic issue. Learners were forced to reconsider their approach.	1	1
Syntax and Error Highlighting Features	The IDE's syntax highlighting, auto-indentation, and inline error notifications supported apprentices in quickly spotting and correcting code mistakes.	1	3
Syntax as role strength	For individuals who naturally took the lead on syntax due to confidence or skill	1	15
Syntax Complexity	Participants frequently encountered syntax errors that disrupted code execution, especially those involving Python-specific rules like indentation, string formatting, or punctuation.	1	151
Syntax Feels Easy	For those who found syntax errors more straightforward or gained confidence resolving them	1	25
Syntax First	Where participants mention syntax errors as their starting point in debugging.	1	3
Syntax for Beginners	For beginner-level ease, familiarity, or exposure to syntax debugging.	1	2

Name	Description	Files	References
Syntax is Tricky	For quotes that describe syntax errors as deceptively hard or initially difficult to handle.	1	6
Syntax Typo	For errors caused by typographical mistakes.	1	1
Technology-Related Collaboration Issues	These quotes reflect how tool-based issues like syncing, lag, edit conflicts, and IDE limitations disrupted collaboration	1	41
Think-Aloud Communication	Apprentices verbalised their reasoning and thought processes to clarify understanding and collaboratively work through errors.	1	21
Time Zone and Scheduling Difficulties	Quotes here reflect challenges related to coordinating across different locations or schedules.	1	5
Tinkering	Apprentices intuitively made small code changes and tested their effects as a way to explore and understand bugs.	1	3
Tool Access or Setup Issues	These quotes relate to initial difficulties in using or setting up collaboration tools.	1	26
Tool-assisted logic check	Software tools like debuggers or linters were used to identify logic faults. Participants credited these for catching errors.	1	2
Tool-Assisted Syntax Fix	For use of features like syntax highlighting, error popups, or debuggers to spot/fix syntax.	1	15
Tracing	Apprentices followed the program's flow line by line to understand how data moved and identify where the logic broke down.	1	7
Trial and Error	Apprentices experimented with different solutions without a predefined plan to see what resolved the issue through observation.	1	10
Trial and Error Limits	Participants noted that random guessing was ineffective for logic bugs. These problems needed deeper thought.	1	1
Turn-Taking and Role Swapping	Regular role alternation ensured balanced mental effort, reduced fatigue, and kept both apprentices engaged throughout the debugging session.	1	7
Type Conversion	For issues converting between data types (e.g. string to number)	1	8
Undefined Variable	For use of variables that were not defined before use.	1	5
Understanding Logic Flow	A clear picture of how logic moved through code aided debugging. This awareness streamlined troubleshooting.	1	5
Unresolved Logic Issues	Some logic problems remained unsolved by session end. The code captures lingering confusion.	1	2

Name	Description	Files	References
Unstable or Inconsistent Internet Connection	These quotes highlight issues caused by poor or unstable internet, affecting real-time collaboration or access to tools.	1	1
Use of Live Share for Remote Collaboration	Apprentices leveraged Live Share to collaboratively edit, navigate, and debug code from separate locations in real time.	1	2
Variable misuse	Mistakes occurred due to inappropriate variable assignment or tracking. This led to faulty logic.	1	1

Codes\\Interview\\Stage 3 - Theme Generation

Name	Description	Files	References
Cognitive Perception and Difficulty	This category reflects how runtime issues were perceived as overwhelming or uncertain, especially when apprentices couldn't predict behaviour or lacked confidence during execution.	1	43
Runtime Overwhelm	When runtime feels particularly complex or challenging.	1	13
Runtime Type Confusion	When the issue involves converting string to number or similar.	1	6
Runtime Uncertainty	For quotes where learners are confident with syntax but unsure about runtime.	1	24
Collaboration and Communication Aids	This category reflects the value of teamwork in resolving logical errors. It includes quotes where learners gained clarity or found solutions by explaining to peers, switching roles, or combining their strengths.	1	13
Collaborative clarity	Understanding emerged more clearly through discussions with peers. Explaining logic to others often led to personal insight.	1	3
Collaborative Insight	New interpretations or corrections were achieved by engaging with someone else's viewpoint. The collaboration brought forth alternative solutions.	1	4
Navigator insight	In pair programming, the navigator offered a useful perspective. The insight usually shifted the course of debugging.	1	1
Paired Strengths	Learners described how teammates complemented their skills. Their collective effort covered individual weaknesses.	1	1
Peer Review Strength	Logic errors were identified through peer feedback. Review mechanisms improved accuracy.	1	1
Peer support	Emotional or technical encouragement came from fellow learners. It acted as a buffer during challenging moments.	1	1
Role swapping	Team members changed roles mid-task to better tackle logic issues. The switch brought fresh perspective.	1	2
Collaborative and Reflective Techniques	This category highlights methods where apprentices explained or reviewed code with others (or to themselves) to gain insight, clarify thinking, and identify errors through reflection or external feedback.	2	20

Name	Description	Files	References
Code Review	Apprentices systematically examined and critiqued each other's code to identify issues, clarify logic, and enhance collaborative problem-solving.	1	12
Peer Review Strength	Logic errors were identified through peer feedback. Review mechanisms improved accuracy.	1	1
Rubber Duck Debugging	Apprentices explained their code aloud, to a partner or inanimate object, to clarify their thinking and uncover logic errors.	1	7
Connectivity Constraints	This category reflects how unstable internet disrupted communication, tool access, and real-time collaboration, especially in remote or bandwidth-limited settings.	1	1
Unstable or Inconsistent Internet Connection	These quotes highlight issues caused by poor or unstable internet, affecting real-time collaboration or access to tools.	1	1
Debugging Approaches for Runtime	This category captures how apprentices attempted to resolve runtime errors, particularly by using print statements or acknowledging a lack of structured strategy.	1	6
Runtime Print Tracking	For those using print statements to trace issues.	1	4
Runtime Strategy Lacking	When trial-and-error or lack of method was highlighted.	1	2
Debugging Tools and Execution Support	This category captures how apprentices leveraged key debugging tools, such as the IDE's step-through functionality and print statements, to inspect program execution, monitor variable states, and detect logical or runtime issues.	1	67
IDE Debugger Usage	Apprentices used the IDE debugger to step through code and inspect variable states, enabling precise identification and correction of logic errors.	1	35
Print Statement Debugging	Apprentices inserted print statements and monitored console outputs to trace program behaviour and identify bugs during execution.	1	32
Distractions and Focus Challenges	Distractions and Focus Challenges This includes issues with concentration due to noise, interruptions, or other environmental factors unique to working remotely from home or other informal settings	1	6
Distraction in Individual Work Environments	These highlight challenges in focus due to remote, uncontrolled environments.	1	6
Foundational Knowledge Gaps	This category reflects difficulties stemming from apprentices' limited prior experience with Python or programming generally, especially in	1	166

Name	Description	Files	References
	handling syntax-specific rules like indentation and punctuation.		
Limited Experience and Skill Gaps	Apprentices expressed difficulty navigating debugging tasks due to being new to programming, lacking foundational knowledge, or still developing confidence in applying core concepts.	1	15
Syntax Complexity	Participants frequently encountered syntax errors that disrupted code execution, especially those involving Python-specific rules like indentation, string formatting, or punctuation.	1	151
Interface Guidance and Visual Feedback	This category represents the supportive role of the IDE's user interface elements, including syntax highlighting, auto-suggestions, version control, and intelligent prompts, which helped learners identify errors and streamline their workflow.	1	6
Other IDE Features	Apprentices benefited from additional IDE tools like version control integration, intelligent suggestions, and code completion to streamline their workflow.	1	3
Syntax and Error Highlighting Features	The IDE's syntax highlighting, auto-indentation, and inline error notifications supported apprentices in quickly spotting and correcting code mistakes.	1	3
Interpretation and Understanding Conflicts	This category includes moments where apprentices interpreted instructions, logic, or errors differently, resulting in delays or confusion in collaborative debugging.	1	33
Misalignment in Understanding	These reflect how apprentices experienced confusion or divergent interpretations of logic or instructions during debugging sessions	1	33
Nature and Source of Syntax Errors	This category captures specific causes of syntax errors such as missing colons, structural mistakes, typographical errors, and overlooked elements. It reflects how apprentices encountered surface-level mistakes that disrupted code execution.	1	45
Code Structure	For syntax issues relating to structural formatting or layout.	1	4
Missing Colon	For specific mention of missing colons in syntax.	1	1
Missing Syntax	For syntax errors due to missing elements like colons, brackets, or forgotten components.	1	39
Syntax Typo	For errors caused by typographical mistakes.	1	1
Perceived Complexity and Emotional Response	This category captures how apprentices found logical errors emotionally or cognitively taxing. It	1	41

Name	Description	Files	References
	includes expressions of frustration, struggle, or general difficulty in making sense of complex conditions or flows.		
Agreement on Challenge	Multiple participants expressed a shared view that certain logical errors posed significant difficulty. The consistency in their sentiments adds weight to the issue's complexity.	1	5
Complex Logic Breakdown	Learners attempted to untangle highly intricate or nested conditions. The difficulty lay not in syntax but in logical architecture.	1	4
Error Complexity	The nature of the logic error was itself intricate and multi-layered. These were not beginner mistakes but advanced logic misfires.	1	10
Flow Confusion	The challenge stemmed from not understanding how code progressed during execution. This lack of clarity hampered logical deductions.	1	1
Found It Challenging	A general admission that the task was tough, without further detail. These expressions still signal cognitive overload.	1	7
General Complexity	Applied when logic problems were described as difficult but without specific explanation. It captures vague but valid struggle.	1	5
Logic Struggle	Captures moments of emotional or cognitive difficulty expressed by learners tackling logical bugs. Participants voiced frustration and mental fatigue in trying to make sense of such errors.	1	4
Logical Errors Challenging	Serves as a general label for statements identifying logic bugs as hard. It doesn't specify which part was problematic.	1	5
Perceived Difficulty of Syntax	This category includes apprentice's perceptions of syntax errors as either easy or deceptively tricky. Some found them manageable due to clear error messages, while others initially underestimated their complexity.	1	33
Syntax Feels Easy	For those who found syntax errors more straightforward or gained confidence resolving them	1	25
Syntax for Beginners	For beginner-level ease, familiarity, or exposure to syntax debugging.	1	2
Syntax is Tricky	For quotes that describe syntax errors as deceptively hard or initially difficult to handle.	1	6
Physical Separation Barriers	This category refers to the challenge of not being able to visually point to code or easily clarify issues due to being physically apart during remote pair programming.	1	11

Name	Description	Files	References
Absence of Physical Presence for Quick Clarification	These represent struggles in not being able to point or visually show parts of code	1	11
Real-Time Collaborative Platforms	This category includes tools that enabled synchronous work across distances, allowing apprentices to co-edit, share, and coordinate their debugging efforts in real time using platforms like Visual Studio Live Share.	1	4
Real-Time Code Sharing and Synchronisation	Tools enabling simultaneous editing and shared visibility helped apprentices maintain alignment and coordinate debugging in real-time.	1	2
Use of Live Share for Remote Collaboration	Apprentices leveraged Live Share to collaboratively edit, navigate, and debug code from separate locations in real time.	1	2
Real-Time Communication and Coordination Tools	This category reflects how verbalisation strategies and collaborative digital tools were used to coordinate thought processes, reduce confusion, and manage the cognitive load during remote or paired debugging.	1	58
Real-Time Tool Support for Coordination	Collaborative tools like IDE features and remote sharing platforms were used to support synchronised thinking and reduce mental strain.	1	37
Think-Aloud Communication	Apprentices verbalised their reasoning and thought processes to clarify understanding and collaboratively work through errors.	1	21
Reasoning and Logic Challenges	This category houses instances where apprentices lacked the cognitive strategies to understand and apply logic effectively, especially when dealing with conditionals, calculations, or the broader logic structure.	1	61
Logical Reasoning Gaps	Participants struggled with understanding or applying correct logic within the code, particularly when handling conditionals, calculations, or the flow of decision-making.	1	61
Reflection and Learning Dispositions	This category captures reflective mindsets where apprentices identified growth, perseverance, or learning from mistakes. It also houses observations about how experience, context, and collaboration influenced their progress.	1	31
Context understanding	Problem-solving success relied on grasping the wider function or scenario. The learner needed to understand not just 'what' but 'why'.	1	3
Experience Builds Mastery	Learners acknowledged that repeated exposure helped them improve. Experience was credited as a major enabler of logical reasoning.	1	2
Growth Mindset	Participants expressed confidence that they could learn and improve with effort. This forward-thinking attitude supports resilience.	1	1

Name	Description	Files	References
Growth Through Challenge	Struggle was reframed as an opportunity for learning. Participants reflected positively on the difficulty.	1	4
Logic and Flow Challenges	Combined challenges in understanding both the logic and how it executed. These situations involved overlapping difficulties.	1	6
Misuse of 'name'	Highlights confusion around Python's special ' <code>__name__</code> ' variable. This is a specific example of concept misunderstanding.	1	1
Progress Despite Errors	Learners recognised forward movement even when mistakes occurred. This shows perseverance.	1	1
Remote Collaboration Limits	The online or distant setup introduced difficulties in understanding logic. Distance added barriers to debugging.	1	8
Runtime Contrast	Participants reflected on how runtime errors differed from logic ones. This comparative insight helped focus their approach.	1	2
Unresolved Logic Issues	Some logic problems remained unsolved by session end. The code captures lingering confusion.	1	2
Variable misuse	Mistakes occurred due to inappropriate variable assignment or tracking. This led to faulty logic.	1	1
Remote Expression Challenges	This category captures how apprentices struggled to communicate ideas clearly without the benefit of facial expressions, gestures, or in-person context, leading to misunderstandings, over-explaining, or extra effort in articulation.	1	60
Difficulty Conveying Thought Process Remotely	These quotes are focused on how apprentices struggled to explain, align, or communicate their reasoning without face-to-face interaction	1	27
Lack of Non-Verbal Feedback	These reflect how the absence of visual, gestural, or facial cues hindered effective communication and understanding during remote debugging	1	33
Runtime Behaviour Confusion	This category represents the difficulty of identifying and resolving bugs that only emerged during code execution—particularly where program behaviour was unpredictable or misunderstood.	1	46
Runtime Error Complexity	Several apprentices found runtime errors difficult to resolve because they often appeared after code execution and required understanding how the program behaved dynamically.	1	46
Scheduling and Coordination Hurdles	This category captures difficulties in syncing schedules across time zones or managing different availability patterns, which limited collaboration windows.	1	5

Name	Description	Files	References
Time Zone and Scheduling Difficulties	Quotes here reflect challenges related to coordinating across different locations or schedules.	1	5
Skill Gaps and Cognitive Limitations	This category includes participants' admissions of limited knowledge, misunderstood logic, or conceptual misapplications. It highlights areas where deeper learning or practice was needed to engage with logic-based bugs	1	18
Acknowledged Limitations	Learners showed self-awareness by noting their own lack of proficiency or understanding. These admissions underline honesty about gaps in capability.	1	3
Analytical Gaps	Denotes errors that occurred due to missed steps or incomplete reasoning processes. This suggests an underdeveloped analytical sequence.	1	2
Calculation Confusion	Errors emerged from difficulties in creating or tracing formula-based logic. Mathematical thinking was the barrier.	1	1
Concept Misuse	Participants misapplied key Python concepts, leading to logic flaws. These misunderstandings pointed to a superficial grasp of coding constructs.	1	2
Conditional Misinterpretation	Learners misunderstood how conditionals executed. This misreading caused flawed logic paths.	1	1
Contextual Misuse	A function or logic piece was applied in the wrong context. The logic was sound, but its placement was flawed.	1	1
Knowledge Gaps	Errors resulted from lacking the foundational knowledge needed to apply logic. This code tracks missing prerequisites.	1	6
Misunderstood Logic Flow	Participants misunderstood how one part of the code affected another. These errors revealed disconnects in logic mapping.	1	1
Needs More Practice	Participant acknowledged needing repetition or further exposure to improve. Practice was seen as key to mastering logic.	1	1
Social and Collaborative Dimensions	This category highlights how syntax debugging was supported by peer explanations, shared roles, and individual strengths in collaborative settings. It includes verbal clarification and role-based task division around syntax.	1	43
Explaining syntax fixes	For verbal explanation, negotiation, or clarification of syntax fixes during collaboration	1	15
Pair support for syntax	For collaborative efforts in addressing syntax errors through shared roles or peer help.	1	13

Name	Description	Files	References
Syntax as role strength	For individuals who naturally took the lead on syntax due to confidence or skill	1	15
Strategic Role Allocation and Rotation	This category captures how apprentices strategically assigned roles and alternated them to balance mental effort, maintain engagement, and leverage individual strengths during debugging sessions.	1	47
Division of Tasks Based on Strengths	Apprentices strategically assigned responsibilities based on individual strengths or comfort zones to manage complexity and maintain focus.	1	22
Driver-Navigator Role Sharing	Apprentices adopted a structured pairing model where one coded while the other observed and guided, helping distribute cognitive demands.	1	18
Turn-Taking and Role Swapping	Regular role alternation ensured balanced mental effort, reduced fatigue, and kept both apprentices engaged throughout the debugging session.	1	7
Strategies and Reasoning Approaches	This category includes structured problem-solving strategies like breaking problems down, isolating faulty logic, following data flow, using tool support, and experimenting methodically with solutions.	2	28
Analytical approach	Describes instances where a participant employed step-by-step reasoning or formal techniques. It highlights a structured way of unravelling complex logic.	1	1
Analytical Demand	Reflects how some debugging tasks required high-level reasoning and mental exertion. Participants perceived the activity as cognitively intensive.	1	1
Big picture review	Participants referred to stepping back and reassessing the entire codebase. This top-down perspective helped in recontextualising the issue.	1	1
Breakdown strategy	Refers to the act of deconstructing a problem into simpler parts to aid resolution. Learners discussed breaking logic into manageable pieces.	1	1
Code Isolation Strategy	Participants isolated specific blocks or lines of code to test or observe behaviour. This strategy helped to narrow the problem area.	1	2
Code Visibility Advantage	Clarity in formatting, naming, or organisation made it easier to follow the logic. Participants attributed their success partly to how readable the code was.	1	1
Data Flow Understanding	Focused on tracking how information moved through variables and functions. This tracking helped diagnose where logic broke down.	1	1

Name	Description	Files	References
Deep Dive Debugging	Marked by a thorough and prolonged engagement with the problem. Participants drilled deep into the logic layer rather than skimming.	1	1
Deep Logic	This reflects the intellectual depth required to trace and correct logic faults rooted in Python intricacies or conceptual frameworks. It signifies scenarios where surface-level knowledge was insufficient.	1	2
Divide and conquer	The issue was resolved by segmenting it into independent subproblems. Learners described resolving each part methodically.	2	3
Methodical Problem Solving	The participant used a structured, procedural approach to identify the issue. This code praises disciplined debugging.	1	1
Pattern-based reasoning	Participants applied familiar logic patterns to solve new problems. This indicates transfer of learning.	1	1
Rubber ducking	Participants verbalised logic step-by-step, often to a peer or non-technical object. This externalisation clarified their thinking.	1	2
Solution experimentation	Debugging involved trying multiple possible solutions to test logic. Learners described trial as a deliberate tactic.	1	1
Strategy Limitations	Existing methods or plans failed to resolve the logic issue. Learners were forced to reconsider their approach.	1	1
Tool-assisted logic check	Software tools like debuggers or linters were used to identify logic faults. Participants credited these for catching errors.	1	2
Trial and Error Limits	Participants noted that random guessing was ineffective for logic bugs. These problems needed deeper thought.	1	1
Understanding Logic Flow	A clear picture of how logic moved through code aided debugging. This awareness streamlined troubleshooting.	1	5
Strategies for Syntax Debugging	This category reflects tactical responses to syntax errors, such as starting with syntax checks, using tools like error highlighting, and applying pattern-recognition techniques to spot errors.	1	33
Fixing Syntax Errors	For comments about actively identifying, correcting, or guiding others through syntax issues.	1	9
Pattern-based syntax strategy	For use of recurring patterns, visual tracing, or structured methods in spotting syntax issues.	1	6

Name	Description	Files	References
Syntax First	Where participants mention syntax errors as their starting point in debugging.	1	3
Tool-Assisted Syntax Fix	For use of features like syntax highlighting, error popups, or debuggers to spot/fix syntax.	1	15
Systematic Reasoning Strategies	This category includes logical and structured approaches where learners followed data flow, stepped through execution, or broke problems into smaller parts to locate and address issues in a focused, disciplined manner.	2	46
Divide and conquer	The issue was resolved by segmenting it into independent subproblems. Learners described resolving each part methodically.	2	3
IDE Debugger Usage	Apprentices used the IDE debugger to step through code and inspect variable states, enabling precise identification and correction of logic errors.	1	35
Methodical Problem Solving	The participant used a structured, procedural approach to identify the issue. This code praises disciplined debugging.	1	1
Tracing	Apprentices followed the program's flow line by line to understand how data moved and identify where the logic broke down.	1	7
Tactical Exploration of Faults	This category covers exploratory tactics where apprentices relied on recognition of error patterns, code segmentation, and hypothesis-testing to find and resolve bugs.	2	20
Pattern Matching	Apprentices looked for recurring structures or familiar error patterns to quickly locate and fix bugs based on previous experience.	1	3
Slicing	Slicing refers to the strategy of breaking down or isolating specific segments of code—such as functions, conditions, or loops—to analyse them independently. This helps apprentices reduce complexity by focusing only on the relevant part of the code where the error is suspected, making it easier to locate and fix bugs collaboratively.	1	3
Solution experimentation	Debugging involved trying multiple possible solutions to test logic. Learners described trial as a deliberate tactic.	1	1
Tinkering	Apprentices intuitively made small code changes and tested their effects as a way to explore and understand bugs.	1	3
Trial and Error	Apprentices experimented with different solutions without a predefined plan to see what resolved the issue through observation.	1	10
Tool Limitations in Remote Setup	This category captures tool-based challenges such as lag, syncing issues, limited shared control, or edit conflicts—each of which	1	41

Name	Description	Files	References
	disrupted the flow of joint work and required additional coordination.		
Technology-Related Collaboration Issues	These quotes reflect how tool-based issues like syncing, lag, edit conflicts, and IDE limitations disrupted collaboration	1	41
Tool Setup and Accessibility Barriers	These are initial or recurring issues in configuring, accessing, or understanding how to use necessary tools like IDEs, version control, or communication platforms.	1	26
Tool Access or Setup Issues	These quotes relate to initial difficulties in using or setting up collaboration tools.	1	26
Types and Causes of Runtime Errors	This category identifies the nature of runtime errors apprentices faced, such as infinite loops, undefined variables, and type conversion issues, all of which occurred during execution.	1	35
Execution Flow	For difficulty understanding the order of execution in Python.	1	5
Infinite Loop	For errors involving loops that do not terminate.	1	17
Type Conversion	For issues converting between data types (e.g. string to number)	1	8
Undefined Variable	For use of variables that were not defined before use.	1	5

Codes\\Interview\\Stage 4 - Theme Review

Name	Description	Files	References
Subtheme 1 - Communication and Collaboration	The three categories here cover verbal/gestural limitations, cognitive misalignment, and tool-based disruptions, all central to remote pair debugging challenges.	1	134
Interpretation and Understanding Conflicts	This category includes moments where apprentices interpreted instructions, logic, or errors differently, resulting in delays or confusion in collaborative debugging.	1	33
Misalignment in Understanding	These reflect how apprentices experienced confusion or divergent interpretations of logic or instructions during debugging sessions	1	33
Remote Expression Challenges	This category captures how apprentices struggled to communicate ideas clearly without the benefit of facial expressions, gestures, or in-person context—leading to misunderstandings, over-explaining, or extra effort in articulation.	1	60
Difficulty Conveying Thought Process Remotely	These quotes are focused on how apprentices struggled to explain, align, or communicate their reasoning without face-to-face interaction	1	27
Lack of Non-Verbal Feedback	These reflect how the absence of visual, gestural, or facial cues hindered effective communication and understanding during remote debugging	1	33
Tool Limitations in Remote Setup	This category captures tool-based challenges such as lag, syncing issues, limited shared control, or edit conflicts, each of which disrupted the flow of joint work and required additional coordination.	1	41
Technology-Related Collaboration Issues	These quotes reflect how tool-based issues like syncing, lag, edit conflicts, and IDE limitations disrupted collaboration	1	41
Subtheme 1 - Syntax Error	All four categories focus on different angles of syntax-related issues: where they come from, how hard they feel, how they are tackled, and how collaboration supports resolution. Clear boundaries and internal coherence are maintained.	1	154
Nature and Source of Syntax Errors	This category captures specific causes of syntax errors such as missing colons, structural mistakes, typographical errors, and overlooked elements. It reflects how apprentices encountered surface-level mistakes that disrupted code execution.	1	45

Name	Description	Files	References
Code Structure	For syntax issues relating to structural formatting or layout.	1	4
Missing Colon	For specific mention of missing colons in syntax.	1	1
Missing Syntax	For syntax errors due to missing elements like colons, brackets, or forgotten components.	1	39
Syntax Typo	For errors caused by typographical mistakes.	1	1
Perceived Difficulty of Syntax	This category includes apprentice's perceptions of syntax errors as either easy or deceptively tricky. Some found them manageable due to clear error messages, while others initially underestimated their complexity.	1	33
Syntax Feels Easy	For those who found syntax errors more straightforward or gained confidence resolving them	1	25
Syntax for Beginners	For beginner-level ease, familiarity, or exposure to syntax debugging.	1	2
Syntax is Tricky	For quotes that describe syntax errors as deceptively hard or initially difficult to handle.	1	6
Social and Collaborative Dimensions	This category highlights how syntax debugging was supported by peer explanations, shared roles, and individual strengths in collaborative settings. It includes verbal clarification and role-based task division around syntax.	1	43
Explaining syntax fixes	For verbal explanation, negotiation, or clarification of syntax fixes during collaboration	1	15
Pair support for syntax	For collaborative efforts in addressing syntax errors through shared roles or peer help.	1	13
Syntax as role strength	For individuals who naturally took the lead on syntax due to confidence or skill	1	15
Strategies for Syntax Debugging	This category reflects tactical responses to syntax errors, such as starting with syntax checks, using tools like error highlighting, and applying pattern-recognition techniques to spot errors.	1	33
Fixing Syntax Errors	For comments about actively identifying, correcting, or guiding others through syntax issues.	1	9
Pattern-based syntax strategy	For use of recurring patterns, visual tracing, or structured methods in spotting syntax issues.	1	6
Syntax First	Where participants mention syntax errors as their starting point in debugging.	1	3
Tool-Assisted Syntax Fix	For use of features like syntax highlighting, error popups, or debuggers to spot/fix syntax.	1	15

Name	Description	Files	References
Subtheme 1 - Technology Utilisation	This subtheme effectively distinguishes between execution tools, interface features, and collaborative platforms, which are all vital to apprentice debugging.	1	77
Debugging Tools and Execution Support	This category captures how apprentices leveraged key debugging tools—such as the IDE's step-through functionality and print statements—to inspect program execution, monitor variable states, and detect logical or runtime issues.	1	67
IDE Debugger Usage	Apprentices used the IDE debugger to step through code and inspect variable states, enabling precise identification and correction of logic errors.	1	35
Print Statement Debugging	Apprentices inserted print statements and monitored console outputs to trace program behaviour and identify bugs during execution.	1	32
Interface Guidance and Visual Feedback	This category represents the supportive role of the IDE's user interface elements, including syntax highlighting, auto-suggestions, version control, and intelligent prompts, which helped learners identify errors and streamline their workflow.	1	6
Other IDE Features	Apprentices benefited from additional IDE tools like version control integration, intelligent suggestions, and code completion to streamline their workflow.	1	3
Syntax and Error Highlighting Features	The IDE's syntax highlighting, auto-indentation, and inline error notifications supported apprentices in quickly spotting and correcting code mistakes.	1	3
Real-Time Collaborative Platforms	This category includes tools that enabled synchronous work across distances, allowing apprentices to co-edit, share, and coordinate their debugging efforts in real time using platforms like Visual Studio Live Share.	1	4
Real-Time Code Sharing and Synchronisation	Tools enabling simultaneous editing and shared visibility helped apprentices maintain alignment and coordinate debugging in real-time.	1	2
Use of Live Share for Remote Collaboration	Apprentices leveraged Live Share to collaboratively edit, navigate, and debug code from separate locations in real time.	1	2
Subtheme 2 - Debugging Strategies & Tactics	These categories cover a wide tactical spectrum, from exploratory to highly methodical, providing a balanced insight into apprentice strategies.	2	86
Collaborative and Reflective Techniques	This category highlights methods where apprentices explained or reviewed code with others (or to themselves) to gain insight, clarify	2	20

Name	Description	Files	References
	thinking, and identify errors through reflection or external feedback.		
Code Review	Apprentices systematically examined and critiqued each other's code to identify issues, clarify logic, and enhance collaborative problem-solving.	1	12
Peer Review Strength	Logic errors were identified through peer feedback. Review mechanisms improved accuracy.	1	1
Rubber Duck Debugging	Apprentices explained their code aloud, to a partner or inanimate object, to clarify their thinking and uncover logic errors.	1	7
Systematic Reasoning Strategies	This category includes logical and structured approaches where learners followed data flow, stepped through execution, or broke problems into smaller parts to locate and address issues in a focused, disciplined manner.	2	46
Divide and conquer	The issue was resolved by segmenting it into independent subproblems. Learners described resolving each part methodically.	2	3
IDE Debugger Usage	Apprentices used the IDE debugger to step through code and inspect variable states, enabling precise identification and correction of logic errors.	1	35
Methodical Problem Solving	The participant used a structured, procedural approach to identify the issue. This code praises disciplined debugging.	1	1
Tracing	Apprentices followed the program's flow line by line to understand how data moved and identify where the logic broke down.	1	7
Tactical Exploration of Faults	This category covers exploratory tactics where apprentices relied on recognition of error patterns, code segmentation, and hypothesis-testing to find and resolve bugs.	2	20
Pattern Matching	Apprentices looked for recurring structures or familiar error patterns to quickly locate and fix bugs based on previous experience.	1	3
Slicing	Slicing refers to the strategy of breaking down or isolating specific segments of code—such as functions, conditions, or loops—to analyse them independently. This helps apprentices reduce complexity by focusing only on the relevant part of the code where the error is suspected, making it easier to locate and fix bugs collaboratively.	1	3
Solution experimentation	Debugging involved trying multiple possible solutions to test logic. Learners described trial as a deliberate tactic.	1	1

Name	Description	Files	References
Tinkering	Apprentices intuitively made small code changes and tested their effects as a way to explore and understand bugs.	1	3
Trial and Error	Apprentices experimented with different solutions without a predefined plan to see what resolved the issue through observation.	1	10
Subtheme 2 - Logical Error	Logical error subthemes contain a rich set of categories. These reflect the complexity of the issues and the ways learners engaged intellectually, emotionally, and socially. Internal logic and external differentiation are intact.	2	131
Collaboration and Communication Aids	This category reflects the value of teamwork in resolving logical errors. It includes quotes where learners gained clarity or found solutions by explaining to peers, switching roles, or combining their strengths.	1	13
Collaborative clarity	Understanding emerged more clearly through discussions with peers. Explaining logic to others often led to personal insight.	1	3
Collaborative Insight	New interpretations or corrections were achieved by engaging with someone else's viewpoint. The collaboration brought forth alternative solutions.	1	4
Navigator insight	In pair programming, the navigator offered a useful perspective. The insight usually shifted the course of debugging.	1	1
Paired Strengths	Learners described how teammates complemented their skills. Their collective effort covered individual weaknesses.	1	1
Peer Review Strength	Logic errors were identified through peer feedback. Review mechanisms improved accuracy.	1	1
Peer support	Emotional or technical encouragement came from fellow learners. It acted as a buffer during challenging moments.	1	1
Role swapping	Team members changed roles mid-task to better tackle logic issues. The switch brought fresh perspective.	1	2
Perceived Complexity and Emotional Response	This category captures how apprentices found logical errors emotionally or cognitively taxing. It includes expressions of frustration, struggle, or general difficulty in making sense of complex conditions or flows.	1	41
Agreement on Challenge	Multiple participants expressed a shared view that certain logical errors posed significant difficulty. The consistency in their sentiments adds weight to the issue's complexity.	1	5

Name	Description	Files	References
Complex Logic Breakdown	Learners attempted to untangle highly intricate or nested conditions. The difficulty lay not in syntax but in logical architecture.	1	4
Error Complexity	The nature of the logic error was itself intricate and multi-layered. These were not beginner mistakes but advanced logic misfires.	1	10
Flow Confusion	The challenge stemmed from not understanding how code progressed during execution. This lack of clarity hampered logical deductions.	1	1
Found It Challenging	A general admission that the task was tough, without further detail. These expressions still signal cognitive overload.	1	7
General Complexity	Applied when logic problems were described as difficult but without specific explanation. It captures vague but valid struggle.	1	5
Logic Struggle	Captures moments of emotional or cognitive difficulty expressed by learners tackling logical bugs. Participants voiced frustration and mental fatigue in trying to make sense of such errors.	1	4
Logical Errors Challenging	Serves as a general label for statements identifying logic bugs as hard. It doesn't specify which part was problematic.	1	5
Reflection and Learning Dispositions	This category captures reflective mindsets where apprentices identified growth, perseverance, or learning from mistakes. It also houses observations about how experience, context, and collaboration influenced their progress.	1	31
Context understanding	Problem-solving success relied on grasping the wider function or scenario. The learner needed to understand not just 'what' but 'why'.	1	3
Experience Builds Mastery	Learners acknowledged that repeated exposure helped them improve. Experience was credited as a major enabler of logical reasoning.	1	2
Growth Mindset	Participants expressed confidence that they could learn and improve with effort. This forward-thinking attitude supports resilience.	1	1
Growth Through Challenge	Struggle was reframed as an opportunity for learning. Participants reflected positively on the difficulty.	1	4
Logic and Flow Challenges	Combined challenges in understanding both the logic and how it executed. These situations involved overlapping difficulties.	1	6
Misuse of 'name'	Highlights confusion around Python's special '__name__' variable. This is a specific example of concept misunderstanding.	1	1

Name	Description	Files	References
Progress Despite Errors	Learners recognised forward movement even when mistakes occurred. This shows perseverance.	1	1
Remote Collaboration Limits	The online or distant setup introduced difficulties in understanding logic. Distance added barriers to debugging.	1	8
Runtime Contrast	Participants reflected on how runtime errors differed from logic ones. This comparative insight helped focus their approach.	1	2
Unresolved Logic Issues	Some logic problems remained unsolved by session end. The code captures lingering confusion.	1	2
Variable misuse	Mistakes occurred due to inappropriate variable assignment or tracking. This led to faulty logic.	1	1
Skill Gaps and Cognitive Limitations	This category includes participants' admissions of limited knowledge, misunderstood logic, or conceptual misapplications. It highlights areas where deeper learning or practice was needed to engage with logic-based bugs	1	18
Acknowledged Limitations	Learners showed self-awareness by noting their own lack of proficiency or understanding. These admissions underline honesty about gaps in capability.	1	3
Analytical Gaps	Denotes errors that occurred due to missed steps or incomplete reasoning processes. This suggests an underdeveloped analytical sequence.	1	2
Calculation Confusion	Errors emerged from difficulties in creating or tracing formula-based logic. Mathematical thinking was the barrier.	1	1
Concept Misuse	Participants misapplied key Python concepts, leading to logic flaws. These misunderstandings pointed to a superficial grasp of coding constructs.	1	2
Conditional Misinterpretation	Learners misunderstood how conditionals executed. This misreading caused flawed logic paths.	1	1
Contextual Misuse	A function or logic piece was applied in the wrong context. The logic was sound, but its placement was flawed.	1	1
Knowledge Gaps	Errors resulted from lacking the foundational knowledge needed to apply logic. This code tracks missing prerequisites.	1	6
Misunderstood Logic Flow	Participants misunderstood how one part of the code affected another. These errors revealed disconnects in logic mapping.	1	1

Name	Description	Files	References
Needs More Practice	Participant acknowledged needing repetition or further exposure to improve. Practice was seen as key to mastering logic.	1	1
Strategies and Reasoning Approaches	This category includes structured problem-solving strategies like breaking problems down, isolating faulty logic, following data flow, using tool support, and experimenting methodically with solutions.	2	28
Analytical approach	Describes instances where a participant employed step-by-step reasoning or formal techniques. It highlights a structured way of unravelling complex logic.	1	1
Analytical Demand	Reflects how some debugging tasks required high-level reasoning and mental exertion. Participants perceived the activity as cognitively intensive.	1	1
Big picture review	Participants referred to stepping back and reassessing the entire codebase. This top-down perspective helped in recontextualising the issue.	1	1
Breakdown strategy	Refers to the act of deconstructing a problem into simpler parts to aid resolution. Learners discussed breaking logic into manageable pieces.	1	1
Code Isolation Strategy	Participants isolated specific blocks or lines of code to test or observe behaviour. This strategy helped to narrow the problem area.	1	2
Code Visibility Advantage	Clarity in formatting, naming, or organisation made it easier to follow the logic. Participants attributed their success partly to how readable the code was.	1	1
Data Flow Understanding	Focused on tracking how information moved through variables and functions. This tracking helped diagnose where logic broke down.	1	1
Deep Dive Debugging	Marked by a thorough and prolonged engagement with the problem. Participants drilled deep into the logic layer rather than skimming.	1	1
Deep Logic	This reflects the intellectual depth required to trace and correct logic faults rooted in Python intricacies or conceptual frameworks. It signifies scenarios where surface-level knowledge was insufficient.	1	2
Divide and conquer	The issue was resolved by segmenting it into independent subproblems. Learners described resolving each part methodically.	2	3

Name	Description	Files	References
Methodical Problem Solving	The participant used a structured, procedural approach to identify the issue. This code praises disciplined debugging.	1	1
Pattern-based reasoning	Participants applied familiar logic patterns to solve new problems. This indicates transfer of learning.	1	1
Rubber ducking	Participants verbalised logic step-by-step, often to a peer or non-technical object. This externalisation clarified their thinking.	1	2
Solution experimentation	Debugging involved trying multiple possible solutions to test logic. Learners described trial as a deliberate tactic.	1	1
Strategy Limitations	Existing methods or plans failed to resolve the logic issue. Learners were forced to reconsider their approach.	1	1
Tool-assisted logic check	Software tools like debuggers or linters were used to identify logic faults. Participants credited these for catching errors.	1	2
Trial and Error Limits	Participants noted that random guessing was ineffective for logic bugs. These problems needed deeper thought.	1	1
Understanding Logic Flow	A clear picture of how logic moved through code aided debugging. This awareness streamlined troubleshooting.	1	5
Subtheme 2 - Technical & Cognitive	The categories under this subtheme clearly reflect technical gaps, cognitive hurdles, and runtime-specific complications.	1	273
Foundational Knowledge Gaps	This category reflects difficulties stemming from apprentices' limited prior experience with Python or programming generally, especially in handling syntax-specific rules like indentation and punctuation.	1	166
Limited Experience and Skill Gaps	Apprentices expressed difficulty navigating debugging tasks due to being new to programming, lacking foundational knowledge, or still developing confidence in applying core concepts.	1	15
Syntax Complexity	Participants frequently encountered syntax errors that disrupted code execution, especially those involving Python-specific rules like indentation, string formatting, or punctuation.	1	151
Reasoning and Logic Challenges	This category houses instances where apprentices lacked the cognitive strategies to understand and apply logic effectively, especially when dealing with conditionals, calculations, or the broader logic structure.	1	61
Logical Reasoning Gaps	Participants struggled with understanding or applying correct logic within the code,	1	61

Name	Description	Files	References
	particularly when handling conditionals, calculations, or the flow of decision-making.		
Runtime Behaviour Confusion	This category represents the difficulty of identifying and resolving bugs that only emerged during code execution, particularly where program behaviour was unpredictable or misunderstood.	1	46
Runtime Error Complexity	Several apprentices found runtime errors difficult to resolve because they often appeared after code execution and required understanding how the program behaved dynamically.	1	46
Subtheme 3 - Cognitive Load Sharing	This subtheme is compact but insightful. It cleanly separates structural role-based tactics from communication-based cognitive coordination.	1	105
Real-Time Communication and Coordination Tools	This category reflects how verbalisation strategies and collaborative digital tools were used to coordinate thought processes, reduce confusion, and manage the cognitive load during remote or paired debugging.	1	58
Real-Time Tool Support for Coordination	Collaborative tools like IDE features and remote sharing platforms were used to support synchronised thinking and reduce mental strain.	1	37
Think-Aloud Communication	Apprentices verbalised their reasoning and thought processes to clarify understanding and collaboratively work through errors.	1	21
Strategic Role Allocation and Rotation	This category captures how apprentices strategically assigned roles and alternated them to balance mental effort, maintain engagement, and leverage individual strengths during debugging sessions.	1	47
Division of Tasks Based on Strengths	Apprentices strategically assigned responsibilities based on individual strengths or comfort zones to manage complexity and maintain focus.	1	22
Driver-Navigator Role Sharing	Apprentices adopted a structured pairing model where one coded while the other observed and guided, helping distribute cognitive demands.	1	18
Turn-Taking and Role Swapping	Regular role alternation ensured balanced mental effort, reduced fatigue, and kept both apprentices engaged throughout the debugging session.	1	7
Subtheme 3 - Environmental and Logistics	These categories are well-bounded, non-overlapping, and together provide a complete view of non-technical barriers affecting collaboration and productivity.	1	49

Name	Description	Files	References
Connectivity Constraints	This category reflects how unstable internet disrupted communication, tool access, and real-time collaboration—especially in remote or bandwidth-limited settings.	1	1
Unstable or Inconsistent Internet Connection	These quotes highlight issues caused by poor or unstable internet, affecting real-time collaboration or access to tools.	1	1
Distractions and Focus Challenges	Distractions and Focus Challenges - This includes issues with concentration due to noise, interruptions, or other environmental factors unique to working remotely from home or other informal settings	1	6
Distraction in Individual Work Environments	These highlight challenges in focus due to remote, uncontrolled environments.	1	6
Physical Separation Barriers	This category refers to the challenge of not being able to visually point to code or easily clarify issues due to being physically apart during remote pair programming.	1	11
Absence of Physical Presence for Quick Clarification	These represent struggles in not being able to point or visually show parts of code	1	11
Scheduling and Coordination Hurdles	This category captures difficulties in syncing schedules across time zones or managing different availability patterns, which limited collaboration windows.	1	5
Time Zone and Scheduling Difficulties	Quotes here reflect challenges related to coordinating across different locations or schedules.	1	5
Tool Setup and Accessibility Barriers	These are initial or recurring issues in configuring, accessing, or understanding how to use necessary tools like IDEs, version control, or communication platforms.	1	26
Tool Access or Setup Issues	These quotes relate to initial difficulties in using or setting up collaboration tools.	1	26
Subtheme 3 - Runtime Error	The runtime error subtheme offers a clear focus on both the symptoms and difficulty level of errors, and learners' strategies (or lack thereof) to resolve them.	1	84
Cognitive Perception and Difficulty	This category reflects how runtime issues were perceived as overwhelming or uncertain, especially when apprentices couldn't predict behaviour or lacked confidence during execution.	1	43
Runtime Overwhelm	When runtime feels particularly complex or challenging.	1	13

Name	Description	Files	References
Runtime Type Confusion	When the issue involves converting string to number or similar.	1	6
Runtime Uncertainty	For quotes where learners are confident with syntax but unsure about runtime.	1	24
Debugging Approaches for Runtime	This category captures how apprentices attempted to resolve runtime errors, particularly by using print statements or acknowledging a lack of structured strategy.	1	6
Runtime Print Tracking	For those using print statements to trace issues.	1	4
Runtime Strategy Lacking	When trial-and-error or lack of method was highlighted.	1	2
Types and Causes of Runtime Errors	This category identifies the nature of runtime errors apprentices faced, such as infinite loops, undefined variables, and type conversion issues, all of which occurred during execution.	1	35
Execution Flow	For difficulty understanding the order of execution in Python.	1	5
Infinite Loop	For errors involving loops that do not terminate.	1	17
Type Conversion	For issues converting between data types (e.g. string to number)	1	8
Undefined Variable	For use of variables that were not defined before use.	1	5

Codes\\Interview\\Stage 5 - Theme Definition

Name	Description	Files	References
Theme 1 - Error Spectrum	This theme captures the range and types of programming errors (Syntax, Logical, and Runtime) that apprentices encountered during collaborative debugging. It highlights how these errors differ in nature, difficulty, and required problem-solving strategies.	2	369
Subtheme 1 - Syntax Error	All four categories focus on different angles of syntax-related issues: where they come from, how hard they feel, how they are tackled, and how collaboration supports resolution. Clear boundaries and internal coherence are maintained.	1	154
Nature and Source of Syntax Errors	This category captures specific causes of syntax errors such as missing colons, structural mistakes, typographical errors, and overlooked elements. It reflects how apprentices encountered surface-level mistakes that disrupted code execution.	1	45
Code Structure	For syntax issues relating to structural formatting or layout.	1	4
Missing Colon	For specific mention of missing colons in syntax.	1	1
Missing Syntax	For syntax errors due to missing elements like colons, brackets, or forgotten components.	1	39
Syntax Typo	For errors caused by typographical mistakes.	1	1
Perceived Difficulty of Syntax	This category includes apprentice's perceptions of syntax errors as either easy or deceptively tricky. Some found them manageable due to clear error messages, while others initially underestimated their complexity.	1	33
Syntax Feels Easy	For those who found syntax errors more straightforward or gained confidence resolving them	1	25
Syntax for Beginners	For beginner-level ease, familiarity, or exposure to syntax debugging.	1	2
Syntax is Tricky	For quotes that describe syntax errors as deceptively hard or initially difficult to handle.	1	6
Social and Collaborative Dimensions	This category highlights how syntax debugging was supported by peer explanations, shared roles, and individual strengths in collaborative settings. It includes	1	43

Name	Description	Files	References
	verbal clarification and role-based task division around syntax.		
Explaining syntax fixes	For verbal explanation, negotiation, or clarification of syntax fixes during collaboration	1	15
Pair support for syntax	For collaborative efforts in addressing syntax errors through shared roles or peer help.	1	13
Syntax as role strength	For individuals who naturally took the lead on syntax due to confidence or skill	1	15
Strategies for Syntax Debugging	This category reflects tactical responses to syntax errors, such as starting with syntax checks, using tools like error highlighting, and applying pattern-recognition techniques to spot errors.	1	33
Fixing Syntax Errors	For comments about actively identifying, correcting, or guiding others through syntax issues.	1	9
Pattern-based syntax strategy	For use of recurring patterns, visual tracing, or structured methods in spotting syntax issues.	1	6
Syntax First	Where participants mention syntax errors as their starting point in debugging.	1	3
Tool-Assisted Syntax Fix	For use of features like syntax highlighting, error popups, or debuggers to spot/fix syntax.	1	15
Subtheme 2 - Logical Error	Logical error subthemes contain a rich set of categories. These reflect the complexity of the issues and the ways learners engaged intellectually, emotionally, and socially. Internal logic and external differentiation are intact.	2	131
Collaboration and Communication Aids	This category reflects the value of teamwork in resolving logical errors. It includes quotes where learners gained clarity or found solutions by explaining to peers, switching roles, or combining their strengths.	1	13
Collaborative clarity	Understanding emerged more clearly through discussions with peers. Explaining logic to others often led to personal insight.	1	3
Collaborative Insight	New interpretations or corrections were achieved by engaging with someone else's viewpoint. The collaboration brought forth alternative solutions.	1	4
Navigator insight	In pair programming, the navigator offered a useful perspective. The insight usually shifted the course of debugging.	1	1

Name	Description	Files	References
Paired Strengths	Learners described how teammates complemented their skills. Their collective effort covered individual weaknesses.	1	1
Peer Review Strength	Logic errors were identified through peer feedback. Review mechanisms improved accuracy.	1	1
Peer support	Emotional or technical encouragement came from fellow learners. It acted as a buffer during challenging moments.	1	1
Role swapping	Team members changed roles mid-task to better tackle logic issues. The switch brought fresh perspective.	1	2
Perceived Complexity and Emotional Response	This category captures how apprentices found logical errors emotionally or cognitively taxing. It includes expressions of frustration, struggle, or general difficulty in making sense of complex conditions or flows.	1	41
Agreement on Challenge	Multiple participants expressed a shared view that certain logical errors posed significant difficulty. The consistency in their sentiments adds weight to the issue's complexity.	1	5
Complex Logic Breakdown	Learners attempted to untangle highly intricate or nested conditions. The difficulty lay not in syntax but in logical architecture.	1	4
Error Complexity	The nature of the logic error was itself intricate and multi-layered. These were not beginner mistakes but advanced logic misfires.	1	10
Flow Confusion	The challenge stemmed from not understanding how code progressed during execution. This lack of clarity hampered logical deductions.	1	1
Found It Challenging	A general admission that the task was tough, without further detail. These expressions still signal cognitive overload.	1	7
General Complexity	Applied when logic problems were described as difficult but without specific explanation. It captures vague but valid struggle.	1	5
Logic Struggle	Captures moments of emotional or cognitive difficulty expressed by learners tackling logical bugs. Participants voiced frustration and mental fatigue in trying to make sense of such errors.	1	4
Logical Errors Challenging	Serves as a general label for statements identifying logic bugs as hard. It doesn't specify which part was problematic.	1	5

Name	Description	Files	References
Reflection and Learning Dispositions	This category captures reflective mindsets where apprentices identified growth, perseverance, or learning from mistakes. It also houses observations about how experience, context, and collaboration influenced their progress.	1	31
Context understanding	Problem-solving success relied on grasping the wider function or scenario. The learner needed to understand not just 'what' but 'why'.	1	3
Experience Builds Mastery	Learners acknowledged that repeated exposure helped them improve. Experience was credited as a major enabler of logical reasoning.	1	2
Growth Mindset	Participants expressed confidence that they could learn and improve with effort. This forward-thinking attitude supports resilience.	1	1
Growth Through Challenge	Struggle was reframed as an opportunity for learning. Participants reflected positively on the difficulty.	1	4
Logic and Flow Challenges	Combined challenges in understanding both the logic and how it executed. These situations involved overlapping difficulties.	1	6
Misuse of 'name'	Highlights confusion around Python's special ' <code>__name__</code> ' variable. This is a specific example of concept misunderstanding.	1	1
Progress Despite Errors	Learners recognised forward movement even when mistakes occurred. This shows perseverance.	1	1
Remote Collaboration Limits	The online or distant setup introduced difficulties in understanding logic. Distance added barriers to debugging.	1	8
Runtime Contrast	Participants reflected on how runtime errors differed from logic ones. This comparative insight helped focus their approach.	1	2
Unresolved Logic Issues	Some logic problems remained unsolved by session end. The code captures lingering confusion.	1	2
Variable misuse	Mistakes occurred due to inappropriate variable assignment or tracking. This led to faulty logic.	1	1
Skill Gaps and Cognitive Limitations	This category includes participants' admissions of limited knowledge, misunderstood logic, or conceptual misapplications. It highlights areas where deeper learning or practice was needed to engage with logic-based bugs	1	18

Name	Description	Files	References
Acknowledged Limitations	Learners showed self-awareness by noting their own lack of proficiency or understanding. These admissions underline honesty about gaps in capability.	1	3
Analytical Gaps	Denotes errors that occurred due to missed steps or incomplete reasoning processes. This suggests an underdeveloped analytical sequence.	1	2
Calculation Confusion	Errors emerged from difficulties in creating or tracing formula-based logic. Mathematical thinking was the barrier.	1	1
Concept Misuse	Participants misapplied key Python concepts, leading to logic flaws. These misunderstandings pointed to a superficial grasp of coding constructs.	1	2
Conditional Misinterpretation	Learners misunderstood how conditionals executed. This misreading caused flawed logic paths.	1	1
Contextual Misuse	A function or logic piece was applied in the wrong context. The logic was sound, but its placement was flawed.	1	1
Knowledge Gaps	Errors resulted from lacking the foundational knowledge needed to apply logic. This code tracks missing prerequisites.	1	6
Misunderstood Logic Flow	Participants misunderstood how one part of the code affected another. These errors revealed disconnects in logic mapping.	1	1
Needs More Practice	Participant acknowledged needing repetition or further exposure to improve. Practice was seen as key to mastering logic.	1	1
Strategies and Reasoning Approaches	This category includes structured problem-solving strategies like breaking problems down, isolating faulty logic, following data flow, using tool support, and experimenting methodically with solutions.	2	28
Analytical approach	Describes instances where a participant employed step-by-step reasoning or formal techniques. It highlights a structured way of unravelling complex logic.	1	1
Analytical Demand	Reflects how some debugging tasks required high-level reasoning and mental exertion. Participants perceived the activity as cognitively intensive.	1	1
Big picture review	Participants referred to stepping back and reassessing the entire codebase. This top-down perspective helped in recontextualising the issue.	1	1

Name	Description	Files	References
Breakdown strategy	Refers to the act of deconstructing a problem into simpler parts to aid resolution. Learners discussed breaking logic into manageable pieces.	1	1
Code Isolation Strategy	Participants isolated specific blocks or lines of code to test or observe behaviour. This strategy helped to narrow the problem area.	1	2
Code Visibility Advantage	Clarity in formatting, naming, or organisation made it easier to follow the logic. Participants attributed their success partly to how readable the code was.	1	1
Data Flow Understanding	Focused on tracking how information moved through variables and functions. This tracking helped diagnose where logic broke down.	1	1
Deep Dive Debugging	Marked by a thorough and prolonged engagement with the problem. Participants drilled deep into the logic layer rather than skimming.	1	1
Deep Logic	This reflects the intellectual depth required to trace and correct logic faults rooted in Python intricacies or conceptual frameworks. It signifies scenarios where surface-level knowledge was insufficient.	1	2
Divide and conquer	The issue was resolved by segmenting it into independent subproblems. Learners described resolving each part methodically.	2	3
Methodical Problem Solving	The participant used a structured, procedural approach to identify the issue. This code praises disciplined debugging.	1	1
Pattern-based reasoning	Participants applied familiar logic patterns to solve new problems. This indicates transfer of learning.	1	1
Rubber ducking	Participants verbalised logic step-by-step, often to a peer or non-technical object. This externalisation clarified their thinking.	1	2
Solution experimentation	Debugging involved trying multiple possible solutions to test logic. Learners described trial as a deliberate tactic.	1	1
Strategy Limitations	Existing methods or plans failed to resolve the logic issue. Learners were forced to reconsider their approach.	1	1
Tool-assisted logic check	Software tools like debuggers or linters were used to identify logic faults. Participants credited these for catching errors.	1	2
Trial and Error Limits	Participants noted that random guessing was ineffective for logic bugs. These problems needed deeper thought.	1	1

Name	Description	Files	References
Understanding Logic Flow	A clear picture of how logic moved through code aided debugging. This awareness streamlined troubleshooting.	1	5
Subtheme 3 - Runtime Error	The runtime error subtheme offers a clear focus on both the symptoms and difficulty level of errors, and learners' strategies (or lack thereof) to resolve them.	1	84
Cognitive Perception and Difficulty	This category reflects how runtime issues were perceived as overwhelming or uncertain, especially when apprentices couldn't predict behaviour or lacked confidence during execution.	1	43
Runtime Overwhelm	When runtime feels particularly complex or challenging.	1	13
Runtime Type Confusion	When the issue involves converting string to number or similar.	1	6
Runtime Uncertainty	For quotes where learners are confident with syntax but unsure about runtime.	1	24
Debugging Approaches for Runtime	This category captures how apprentices attempted to resolve runtime errors, particularly by using print statements or acknowledging a lack of structured strategy.	1	6
Runtime Print Tracking	For those using print statements to trace issues.	1	4
Runtime Strategy Lacking	When trial-and-error or lack of method was highlighted.	1	2
Types and Causes of Runtime Errors	This category identifies the nature of runtime errors apprentices faced, such as infinite loops, undefined variables, and type conversion issues, all of which occurred during execution.	1	35
Execution Flow	For difficulty understanding the order of execution in Python.	1	5
Infinite Loop	For errors involving loops that do not terminate.	1	17
Type Conversion	For issues converting between data types (e.g. string to number)	1	8
Undefined Variable	For use of variables that were not defined before use.	1	5
Theme 2 - Technical and Cognitive Skills	This theme reflects the skills, tools, and cognitive strategies that apprentices deployed to debug effectively. It includes the use of IDE features, reasoning techniques, and collaborative planning to manage complexity and solve problems.	2	268
Subtheme 1 - Technology Utilisation	This subtheme effectively distinguishes between execution tools, interface features,	1	77

Name	Description	Files	References
	and collaborative platforms, which are all vital to apprentice debugging.		
Debugging Tools and Execution Support	This category captures how apprentices leveraged key debugging tools—such as the IDE’s step-through functionality and print statements—to inspect program execution, monitor variable states, and detect logical or runtime issues.	1	67
IDE Debugger Usage	Apprentices used the IDE debugger to step through code and inspect variable states, enabling precise identification and correction of logic errors.	1	35
Print Statement Debugging	Apprentices inserted print statements and monitored console outputs to trace program behaviour and identify bugs during execution.	1	32
Interface Guidance and Visual Feedback	This category represents the supportive role of the IDE’s user interface elements, including syntax highlighting, auto-suggestions, version control, and intelligent prompts, which helped learners identify errors and streamline their workflow.	1	6
Other IDE Features	Apprentices benefited from additional IDE tools like version control integration, intelligent suggestions, and code completion to streamline their workflow.	1	3
Syntax and Error Highlighting Features	The IDE’s syntax highlighting, auto-indentation, and inline error notifications supported apprentices in quickly spotting and correcting code mistakes.	1	3
Real-Time Collaborative Platforms	This category includes tools that enabled synchronous work across distances, allowing apprentices to co-edit, share, and coordinate their debugging efforts in real time using platforms like Visual Studio Live Share.	1	4
Real-Time Code Sharing and Synchronisation	Tools enabling simultaneous editing and shared visibility helped apprentices maintain alignment and coordinate debugging in real-time.	1	2
Use of Live Share for Remote Collaboration	Apprentices leveraged Live Share to collaboratively edit, navigate, and debug code from separate locations in real time.	1	2
Subtheme 2 - Debugging Strategies & Tactics	These categories cover a wide tactical spectrum, from exploratory to highly methodical, providing a balanced insight into apprentice strategies.	2	86
Collaborative and Reflective Techniques	This category highlights methods where apprentices explained or reviewed code with others (or to themselves) to gain insight,	2	20

Name	Description	Files	References
	clarify thinking, and identify errors through reflection or external feedback.		
Code Review	Apprentices systematically examined and critiqued each other's code to identify issues, clarify logic, and enhance collaborative problem-solving.	1	12
Peer Review Strength	Logic errors were identified through peer feedback. Review mechanisms improved accuracy.	1	1
Rubber Duck Debugging	Apprentices explained their code aloud—to a partner or inanimate object, to clarify their thinking and uncover logic errors.	1	7
Systematic Reasoning Strategies	This category includes logical and structured approaches where learners followed data flow, stepped through execution, or broke problems into smaller parts to locate and address issues in a focused, disciplined manner.	2	46
Divide and conquer	The issue was resolved by segmenting it into independent subproblems. Learners described resolving each part methodically.	2	3
IDE Debugger Usage	Apprentices used the IDE debugger to step through code and inspect variable states, enabling precise identification and correction of logic errors.	1	35
Methodical Problem Solving	The participant used a structured, procedural approach to identify the issue. This code praises disciplined debugging.	1	1
Tracing	Apprentices followed the program's flow line by line to understand how data moved and identify where the logic broke down.	1	7
Tactical Exploration of Faults	This category covers exploratory tactics where apprentices relied on recognition of error patterns, code segmentation, and hypothesis-testing to find and resolve bugs.	2	20
Pattern Matching	Apprentices looked for recurring structures or familiar error patterns to quickly locate and fix bugs based on previous experience.	1	3
Slicing	Slicing refers to the strategy of breaking down or isolating specific segments of code—such as functions, conditions, or loops—to analyse them independently. This helps apprentices reduce complexity by focusing only on the relevant part of the code where the error is suspected, making it easier to locate and fix bugs collaboratively.	1	3

Name	Description	Files	References
Solution experimentation	Debugging involved trying multiple possible solutions to test logic. Learners described trial as a deliberate tactic.	1	1
Tinkering	Apprentices intuitively made small code changes and tested their effects as a way to explore and understand bugs.	1	3
Trial and Error	Apprentices experimented with different solutions without a predefined plan to see what resolved the issue through observation.	1	10
Subtheme 3 - Cognitive Load Sharing	This subtheme is compact but insightful. It cleanly separates structural role-based tactics from communication-based cognitive coordination.	1	105
Real-Time Communication and Coordination Tools	This category reflects how verbalisation strategies and collaborative digital tools were used to coordinate thought processes, reduce confusion, and manage the cognitive load during remote or paired debugging.	1	58
Real-Time Tool Support for Coordination	Collaborative tools like IDE features and remote sharing platforms were used to support synchronised thinking and reduce mental strain.	1	37
Think-Aloud Communication	Apprentices verbalised their reasoning and thought processes to clarify understanding and collaboratively work through errors.	1	21
Strategic Role Allocation and Rotation	This category captures how apprentices strategically assigned roles and alternated them to balance mental effort, maintain engagement, and leverage individual strengths during debugging sessions.	1	47
Division of Tasks Based on Strengths	Apprentices strategically assigned responsibilities based on individual strengths or comfort zones to manage complexity and maintain focus.	1	22
Driver-Navigator Role Sharing	Apprentices adopted a structured pairing model where one coded while the other observed and guided, helping distribute cognitive demands.	1	18
Turn-Taking and Role Swapping	Regular role alternation ensured balanced mental effort, reduced fatigue, and kept both apprentices engaged throughout the debugging session.	1	7
Theme 3 - Challenges	This theme represents the barriers, limitations, and points of difficulty apprentices experienced during remote pair debugging. It includes communication constraints, technical skill gaps, and environmental disruptions affecting their workflow.	1	456

Name	Description	Files	References
Subtheme 1 - Communication and Collaboration	The three categories here cover verbal/gestural limitations, cognitive misalignment, and tool-based disruptions, all central to remote pair debugging challenges.	1	134
Interpretation and Understanding Conflicts	This category includes moments where apprentices interpreted instructions, logic, or errors differently, resulting in delays or confusion in collaborative debugging.	1	33
Misalignment in Understanding	These reflect how apprentices experienced confusion or divergent interpretations of logic or instructions during debugging sessions	1	33
Remote Expression Challenges	This category captures how apprentices struggled to communicate ideas clearly without the benefit of facial expressions, gestures, or in-person context—leading to misunderstandings, over-explaining, or extra effort in articulation.	1	60
Difficulty Conveying Thought Process Remotely	These quotes are focused on how apprentices struggled to explain, align, or communicate their reasoning without face-to-face interaction	1	27
Lack of Non-Verbal Feedback	These reflect how the absence of visual, gestural, or facial cues hindered effective communication and understanding during remote debugging	1	33
Tool Limitations in Remote Setup	This category captures tool-based challenges such as lag, syncing issues, limited shared control, or edit conflicts, each of which disrupted the flow of joint work and required additional coordination.	1	41
Technology-Related Collaboration Issues	These quotes reflect how tool-based issues like syncing, lag, edit conflicts, and IDE limitations disrupted collaboration	1	41
Subtheme 2 - Technical & Cognitive	The categories under this subtheme clearly reflect technical gaps, cognitive hurdles, and runtime-specific complications.	1	273
Foundational Knowledge Gaps	This category reflects difficulties stemming from apprentices' limited prior experience with Python or programming generally, especially in handling syntax-specific rules like indentation and punctuation.	1	166
Limited Experience and Skill Gaps	Apprentices expressed difficulty navigating debugging tasks due to being new to programming, lacking foundational knowledge, or still developing confidence in applying core concepts.	1	15

Name	Description	Files	References
Syntax Complexity	Participants frequently encountered syntax errors that disrupted code execution, especially those involving Python-specific rules like indentation, string formatting, or punctuation.	1	151
Reasoning and Logic Challenges	This category houses instances where apprentices lacked the cognitive strategies to understand and apply logic effectively, especially when dealing with conditionals, calculations, or the broader logic structure.	1	61
Logical Reasoning Gaps	Participants struggled with understanding or applying correct logic within the code, particularly when handling conditionals, calculations, or the flow of decision-making.	1	61
Runtime Behaviour Confusion	This category represents the difficulty of identifying and resolving bugs that only emerged during code execution, particularly where program behaviour was unpredictable or misunderstood.	1	46
Runtime Error Complexity	Several apprentices found runtime errors difficult to resolve because they often appeared after code execution and required understanding how the program behaved dynamically.	1	46
Subtheme 3 - Environmental and Logistics	These categories are well-bounded, non-overlapping, and together provide a complete view of non-technical barriers affecting collaboration and productivity.	1	49
Connectivity Constraints	This category reflects how unstable internet disrupted communication, tool access, and real-time collaboration, especially in remote or bandwidth-limited settings.	1	1
Unstable or Inconsistent Internet Connection	These quotes highlight issues caused by poor or unstable internet, affecting real-time collaboration or access to tools.	1	1
Distractions and Focus Challenges	Distractions and Focus Challenges This includes issues with concentration due to noise, interruptions, or other environmental factors unique to working remotely from home or other informal settings	1	6
Distraction in Individual Work Environments	These highlight challenges in focus due to remote, uncontrolled environments.	1	6
Physical Separation Barriers	This category refers to the challenge of not being able to visually point to code or easily clarify issues due to being physically apart during remote pair programming.	1	11

Name	Description	Files	References
Absence of Physical Presence for Quick Clarification	These represent struggles in not being able to point or visually show parts of code	1	11
Scheduling and Coordination Hurdles	This category captures difficulties in syncing schedules across time zones or managing different availability patterns, which limited collaboration windows.	1	5
Time Zone and Scheduling Difficulties	Quotes here reflect challenges related to coordinating across different locations or schedules.	1	5
Tool Setup and Accessibility Barriers	These are initial or recurring issues in configuring, accessing, or understanding how to use necessary tools like IDEs, version control, or communication platforms.	1	26
Tool Access or Setup Issues	These quotes relate to initial difficulties in using or setting up collaboration tools.	1	26

Appendix O: Focus Group Codebook

DPP

Codes\\Focus Group\\Stage 1 & 2 - Familiarisation & Coding

Name	Description	Files	References
Building mental models through documentation	Writing comments and diagrams helps apprentices internalise code structure.	1	1
Casual reasoning as entry point	Debugging often starts with general intuition rather than a structured plan.	1	1
Casual reasoning misses logical errors	Relying solely on intuition often overlooks deeper logical issues.	1	1
Casual reasoning reliance	Initial debugging efforts are frequently guided by intuitive rather than logical reasoning.	1	1
Challenges in tracing code	Apprentices often find it difficult to follow code execution paths.	1	1
Code complexity is overwhelming	Large and unfamiliar codebases can hinder apprentices' navigation and focus.	1	1
Confirmation bias skews debugging	Preconceived assumptions can prevent apprentices from seeing simple bugs.	1	1
Copying solutions without understanding	Replicating peer solutions without comprehension weakens problem-solving development.	1	1
Debugging depends on bug type	Strategy effectiveness depends on the complexity and category of the bug.	1	1
Debugging is influenced by learning style	Effective debugging strategies vary based on an apprentice's preferred learning style.	1	1
Debugging is shaped by collaboration quality	Team synergy significantly influences the effectiveness of joint debugging.	1	1
Debugging strategy depends on context	Approaches to debugging vary depending on the project and type of bug.	1	1
Debugging tasks build confidence incrementally	Solving increasingly complex bugs builds apprentice confidence step by step.	1	1
Difficulty segmenting code	Apprentices struggle to break complex problems into manageable parts.	1	1
Documenting aids learning	Keeping records of debugging efforts enhances long-term problem-solving skills.	1	1

Name	Description	Files	References
Effective use of top-down	A structured top-down strategy improves debugging outcomes.	1	1
Error messages mislead	Misreading error outputs can lead apprentices down unproductive paths.	1	1
Explaining debugging process	Verbalising thought processes helps apprentices arrive at solutions.	1	1
Exploring before understanding	Jumping into debugging without a plan often wastes effort.	1	1
External time pressure disrupts debugging	High-pressure scenarios often lead apprentices to abandon systematic debugging.	1	1
Feedback from code review reshapes mindset	Constructive feedback during reviews influences debugging confidence and approach.	1	1
Fixation on wrong sections	Focusing narrowly on specific code sections leads to oversight of wider issues.	1	1
Go-to use of print statements	Print statements are a preferred initial debugging method due to familiarity.	1	1
Growth in tool appreciation	Apprentices learn to value tool efficiency once they overcome initial hesitation.	1	1
Guessing based on code understanding	Apprentices rely on surface-level comprehension of code to make educated guesses about bug locations.	1	1
Hesitation to adopt tools	New users often delay embracing powerful debugging tools due to fear or lack of confidence.	1	1
Hypothesising from current code state	Apprentices form assumptions based on initial code inspection.	1	1
IDE debugger solves python bug	Learning debugger tools enabled successful resolution of Python bugs.	1	1
IDE familiarity improves speed	Familiarity with debugging environments leads to faster problem resolution.	1	1
IDE intimidation delays adoption	Fear of using debugger features prolongs reliance on basic methods.	1	1
Incomplete mental model causes missteps	Gaps in understanding program flow often lead to flawed assumptions.	1	1
Initial intimidation with IDEs	Beginners often find IDE tools overwhelming, delaying their usage.	1	1
Initial struggle with advanced tools	Advanced debugging tools pose challenges for novices.	1	1
Jumping to conclusions without testing	Making changes without verification disrupts debugging accuracy.	1	1
Lack of consistency in strategy	Frequent switching between debugging approaches without persistence hinders learning.	1	1

Name	Description	Files	References
Lack of planning leads to repeated mistakes	Without structured reflection, apprentices often repeat ineffective actions.	1	1
Learning through pair programming	Collaborative programming accelerates understanding of debugging processes.	1	1
Mastering slicing	Learning to slice code effectively improves tracing and debugging accuracy.	1	1
Mentorship via code reviews	Experienced developers guiding apprentices significantly improves their debugging effectiveness.	1	1
Not understanding broader impact	Neglecting the system-wide effects of a bug fix can cause further errors.	1	1
Overconfidence hides errors	Confidence without verification can blind apprentices to simple coding mistakes.	1	1
Pair programming exposes memory leak	Collaborative debugging quickly uncovered a memory leak issue.	1	1
Pattern matching experience	Recognising recurring error patterns assists in efficient problem solving.	1	1
Pattern matching in Python debugging	Python debugging becomes easier when apprentices identify recurring structural patterns.	1	1
Peer-led walkthroughs encourage reflection	Explaining code to peers prompts apprentices to think more critically about their logic.	1	1
Progress from prints to breakpoints	Debugging maturity shows in transitioning from print statements to advanced IDE features.	1	1
Quiet debugging helps focus	A calm, low-distraction environment enhances debugging concentration.	1	1
Regular debugging journals enhance strategy retention	Consistent journaling of bugs and fixes helps apprentices avoid repeating mistakes.	1	1
Replication supports root cause analysis	Reproducing bugs consistently helps clarify underlying causes.	1	1
Rubber duck debugging helps recursion	Explaining recursive problems aloud clarified their solution.	1	1
Rushed learning under pressure	Time constraints force apprentices to prioritise quick fixes over deeper understanding.	1	1
Sharpness through code review	Participating in code reviews sharpens bug detection skills.	1	1
Simulation of bugs helps strategy selection	Creating and solving artificial bugs allows apprentices to test and compare strategies.	1	1
Skipping small tests leads to big issues	Neglecting to test small units can result in wasted debugging time	1	1
Slicing improves isolation	Slicing enables more precise identification of fault origins in code.	1	1

Name	Description	Files	References
Step-by-step execution preferred	Structured, sequential debugging aids apprentices in isolating and resolving issues.	1	1
Structured top-down debugging	A systematic top-down method simplifies complex debugging tasks.	1	1
Structured training improves strategy use	Formal instruction in debugging techniques accelerates apprentice development.	1	1
Struggling to interpret error logs	Complex logs and stack traces are difficult for new apprentices to decode.	1	1
Success boosted by guided debugging	Step-by-step guidance in early debugging exercises improves long-term independence.	1	1
Switch to trial-and-error	When intuition fails, apprentices default to experimenting with fixes.	1	1
Testing leads to early bug discovery	Writing tests regularly helps apprentices catch bugs early.	1	1
Tool choice influenced by language	Programming language and tech stack shape the debugging approach.	1	1
Tool comfort impacts strategy	Confidence in using IDEs influences which debugging methods are applied.	1	1
Trial-and-error fits web debugging	Unstructured trial-and-error can work well in simpler web-based contexts.	1	1
Trial-and-error method	Debugging often begins with non-systematic experimentation that can be inefficient.	1	1
Understanding bug impact	Comprehending how a fix affects the whole system is crucial to effective debugging.	1	1
Unit testing enhances robustness	Implementing unit tests boosts confidence in code stability.	1	1
Use of isolation techniques	Employing isolation helps apprentices identify and address specific errors.	1	1
Visual cues in IDEs improve flow tracing	Graphical IDE features help clarify complex function flows for visual learners.	1	1
Visual debugging supports understanding	Graphical representations aid in tracing program flow and state changes.	1	1
Visual strategies for visual learners	Visual tools like debuggers support those with visual learning preferences.	1	1

Codes\\Focus Group\\Stage 3 - Theme Generation

Name	Description	Files	References
Bug Replication	Focuses on the importance of recreating bugs as a strategic practice to understand error behaviour and trace root causes effectively.	1	2
Replication supports root cause analysis	Reproducing bugs consistently helps clarify underlying causes.	1	1
Rushed learning under pressure	Time constraints force apprentices to prioritise quick fixes over deeper understanding.	1	1
Cognitive Load	Describes the mental burden apprentices experience when managing multiple elements of code logic, often leading to overwhelm or errors in reasoning.	2	8
Challenges in tracing code	Apprentices often find it difficult to follow code execution paths.	1	1
Code complexity is overwhelming	Large and unfamiliar codebases can hinder apprentices' navigation and focus.	1	1
Difficulty segmenting code	Apprentices struggle to break complex problems into manageable parts.	1	1
Fixation on wrong sections	Focusing narrowly on specific code sections leads to oversight of wider issues.	1	1
Incomplete mental model causes missteps	Gaps in understanding program flow often lead to flawed assumptions.	1	1
Not understanding broader impact	Neglecting the system-wide effects of a bug fix can cause further errors.	1	1
Overconfidence hides errors	Confidence without verification can blind apprentices to simple coding mistakes.	1	1
Understanding bug impact	Comprehending how a fix affects the whole system is crucial to effective debugging.	1	1
Collaborative Learning	Captures how shared thinking, verbalisation, and peer interaction during debugging foster deeper understanding and strategic refinement.	2	4
Explaining debugging process	Verbalising thought processes helps apprentices arrive at solutions.	1	1
Feedback from code review reshapes mindset	Constructive feedback during reviews influences debugging confidence and approach.	1	1
Mentorship via code reviews	Experienced developers guiding apprentices significantly improves their debugging effectiveness.	1	1
Sharpness through code review	Participating in code reviews sharpens bug detection skills.	1	1

Name	Description	Files	References
Debugging Strategy Selection	Refers to the gradual, often scaffolded, acquisition of debugging expertise and confidence through repeated exposure to increasingly complex tasks.	2	4
Debugging depends on bug type	Strategy effectiveness depends on the complexity and category of the bug.	1	1
Debugging strategy depends on context	Approaches to debugging vary depending on the project and type of bug.	1	1
Simulation of bugs helps strategy selection	Creating and solving artificial bugs allows apprentices to test and compare strategies.	1	1
Tool choice influenced by language	Programming language and tech stack shape the debugging approach.	1	1
Environment Factors	Considers how quiet spaces, distractions, or time pressures in the learning or work setting either support or hinder focused debugging.	1	2
External time pressure disrupts debugging	High-pressure scenarios often lead apprentices to abandon systematic debugging.	1	1
Quiet debugging helps focus	A calm, low-distraction environment enhances debugging concentration.	1	1
Error Interpretation	Highlights how apprentices read, misread, or apply meaning to error messages and logs, which directly affects bug diagnosis and resolution pathways.	2	2
Error messages mislead	Misreading error outputs can lead apprentices down unproductive paths.	1	1
Struggling to interpret error logs	Complex logs and stack traces are difficult for new apprentices to decode.	1	1
General Reasoning Pattern	Describes the common tendency of apprentices to begin debugging using informal, intuitive, or surface-level logic rather than structured analytical methods.	1	1
Exploring before understanding	Jumping into debugging without a plan often wastes effort.	1	1
Knowledge Development	Refers to the gradual, often scaffolded, acquisition of debugging expertise and confidence through repeated exposure to increasingly complex tasks.	2	3
Building mental models through documentation	Writing comments and diagrams helps apprentices internalise code structure.	1	1
Documenting aids learning	Keeping records of debugging efforts enhances long-term problem-solving skills.	1	1

Name	Description	Files	References
Regular debugging journals enhance strategy retention	Consistent journaling of bugs and fixes helps apprentices avoid repeating mistakes.	1	1
Learning Style Influence	Recognises that visual, verbal, and hands-on learners respond differently to debugging strategies, influencing their performance and tool preferences.	1	2
Debugging is influenced by learning style	Effective debugging strategies vary based on an apprentice's preferred learning style.	1	1
Visual strategies for visual learners	Visual tools like debuggers support those with visual learning preferences.	1	1
Misguided Assumptions	Refers to errors in debugging that stem from overconfidence, confirmation bias, or reliance on incorrect prior beliefs about how the code should behave.	1	6
Confirmation bias skews debugging	Preconceived assumptions can prevent apprentices from seeing simple bugs.	1	1
Copying solutions without understanding	Replicating peer solutions without comprehension weakens problem-solving development.	1	1
Jumping to conclusions without testing	Making changes without verification disrupts debugging accuracy.	1	1
Lack of consistency in strategy	Frequent switching between debugging approaches without persistence hinders learning.	1	1
Lack of planning leads to repeated mistakes	Without structured reflection, apprentices often repeat ineffective actions.	1	1
Skipping small tests leads to big issues	Neglecting to test small units can result in wasted debugging time	1	1
Peer Collaboration	Encompasses the positive effects of paired debugging, walkthroughs, code reviews, and feedback from peers or mentors in scaffolding problem-solving.	1	4
Learning through pair programming	Collaborative programming accelerates understanding of debugging processes.	1	1
Pair programming exposes memory leak	Collaborative debugging quickly uncovered a memory leak issue.	1	1
Peer-led walkthroughs encourage reflection	Explaining code to peers prompts apprentices to think more critically about their logic.	1	1
Rubber duck debugging helps recursion	Explaining recursive problems aloud clarified their solution.	1	1

Name	Description	Files	References
Strategy Consolidation	Reflects the apprentices' ability to retain, refine, and reflect on effective debugging techniques through habits like journaling and documentation.	2	3
Debugging tasks build confidence incrementally	Solving increasingly complex bugs builds apprentice confidence step by step.	1	1
Structured training improves strategy use	Formal instruction in debugging techniques accelerates apprentice development.	1	1
Success boosted by guided debugging	Step-by-step guidance in early debugging exercises improves long-term independence.	1	1
Structured Strategy	Involves the deliberate use of planned debugging approaches like slicing, top-down decomposition, and test-driven methods that promote efficiency and clarity.	2	10
Effective use of top-down	A structured top-down strategy improves debugging outcomes.	1	1
Mastering slicing	Learning to slice code effectively improves tracing and debugging accuracy.	1	1
Pattern matching experience	Recognising recurring error patterns assists in efficient problem solving.	1	1
Pattern matching in Python debugging	Python debugging becomes easier when apprentices identify recurring structural patterns.	1	1
Slicing improves isolation	Slicing enables more precise identification of fault origins in code.	1	1
Step-by-step execution preferred	Structured, sequential debugging aids apprentices in isolating and resolving issues.	1	1
Structured top-down debugging	A systematic top-down method simplifies complex debugging tasks.	1	1
Testing leads to early bug discovery	Writing tests regularly helps apprentices catch bugs early.	1	1
Unit testing enhances robustness	Implementing unit tests boosts confidence in code stability.	1	1
Use of isolation techniques	Employing isolation helps apprentices identify and address specific errors.	1	1
Tool Familiarity	Refers to how apprentices' comfort, exposure, and understanding of debugging tools—especially IDEs—affect their willingness and effectiveness in using them.	2	12
Go-to use of print statements	Print statements are a preferred initial debugging method due to familiarity.	1	1
Growth in tool appreciation	Apprentices learn to value tool efficiency once they overcome initial hesitation.	1	1

Name	Description	Files	References
Hesitation to adopt tools	New users often delay embracing powerful debugging tools due to fear or lack of confidence.	1	1
IDE debugger solves python bug	Learning debugger tools enabled successful resolution of Python bugs.	1	1
IDE familiarity improves speed	Familiarity with debugging environments leads to faster problem resolution.	1	1
IDE intimidation delays adoption	Fear of using debugger features prolongs reliance on basic methods.	1	1
Initial intimidation with IDEs	Beginners often find IDE tools overwhelming, delaying their usage.	1	1
Initial struggle with advanced tools	Advanced debugging tools pose challenges for novices.	1	1
Progress from prints to breakpoints	Debugging maturity shows in transitioning from print statements to advanced IDE features.	1	1
Tool comfort impacts strategy	Confidence in using IDEs influences which debugging methods are applied.	1	1
Visual cues in IDEs improve flow tracing	Graphical IDE features help clarify complex function flows for visual learners.	1	1
Visual debugging supports understanding	Graphical representations aid in tracing program flow and state changes.	1	1
Unstructured Debugging	Captures instances where apprentices use inconsistent, reactive, or haphazard debugging tactics without a coherent plan, often resulting in inefficiency.	2	8
Casual reasoning as entry point	Debugging often starts with general intuition rather than a structured plan.	1	1
Casual reasoning misses logical errors	Relying solely on intuition often overlooks deeper logical issues.	1	1
Casual reasoning reliance	Initial debugging efforts are frequently guided by intuitive rather than logical reasoning.	1	1
Guessing based on code understanding	Apprentices rely on surface-level comprehension of code to make educated guesses about bug locations.	1	1
Hypothesising from current code state	Apprentices form assumptions based on initial code inspection.	1	1
Switch to trial-and-error	When intuition fails, apprentices default to experimenting with fixes.	1	1
Trial-and-error fits web debugging	Unstructured trial-and-error can work well in simpler web-based contexts.	1	1
Trial-and-error method	Debugging often begins with non-systematic experimentation that can be inefficient.	1	1

Codes\\Focus Group\\Stage 4 - Theme Review

Name	Description	Files	References
Subtheme 1.1 – Debugging Mindsets and Reasoning Patterns	These reflect the intuitive, non-systematic approaches apprentices use, and common mistakes made due to overconfidence, assumption, or incomplete understanding.	2	15
General Reasoning Pattern	Describes the common tendency of apprentices to begin debugging using informal, intuitive, or surface-level logic rather than structured analytical methods.	1	1
Exploring before understanding	Jumping into debugging without a plan often wastes effort.	1	1
Misguided Assumptions	Refers to errors in debugging that stem from overconfidence, confirmation bias, or reliance on incorrect prior beliefs about how the code should behave.	1	6
Confirmation bias skews debugging	Preconceived assumptions can prevent apprentices from seeing simple bugs.	1	1
Copying solutions without understanding	Replicating peer solutions without comprehension weakens problem-solving development.	1	1
Jumping to conclusions without testing	Making changes without verification disrupts debugging accuracy.	1	1
Lack of consistency in strategy	Frequent switching between debugging approaches without persistence hinders learning.	1	1
Lack of planning leads to repeated mistakes	Without structured reflection, apprentices often repeat ineffective actions.	1	1
Skipping small tests leads to big issues	Neglecting to test small units can result in wasted debugging time	1	1
Unstructured Debugging	Captures instances where apprentices use inconsistent, reactive, or haphazard debugging tactics without a coherent plan, often resulting in inefficiency.	2	8
Casual reasoning as entry point	Debugging often starts with general intuition rather than a structured plan.	1	1
Casual reasoning misses logical errors	Relying solely on intuition often overlooks deeper logical issues.	1	1

Name	Description	Files	References
Casual reasoning reliance	Initial debugging efforts are frequently guided by intuitive rather than logical reasoning.	1	1
Guessing based on code understanding	Apprentices rely on surface-level comprehension of code to make educated guesses about bug locations.	1	1
Hypothesising from current code state	Apprentices form assumptions based on initial code inspection.	1	1
Switch to trial-and-error	When intuition fails, apprentices default to experimenting with fixes.	1	1
Trial-and-error fits web debugging	Unstructured trial-and-error can work well in simpler web-based contexts.	1	1
Trial-and-error method	Debugging often begins with non-systematic experimentation that can be inefficient.	1	1
Subtheme 1.2 – Managing Debugging Complexity	Focuses on the cognitive strain during debugging tasks and misinterpretations (e.g., error logs or unfamiliar codebases).	2	10
Cognitive Load	Describes the mental burden apprentices experience when managing multiple elements of code logic, often leading to overwhelm or errors in reasoning.	2	8
Challenges in tracing code	Apprentices often find it difficult to follow code execution paths.	1	1
Code complexity is overwhelming	Large and unfamiliar codebases can hinder apprentices' navigation and focus.	1	1
Difficulty segmenting code	Apprentices struggle to break complex problems into manageable parts.	1	1
Fixation on wrong sections	Focusing narrowly on specific code sections leads to oversight of wider issues.	1	1
Incomplete mental model causes missteps	Gaps in understanding program flow often lead to flawed assumptions.	1	1
Not understanding broader impact	Neglecting the system-wide effects of a bug fix can cause further errors.	1	1
Overconfidence hides errors	Confidence without verification can blind apprentices to simple coding mistakes.	1	1
Understanding bug impact	Comprehending how a fix affects the whole system is crucial to effective debugging.	1	1
Error Interpretation	Highlights how apprentices read, misread, or apply meaning to error messages and logs, which directly affects bug diagnosis and resolution pathways.	2	2

Name	Description	Files	References
Error messages mislead	Misreading error outputs can lead apprentices down unproductive paths.	1	1
Struggling to interpret error logs	Complex logs and stack traces are difficult for new apprentices to decode.	1	1
Subtheme 2.1 – Tool Adoption and Preferences	Shows the evolution from print statements to IDE debuggers and the way different learning styles impact tool preference.	2	14
Learning Style Influence	Recognises that visual, verbal, and hands-on learners respond differently to debugging strategies, influencing their performance and tool preferences.	1	2
Debugging is influenced by learning style	Effective debugging strategies vary based on an apprentice's preferred learning style.	1	1
Visual strategies for visual learners	Visual tools like debuggers support those with visual learning preferences.	1	1
Tool Familiarity	Refers to how apprentices' comfort, exposure, and understanding of debugging tools—especially IDEs—affect their willingness and effectiveness in using them.	2	12
Go-to use of print statements	Print statements are a preferred initial debugging method due to familiarity.	1	1
Growth in tool appreciation	Apprentices learn to value tool efficiency once they overcome initial hesitation.	1	1
Hesitation to adopt tools	New users often delay embracing powerful debugging tools due to fear or lack of confidence.	1	1
IDE debugger solves python bug	Learning debugger tools enabled successful resolution of Python bugs.	1	1
IDE familiarity improves speed	Familiarity with debugging environments leads to faster problem resolution.	1	1
IDE intimidation delays adoption	Fear of using debugger features prolongs reliance on basic methods.	1	1
Initial intimidation with IDEs	Beginners often find IDE tools overwhelming, delaying their usage.	1	1
Initial struggle with advanced tools	Advanced debugging tools pose challenges for novices.	1	1
Progress from prints to breakpoints	Debugging maturity shows in transitioning from print statements to advanced IDE features.	1	1

Name	Description	Files	References
Tool comfort impacts strategy	Confidence in using IDEs influences which debugging methods are applied.	1	1
Visual cues in IDEs improve flow tracing	Graphical IDE features help clarify complex function flows for visual learners.	1	1
Visual debugging supports understanding	Graphical representations aid in tracing program flow and state changes.	1	1
Subtheme 2.2 – Environment and Interface Support	Covers external conditions (e.g., pressure, IDE visuals) and how they influence strategic choices in debugging.	2	6
Debugging Strategy Selection	Refers to the gradual, often scaffolded, acquisition of debugging expertise and confidence through repeated exposure to increasingly complex tasks.	2	4
Debugging depends on bug type	Strategy effectiveness depends on the complexity and category of the bug.	1	1
Debugging strategy depends on context	Approaches to debugging vary depending on the project and type of bug.	1	1
Simulation of bugs helps strategy selection	Creating and solving artificial bugs allows apprentices to test and compare strategies.	1	1
Tool choice influenced by language	Programming language and tech stack shape the debugging approach.	1	1
Environment Factors	Considers how quiet spaces, distractions, or time pressures in the learning or work setting either support or hinder focused debugging.	1	2
External time pressure disrupts debugging	High-pressure scenarios often lead apprentices to abandon systematic debugging.	1	1
Quiet debugging helps focus	A calm, low-distraction environment enhances debugging concentration.	1	1
Subtheme 3.1 – Structured Debugging Approaches	Encapsulates systematic methods like top-down debugging, use of slicing, simulation, journaling, and repeating bugs for deeper insight.	2	15
Bug Replication	Focuses on the importance of recreating bugs as a strategic practice to understand error behaviour and trace root causes effectively.	1	2
Replication supports root cause analysis	Reproducing bugs consistently helps clarify underlying causes.	1	1

Name	Description	Files	References
Rushed learning under pressure	Time constraints force apprentices to prioritise quick fixes over deeper understanding.	1	1
Strategy Consolidation	Reflects the apprentices' ability to retain, refine, and reflect on effective debugging techniques through habits like journaling and documentation.	2	3
Debugging tasks build confidence incrementally	Solving increasingly complex bugs builds apprentice confidence step by step.	1	1
Structured training improves strategy use	Formal instruction in debugging techniques accelerates apprentice development.	1	1
Success boosted by guided debugging	Step-by-step guidance in early debugging exercises improves long-term independence.	1	1
Structured Strategy	Involves the deliberate use of planned debugging approaches like slicing, top-down decomposition, and test-driven methods that promote efficiency and clarity.	2	10
Effective use of top-down	A structured top-down strategy improves debugging outcomes.	1	1
Mastering slicing	Learning to slice code effectively improves tracing and debugging accuracy.	1	1
Pattern matching experience	Recognising recurring error patterns assists in efficient problem solving.	1	1
Pattern matching in Python debugging	Python debugging becomes easier when apprentices identify recurring structural patterns.	1	1
Slicing improves isolation	Slicing enables more precise identification of fault origins in code.	1	1
Step-by-step execution preferred	Structured, sequential debugging aids apprentices in isolating and resolving issues.	1	1
Structured top-down debugging	A systematic top-down method simplifies complex debugging tasks.	1	1
Testing leads to early bug discovery	Writing tests regularly helps apprentices catch bugs early.	1	1
Unit testing enhances robustness	Implementing unit tests boosts confidence in code stability.	1	1
Use of isolation techniques	Employing isolation helps apprentices identify and address specific errors.	1	1

Name	Description	Files	References
Subtheme 3.2 – Collaborative Debugging	Trainers and mentors described how pair programming, peer walk-throughs, and code reviews helped apprentices verbalise their thoughts, spot errors faster, and build confidence. These collaborative practices were seen as instrumental in building debugging acumen.	2	8
Collaborative Learning	Captures how shared thinking, verbalisation, and peer interaction during debugging foster deeper understanding and strategic refinement.	2	4
Explaining debugging process	Verbalising thought processes helps apprentices arrive at solutions.	1	1
Feedback from code review reshapes mindset	Constructive feedback during reviews influences debugging confidence and approach.	1	1
Mentorship via code reviews	Experienced developers guiding apprentices significantly improves their debugging effectiveness.	1	1
Sharpness through code review	Participating in code reviews sharpens bug detection skills.	1	1
Peer Collaboration	Encompasses the positive effects of paired debugging, walkthroughs, code reviews, and feedback from peers or mentors in scaffolding problem-solving.	1	4
Learning through pair programming	Collaborative programming accelerates understanding of debugging processes.	1	1
Pair programming exposes memory leak	Collaborative debugging quickly uncovered a memory leak issue.	1	1
Peer-led walkthroughs encourage reflection	Explaining code to peers prompts apprentices to think more critically about their logic.	1	1
Rubber duck debugging helps recursion	Explaining recursive problems aloud clarified their solution.	1	1
Subtheme 3.3 – Reflection and Growth	Focuses on how documentation, journals, and guided walkthroughs promote debugging maturity and long-term learning.	2	3
Knowledge Development	Refers to the gradual, often scaffolded, acquisition of debugging expertise and confidence through repeated exposure to increasingly complex tasks.	2	3

Name	Description	Files	References
Building mental models through documentation	Writing comments and diagrams helps apprentices internalise code structure.	1	1
Documenting aids learning	Keeping records of debugging efforts enhances long-term problem-solving skills.	1	1
Regular debugging journals enhance strategy retention	Consistent journaling of bugs and fixes helps apprentices avoid repeating mistakes.	1	1

Codes\\Focus Group\\Stage 5 - Theme Definition

Name	Description	Files	References
Theme 1 - Nature and Handling of Debugging Errors	This theme reflects mentors' and trainers' observations of how apprentices approach and react to debugging errors, based on their reasoning, initial assumptions, and ability to manage problem complexity. It also includes the researcher's account of apprentice reactions during live debugging and interviews.	2	25
Subtheme 1.1 – Debugging Mindsets and Reasoning Patterns	These reflect the intuitive, non-systematic approaches apprentices use, and common mistakes made due to overconfidence, assumption, or incomplete understanding.	2	15
General Reasoning Pattern	Describes the common tendency of apprentices to begin debugging using informal, intuitive, or surface-level logic rather than structured analytical methods.	1	1
Exploring before understanding	Jumping into debugging without a plan often wastes effort.	1	1
Misguided Assumptions	Refers to errors in debugging that stem from overconfidence, confirmation bias, or reliance on incorrect prior beliefs about how the code should behave.	1	6
Confirmation bias skews debugging	Preconceived assumptions can prevent apprentices from seeing simple bugs.	1	1
Copying solutions	Replicating peer solutions without comprehension weakens problem-solving development.	1	1

Name	Description	Files	References
without understanding			
Jumping to conclusions without testing	Making changes without verification disrupts debugging accuracy.	1	1
Lack of consistency in strategy	Frequent switching between debugging approaches without persistence hinders learning.	1	1
Lack of planning leads to repeated mistakes	Without structured reflection, apprentices often repeat ineffective actions.	1	1
Skipping small tests leads to big issues	Neglecting to test small units can result in wasted debugging time	1	1
Unstructured Debugging	Captures instances where apprentices use inconsistent, reactive, or haphazard debugging tactics without a coherent plan, often resulting in inefficiency.	2	8
Casual reasoning as entry point	Debugging often starts with general intuition rather than a structured plan.	1	1
Casual reasoning misses logical errors	Relying solely on intuition often overlooks deeper logical issues.	1	1
Casual reasoning reliance	Initial debugging efforts are frequently guided by intuitive rather than logical reasoning.	1	1
Guessing based on code understanding	Apprentices rely on surface-level comprehension of code to make educated guesses about bug locations.	1	1
Hypothesising from current code state	Apprentices form assumptions based on initial code inspection.	1	1
Switch to trial-and-error	When intuition fails, apprentices default to experimenting with fixes.	1	1
Trial-and-error fits web debugging	Unstructured trial-and-error can work well in simpler web-based contexts.	1	1
Trial-and-error method	Debugging often begins with non-systematic experimentation that can be inefficient.	1	1
Subtheme 1.2 – Managing Debugging Complexity	Focuses on the cognitive strain during debugging tasks and misinterpretations (e.g., error logs or unfamiliar codebases).	2	10
Cognitive Load	Describes the mental burden apprentices experience when managing multiple elements	2	8

Name	Description	Files	References
	of code logic, often leading to overwhelm or errors in reasoning.		
Challenges in tracing code	Apprentices often find it difficult to follow code execution paths.	1	1
Code complexity is overwhelming	Large and unfamiliar codebases can hinder apprentices' navigation and focus.	1	1
Difficulty segmenting code	Apprentices struggle to break complex problems into manageable parts.	1	1
Fixation on wrong sections	Focusing narrowly on specific code sections leads to oversight of wider issues.	1	1
Incomplete mental model causes missteps	Gaps in understanding program flow often lead to flawed assumptions.	1	1
Not understanding broader impact	Neglecting the system-wide effects of a bug fix can cause further errors.	1	1
Overconfidence hides errors	Confidence without verification can blind apprentices to simple coding mistakes.	1	1
Understanding bug impact	Comprehending how a fix affects the whole system is crucial to effective debugging.	1	1
Error Interpretation	Highlights how apprentices read, misread, or apply meaning to error messages and logs, which directly affects bug diagnosis and resolution pathways.	2	2
Error messages mislead	Misreading error outputs can lead apprentices down unproductive paths.	1	1
Struggling to interpret error logs	Complex logs and stack traces are difficult for new apprentices to decode.	1	1
Theme 2 - Technology's Role in Debugging Processes	This theme highlights mentor perspectives on how apprentices engage with debugging tools, from print statements to IDEs, and how their learning styles, comfort levels, and environmental context affect the effectiveness of those tools.	2	20
Subtheme 2.1 – Tool Adoption and Preferences	Shows the evolution from print statements to IDE debuggers and the way different learning styles impact tool preference.	2	14
Learning Style Influence	Recognises that visual, verbal, and hands-on learners respond differently to debugging strategies, influencing their performance and tool preferences.	1	2

Name	Description	Files	References
Debugging is influenced by learning style	Effective debugging strategies vary based on an apprentice's preferred learning style.	1	1
Visual strategies for visual learners	Visual tools like debuggers support those with visual learning preferences.	1	1
Tool Familiarity	Refers to how apprentices' comfort, exposure, and understanding of debugging tools—especially IDEs—affect their willingness and effectiveness in using them.	2	12
Go-to use of print statements	Print statements are a preferred initial debugging method due to familiarity.	1	1
Growth in tool appreciation	Apprentices learn to value tool efficiency once they overcome initial hesitation.	1	1
Hesitation to adopt tools	New users often delay embracing powerful debugging tools due to fear or lack of confidence.	1	1
IDE debugger solves python bug	Learning debugger tools enabled successful resolution of Python bugs.	1	1
IDE familiarity improves speed	Familiarity with debugging environments leads to faster problem resolution.	1	1
IDE intimidation delays adoption	Fear of using debugger features prolongs reliance on basic methods.	1	1
Initial intimidation with IDEs	Beginners often find IDE tools overwhelming, delaying their usage.	1	1
Initial struggle with advanced tools	Advanced debugging tools pose challenges for novices.	1	1
Progress from prints to breakpoints	Debugging maturity shows in transitioning from print statements to advanced IDE features.	1	1
Tool comfort impacts strategy	Confidence in using IDEs influences which debugging methods are applied.	1	1
Visual cues in IDEs improve flow tracing	Graphical IDE features help clarify complex function flows for visual learners.	1	1
Visual debugging supports understanding	Graphical representations aid in tracing program flow and state changes.	1	1

Name	Description	Files	References
Subtheme 2.2 – Environment and Interface Support	Covers external conditions (e.g., pressure, IDE visuals) and how they influence strategic choices in debugging.	2	6
Debugging Strategy Selection	Refers to the gradual, often scaffolded, acquisition of debugging expertise and confidence through repeated exposure to increasingly complex tasks.	2	4
Debugging depends on bug type	Strategy effectiveness depends on the complexity and category of the bug.	1	1
Debugging strategy depends on context	Approaches to debugging vary depending on the project and type of bug.	1	1
Simulation of bugs helps strategy selection	Creating and solving artificial bugs allows apprentices to test and compare strategies.	1	1
Tool choice influenced by language	Programming language and tech stack shape the debugging approach.	1	1
Environment Factors	Considers how quiet spaces, distractions, or time pressures in the learning or work setting either support or hinder focused debugging.	1	2
External time pressure disrupts debugging	High-pressure scenarios often lead apprentices to abandon systematic debugging.	1	1
Quiet debugging helps focus	A calm, low-distraction environment enhances debugging concentration.	1	1
Theme 3 - Strategies and Challenges in Debugging	This theme captures how trainers and mentors observed apprentices developing, adapting, or struggling with debugging strategies, and how peer collaboration, repetition, and training interventions shaped their effectiveness.	2	26
Subtheme 3.1 – Structured Debugging Approaches	Encapsulates systematic methods like top-down debugging, use of slicing, simulation, journaling, and repeating bugs for deeper insight.	2	15
Bug Replication	Focuses on the importance of recreating bugs as a strategic practice to understand error behaviour and trace root causes effectively.	1	2
Replication supports root cause analysis	Reproducing bugs consistently helps clarify underlying causes.	1	1

Name	Description	Files	References
Rushed learning under pressure	Time constraints force apprentices to prioritise quick fixes over deeper understanding.	1	1
Strategy Consolidation	Reflects the apprentices' ability to retain, refine, and reflect on effective debugging techniques through habits like journaling and documentation.	2	3
Debugging tasks build confidence incrementally	Solving increasingly complex bugs builds apprentice confidence step by step.	1	1
Structured training improves strategy use	Formal instruction in debugging techniques accelerates apprentice development.	1	1
Success boosted by guided debugging	Step-by-step guidance in early debugging exercises improves long-term independence.	1	1
Structured Strategy	Involves the deliberate use of planned debugging approaches like slicing, top-down decomposition, and test-driven methods that promote efficiency and clarity.	2	10
Effective use of top-down	A structured top-down strategy improves debugging outcomes.	1	1
Mastering slicing	Learning to slice code effectively improves tracing and debugging accuracy.	1	1
Pattern matching experience	Recognising recurring error patterns assists in efficient problem solving.	1	1
Pattern matching in Python debugging	Python debugging becomes easier when apprentices identify recurring structural patterns.	1	1
Slicing improves isolation	Slicing enables more precise identification of fault origins in code.	1	1
Step-by-step execution preferred	Structured, sequential debugging aids apprentices in isolating and resolving issues.	1	1
Structured top-down debugging	A systematic top-down method simplifies complex debugging tasks.	1	1
Testing leads to early bug discovery	Writing tests regularly helps apprentices catch bugs early.	1	1

Name	Description	Files	References
Unit testing enhances robustness	Implementing unit tests boosts confidence in code stability.	1	1
Use of isolation techniques	Employing isolation helps apprentices identify and address specific errors.	1	1
Subtheme 3.2 – Collaborative Debugging	Trainers and mentors described how pair programming, peer walk-throughs, and code reviews helped apprentices verbalise their thoughts, spot errors faster, and build confidence. These collaborative practices were seen as instrumental in building debugging acumen.	2	8
Collaborative Learning	Captures how shared thinking, verbalisation, and peer interaction during debugging foster deeper understanding and strategic refinement.	2	4
Explaining debugging process	Verbalising thought processes helps apprentices arrive at solutions.	1	1
Feedback from code review reshapes mindset	Constructive feedback during reviews influences debugging confidence and approach.	1	1
Mentorship via code reviews	Experienced developers guiding apprentices significantly improves their debugging effectiveness.	1	1
Sharpness through code review	Participating in code reviews sharpens bug detection skills.	1	1
Peer Collaboration	Encompasses the positive effects of paired debugging, walkthroughs, code reviews, and feedback from peers or mentors in scaffolding problem-solving.	1	4
Learning through pair programming	Collaborative programming accelerates understanding of debugging processes.	1	1
Pair programming exposes memory leak	Collaborative debugging quickly uncovered a memory leak issue.	1	1
Peer-led walkthroughs encourage reflection	Explaining code to peers prompts apprentices to think more critically about their logic.	1	1
Rubber duck debugging helps recursion	Explaining recursive problems aloud clarified their solution.	1	1

Name	Description	Files	References
Subtheme 3.3 – Reflection and Growth	Focuses on how documentation, journals, and guided walkthroughs promote debugging maturity and long-term learning.	2	3
Knowledge Development	Refers to the gradual, often scaffolded, acquisition of debugging expertise and confidence through repeated exposure to increasingly complex tasks.	2	3
Building mental models through documentation	Writing comments and diagrams helps apprentices internalise code structure.	1	1
Documenting aids learning	Keeping records of debugging efforts enhances long-term problem-solving skills.	1	1
Regular debugging journals enhance strategy retention	Consistent journaling of bugs and fixes helps apprentices avoid repeating mistakes.	1	1