The Iceberg Theory on the Shared Understanding of Non-Functional Requirements in Continuous Software Engineering

by

Colin Werner
Master of Mathematics, University of Waterloo, 2012
Bachelor of Computer Science, University of Waterloo, 2007

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science, University of Victoria

© Colin Werner, 2024
University of Victoria

All rights reserved. This proposal may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

The Iceberg Theory on the Shared Understanding of Non-Functional Requirements in Continuous Software Engineering

by

Colin Werner
Master of Mathematics, University of Waterloo, 2012
Bachelor of Computer Science, University of Waterloo, 2007

Supervisory Committee

Dr. Daniela Damian, Supervisor
Department of Computer Science, University of Victoria, Canada

---

Dr. Neil Ernst, Departmental Member
Department of Computer Science, University of Victoria, Canada

---

Dr. Daniel M. Berry, Outside Member
Cheriton School of Computer Science, University of Waterloo, Canada

---

Dr. Klaas-Jan Stol, Outside Member
School of Computer Science and Information Technology, University of College Cork, Ireland

ABSTRACT

While software is largely considered to be heavily associated with technology, it is ultimately software developers that design, discuss, architect, write, test, re-write, and maintain the code that is compiled into the respective software. These humans are, after all, not perfect, and for the most part are not working in isolation from one another. Thus building a shared understanding amongst a group of software developers, including requirements, is key to ensuring downstream software activities are efficient and effective. Non-functional requirements (NFR), which include performance, availability, and maintainability, are vitally important to overall software quality and that the software fulfills its intended purpose. Research has shown that NFRs are, in practice, poorly defined and difficult to verify, especially in agile environments. A lack of attention to NFRs could potentially derail a software project. Many an organization frequently incurs technical debt by making trade-offs between the timely delivery of promised software features and rigorous system design that incorporates sufficient attention for vital NFRs.

The software industry has always sought to shorten the time of delivery of systems and features, including the adoption of iterative and incremental methods, in particular agile methods, which have become the norm. Practices such as Continuous Integration, which relies on automatically testing newly integrated code, inspired the automation of other activities within software development, allowing the whole development process to become more *continuous*. This has led to a trend that has been called continuous software engineering (CSE). CSE relies on automated and fast releases of new versions, delivering new features quickly to users. However, feature development is usually driven by functional requirements (FR), while such fast delivery frequently means that non-functional requirements receive less attention. Previous work has pointed out that NFRs are frequently neglected in agile development, and indeed, little work exists that has explored NFRs in the context of CSE. A major complication of an NFR is that it relates to an entire system's architecture, which is problematic for two reasons. First, evaluating the impact of frequent updates, which comes with a continuous software engineering process, on NFRs is very challenging. Second, it can be challenging for all developers in a project to have a shared and common understanding of a system's architecture, in particular for very large systems.

In this dissertation, I describe a multi-year, multi-case study to empirically investigate how four organizations, for which NFRs are paramount to their business survival, build, manage, and maintain the shared understanding of NFRs in their continuous practices. My research goal is to develop a deep and rich understanding of the relationship between an organization and their shared understanding of NFRs in CSE. Through the results and insights from this in-depth research, I developed the Iceberg Theory on the complex and intricate relationship between a shared understanding of NFRs and CSE. The theory includes a classification of shared understanding, a lack of a shared understanding, nine practices an organization may use to build a shared understanding, in addition to the associated challenges and the triggers that prompted an organization to build said shared understanding.

Contents

Supervisory Committee ii

Abstract iii

Contents v

List of Tables ix

List of Figures x

1 Introduction 1
- 1.1 Software Quality 2
- 1.2 Requirements Engineering 3
- 1.3 Non-Functional Requirements 5
- 1.4 Continuous Software Engineering 6
- 1.5 Shared Understanding 8
- 1.6 Theory Building in Software Engineering 9
- 1.7 Research Goal & Questions 9
  - 1.7.1 Scope 10
  - 1.7.2 Underlying Conceptual Framework 10
  - 1.7.3 Strategy for Generating the Theory 11
- 1.8 Contributions 13
  - 1.8.1 Publications 14
- 1.9 Structure 15

2 Background and Related Work 16
- 2.1 Software Quality & Rework 16
- 2.2 Shared Understanding 17
- 2.3 Requirements Engineering 18

2.4 Non-Functional Requirements 19
2.5 Continuous Software Engineering 20
2.6 Theory Building in Software Engineering 23

3 Research Method 24
3.1 Preparation 26
3.1.1 Site Selection 27
3.1.2 Preliminary Study 28
3.2 RQ1: Shared Understanding and CSE 29
3.3 RQ2: Challenges and Practices 30
3.4 Developing the Theory 31
3.5 Ethics 32
3.6 Risks and Limitations 32
3.6.1 Credibility 32
3.6.2 Dependability 33
3.6.3 Confirmability 34
3.6.4 Transferability 34

4 Interplay Between a Shared Understanding and CSE 37
4.1 Research Methodology 37
4.1.1 Data Collection 38
4.1.2 Data Analysis 40
4.2 Findings 41
4.2.1 What Contributes to Lack of Shared Understanding of NFRs? (RQ1.1) 41
4.2.2 Which NFRs are Most Associated with Lack of Shared Understanding? (RQ1.2) 44
4.2.3 What Amount of a Lack of Shared Understanding is Accidental versus Essential? (RQ1.3) 46
4.2.4 What practices can be employed to avoid a lack of shared understanding? (RQ1.4) 46
4.2.5 What Triggered These Organizations to Build a Shared Understanding? 48
4.3 Discussion 50
4.4 Threats to Validity 51

5 Practices & Challenges with a Shared Understanding 53
5.1 Research Methodology . . . 53
5.1.1 Data Collection . . . 54
5.1.2 Data Analysis and Results Validation . . . 54
5.2 Findings . . . 58
5.2.1 Practices For Handling Non-Functional Requirements in Continuous Software Engineering (RQ2.1) . . . 59
5.2.2 Challenges (RQ2.2) . . . 63
5.3 Discussion . . . 68
5.4 Threats to Validity . . . 69

6 An In-Depth Description of the Iceberg Theory 72
6.1 Lack of Shared Understanding . . . 75
6.2 Triggers to Build a Shared Understanding . . . 77
6.2.1 New Requirements . . . 77
6.2.2 Customer Demands . . . 78
6.2.3 Regulation . . . 78
6.2.4 Technical Debt . . . 79
6.2.5 Scope Creep . . . 79
6.2.6 Service Disruptions . . . 80
6.2.7 Triggers Prompt Action . . . 80
6.3 Practices in Building a Shared Understanding . . . 81
6.3.1 Put a Number on the NFR . . . 81
6.3.2 Let Someone Else Manage the NFR . . . 82
6.3.3 Write Your Own Tool . . . 84
6.3.4 Put the NFR in Source Control . . . 85
6.3.5 Solicit Feedback . . . 85
6.3.6 Leverage Past Experience . . . 86
6.3.7 Develop Wireframes . . . 87
6.3.8 Discuss Problems and Solutions . . . 87
6.3.9 Ask Questions . . . 88
6.4 Challenges to Building a Shared Understanding . . . 89
6.4.1 Loss of Control . . . 89
6.4.2 Deprioritization of NFRs . . . 89
6.4.3 Lack of Domain Knowledge . . . 90

6.4.4 Inadequate Communication 91
6.4.5 NFRs are Hard to Automate 92
6.5 A Shared Understanding of NFRs 92

7 Discussion & Conclusion **94**
7.1 Shared Understanding and CSE 94
7.2 NFRs in CSE: A Silver Lining 97
7.3 The Importance of Configurability as an NFR 99
7.4 How an Organization Can Build a Shared Understanding in CSE . . 101
7.5 Final Conclusion 103

Bibliography **106**

A Publications **122**
A.1 Publications from this Dissertation 122
A.2 Other Publications 122

B RQ1 Artifacts **125**
B.1 RQ1 Focus Group Questions 125
B.2 RQ1 Complete Codebook 125

C RQ2 Artifacts **130**
C.1 RQ2 Interview Questions 130
C.2 RQ2 Complete Codebook 131

List of Tables

Table 3.1 Data Collection Activities 26
Table 3.2 Selected Site Descriptions 28
Table 3.3 Theory Concept Development 36

Table 4.1 Summary of Factors Contributing to LSU of an NFR (RQ1.1) . 41

Table 5.1 Participant Information and Demographics 55
Table 5.2 Progression of Thematic Analysis 55
Table 5.3 Codebook Samples 57
Table 5.5 Practices and Challenges with Shared Understanding of NFRs with CSE 59
Table 5.4 Ranking of NFRs 59

Table B.1 RQ1 Codebook 126

Table C.1 RQ2 Complete Codebook 131

List of Figures

Figure 1.1 Glinz and Fricker's Forms and Categories of Shared Understanding [1] . . . 11

Figure 3.1 Summary of Research Methodology . . . 25

Figure 4.1 Sampling of Rework Tasks . . . 38

Figure 4.2 NFRs with ≥ 6 NFR-related tasks (RQ1.2) . . . 45

Figure 6.1 The Iceberg Theory . . . 73

Chapter 1

Introduction

The development of software has long been known to be a complex and expensive undertaking. For one it takes a specific skill set to develop software, although the number of developers has grown substantially over the years as higher-order programming languages have made the development of software more accessible. With a larger portion of the population able to write software, perhaps with little to no training, the amount of software being developed has skyrocketed.

However, not only has the amount of software being developed increased, but the *complexity* of software has also increased. The complexity is in part due to the ubiquitous nature of software, as it is now ever-present in automobiles, refrigerators, and an ever-increasing number of Internet of Things (IoT) devices.

Furthermore, while the ubiquity of software is vast, the rich functionality that each device supports is also a large contributing factor to software complexity. With the continuous increase in both the size and complexity of software, there has been a struggle to maintain a high level of quality across the deep and broad domains of software. Lower-quality software requires rework to increase the quality to an acceptable level before release, ultimately driving up the cost of software even more. Unfortunately, low-quality software may have several underlying causes. Low-quality software may be caused by unrealistic schedules, overly complex software, poor project execution, or incorrect or incomplete requirements.

The concept of software engineering was proposed to bring traditional engineering approaches to software development, with the primary goal of *improving* software development to be more reliable, efficient, and effective. The term software engineering first appeared in the 1960s to combat a plethora of software problems, including over budget, over schedule, and ultimately to increase software quality. This so-called

software crisis [2] was primarily caused by the ever-increasing number of software developers writing and creating more and more software. Initially, the definition of software engineering took on an overly specific definition, perhaps too narrowly focused on solely the engineering of software, unlike other engineering disciplines that included many other facets not originally included in software engineering [3]. Over time, the definition and role of software engineering has expanded to include a broader to include several different phases, not only related to software design, implementation, and testing. Today the software engineering body of knowledge includes areas such as software maintenance, configuration management, process, models, quality, economics, and requirements engineering [4].

1.1 Software Quality

Software quality has been at the forefront since the beginning of software. High software quality is still one of the most sought-after attributes of software and one that continues to benefit from subsequent research. Software quality has been researched in-depth in almost every domain, from requirements to programming languages, processes, and education. While achievements have been made there still does not yet exist a silver bullet for software quality. Despite the amount of research, achieving higher quality software is highly sought after in practice.

One problem with software quality is neglecting requirements resulting in rework: the extra work needed to fix problems in software due to poorly understood requirements. Rework is extremely costly [5] and accounts for 40-50% of the effort on software projects [6].

It is well known that the earlier something is caught the easier and cheaper it is to fix, as fixing an erroneous requirement before any analysis or development has begun. There are two aspects of software quality:

1. are we building the *right* thing? i.e. validation, and

2. are we building the thing *right*? i.e. verification.

The ability to build the *right* thing *right* is the key to software success.

Verification of software ensures that your software is consistent and meets the requirements. However, if a requirement is not correct, then verification is perhaps a wasted effort until you can understand what it is that your customer wants, i.e. you

need to get the requirement right. Understanding exactly what your customer wants is paramount to software success, and validation confirms whether you have a shared understanding with your customer. Once you understand exactly what your customer wants, then you are said to have a shared understanding with your customer. While shared understanding is important between your customer and your organization, it is also important *within* your organization.

One of the many problems negatively affecting software quality is requirements-related rework: the extra work needed to fix problems in software due to poorly understood requirements. Rework is extremely costly [5] and accounts for 40-50% of the effort on software projects [6].

1.2 Requirements Engineering

Requirements engineering is one domain under the software engineering umbrella that aims to make software engineering easier, more economical, and improve the overall quality of software. While the term software engineering has been around since the 1960s, the term requirements engineering was not mainstream until the 1990s, coinciding with the inaugural International Requirements Engineering Conference in 1993. Requirements engineering has been shown to play an important role in software engineering, and often the best software is not created in the development phase, but in the requirements phase.

While writing a software requirement might seem trivial, it is an arduous task, especially when considering that the individual *writing* a requirement may not be the person *implementing*, *designing*, or *testing* that requirement. For a requirement to be effective, it must meet a high standard of characteristics, including being concise, complete, consistent, atomic, traceable, current, unambiguous, ranked, and verifiable. Thus, requirements engineering has an entirely dedicated and unique process for itself.

Requirements engineering in the traditional waterfall model typically starts at the beginning of a software project and continues through the entire life cycle involving six primary activities: elicitation, analysis, negotiation, documentation, validation, verification, and management [7]. Typically, the first activity of traditional requirements engineering is elicitation. Elicitation is when the stakeholders requiring a software solution will meet with an entity that can provide that software solution, or perhaps only the requirements for a software solution. During elicitation, the requirements for a particular solution are typically documented in a software requirements document,

or as part of the more comprehensive software requirements specification. Elicitation is typically a key component in extracting tacit, or implicit, knowledge that is considered difficult to communicate through any medium of communication.

Once requirements have been identified a written proposal is drafted, including visual diagrams and models, as appropriate, and reviewed with stakeholders to ensure any conflicts are resolved. With a defined scope, requirements negotiation is usually the next stage whereby the exact scope of the software will be decided. Additional requirements may be added that were not originally considered; requirements may also be prioritized, or even descoped from the product entirely. Once the negotiation phase is complete the requirements document serves as a contract between the stakeholders and developers.

With a requirements complete document, the stakeholders may require the developers to produce a requirements specification, which includes the requirements but also describes the proposed system behaviour to satisfy the requirements. With a relatively complete requirements specification document the next stage is validation of the software requirements specification. Validation checks that the documented requirements and specifications are consistent and meet the needs of the stakeholders, i.e. they are building the *right* product. Once the system, or part of the system, has been implemented then the requirements should be verified, i.e. they build the product *right*.

The last stage, requirements management, should be present throughout each requirement engineering activity, as change is usually inevitable and thus should be expected. As part of requirements management, previous stages (e.g. analysis or validation) of requirements engineering may be revisited at any number of times and as the requirements may continue throughout the life of any particular software product. Given the ever-changing nature of requirements, effective requirements management includes traceability, which helps reduce waste and ensure effective and efficient changes in requirements throughout the other stages of software development (implementation, testing, deployment, maintenance, etc.)

While there are numerous classifications of software requirements, I classify a requirement as either *functional* (FR) or *non-functional* (NFR). Overall, there is consensus that an FR "specifies an action that the system must be able to perform, without considering physical constraints" [8], or, in other words, what the product or system must do. FRs are fairly straightforward and define the desired change in the environment, e.g. *if a user selects X then the user shall be presented with Y.*

However, on the flip side, the exact definition of an NFR usually invokes a heated debate amongst requirements engineers.

1.3 Non-Functional Requirements

NFRs are not only known by the term non-functional requirements, as NFRs go by a multitude of terms, including quality requirements or attributes, architecturally significant requirements, or the *-illities* (usability, reliability, maintainability, etc.). For this dissertation, I shall use the term NFR and Glinz's definition as an "attribute of or constraint on a system" [9], whereas an FR represents *what* the desired change in the environment is and help define the system design.

NFRs are very important in software projects [9] as NFRs are key to ensuring a system is usable, highly performing, and maintainable, among other NFRs. For many software organizations, particularly in today's increasingly service-oriented, fast-paced software development marketplace, NFRs, such as system uptime, code maintainability, and responsiveness, are vital to success [10]. For example, a recent software outage at Amazon was estimated to cost 99 million USD for the 63 minutes Amazon's site was down [11]. Despite their importance, research shows that NFRs present more engineering difficulties than FRs [12]. Studies of practice demonstrate NFRs are difficult to explicitly express [13] and even more difficult to verify or validate [14], whether they are part of a formal requirements specification or an agile user story.

Furthermore, NFRs can have a great effect on a system's design and architecture [15, 16]. No other single artifact has the ability to shape an architecture, thus getting the NFRs *right* is paramount to the success of any quality software product. The effect of ignoring NFRs for too long could spell disaster for a software product due to the importance of NFRs on software architecture.

As software engineering has evolved over the years, requirements engineering has also continued to adapt. In particular, with the advent of agile software development, agile requirements engineering has seen an increase in research and industrial applications. At the onset, the four values of Agile Manifesto[1] seem to contradict everything software engineers and researchers know about traditional requirements engineering. In particular, *working software over comprehensive documentation*, where

[1] `https://www.agilemanifesto.org`

a requirement specification would likely be considered comprehensive documentation. In addition, *customer collaboration over contract negotiation*, while requirements engineering does involve stakeholders to negotiate the requirements and specifications, it typically is only early on, and not throughout, the requirements engineering life cycle. While *responding to change over following a plan* could be part of the requirements engineering life cycle, depending on whether the entire requirement specification is completed at the beginning and how well change is incorporated throughout the life cycle. Finally, *individuals and interactions over processes and tools* is certainly open to interpretation, as there does exist an ISO standard for a requirements engineering process[2], although this doesn't mean an organization has to follow it.

Software organizations have adapted and many consider user stories to replace traditional requirements. Although user stories and requirements are each written from a different, unique perspective. User stories are usually written from the perspective of an end-user, e.g. *As a user, I would like to...*. Since a user story is typically hyper-focused on a user's perspective, there is considerable overlap with FRs; however, where user stories fall short is the inclusion of NFRs. Given the aforementioned importance of NFRs in determining a system's overall architecture, this lack of attention is a growing concern. While a reasonably clear understanding of NFRs exists in general, and how agile software development and requirements engineering affect one another, it is not well-understood how continuous software engineering (CSE), which leverages agile practices, deals with NFRs.

1.4 Continuous Software Engineering

CSE has garnered considerable attention and requires an organization to support rapid, frequent builds; automated tests; and transparency of the build and test process. Despite CSE being a firmly established mainstream industry, the details surrounding the birth of CSE are open to debate. A form of CSE, *continuous integration*, was first used in Grady Booch's book [17] and subsequently incorporated into the practices of extreme programming [18]. Continuous integration revolves around the ability to rapidly integrate small code changes into an application. However, continuous integration did not gain widespread momentum until relatively recently, in large

[2] `https://standards.ieee.org/standard/29148-2018.html`

part due to Martin Fowler's influential blog posts[3] breaking down the 10 practices required to implement continuous integration:

1. Maintain a single source repository
2. Automate the build
3. Make your build self-testing
4. Everyone commits to the mainline every day
5. Every commit should build the mainline on an integration machine
6. Keep the build fast
7. Test in a clone of the production environment
8. Make it easy for anyone to get the latest executable
9. Ensure that system state and changes are visible
10. Automate deployment

As part of CSE, several extensions to continuous integration exist. The two most popular terms are continuous delivery and continuous deployment. Continuous delivery adds the extra practice of keeping the software in a constantly releasable state. Continuous deployment adds the practice of deploying every single change to a production environment. Continuous delivery and deployment increase visibility, provide faster feedback to developers and allow stakeholders to be more involved in evaluating the product.

Continuous delivery and deployment also help bring together two organizations, development and operations, into the coined term DevOps. DevOps is built on lean and agile practices and is a contributing success factor to both continuous delivery and deployment. DevOps is believed to have torn down the proverbial wall between development teams and organizations, whereby there is no distinction between members of either team and they are in fact on the same team. The theory indicates that tearing down the proverbial wall will enable an organization to develop and deliver more efficient and effective software by integrating the development and operations

[3] `https://martinfowler.com/articles/originalContinuousIntegration.html`

teams to encourage the sharing of knowledge and build and maintain a mutual understanding.

CSE goes beyond development, or operations, and extends into other activities, such as continuous planning, continuous budgeting, security, etc. CSE can be applied to bring an entire organization together in a manner that everyone is operating on a similar *continuous* cadence. So not only are the development and operations teams sharing and building knowledge but also include as many aspects of the organization as possible.

From a development perspective, there are many perceived benefits for an organization to adopt CSE, including:

1. a decrease in merge complications,
2. an increase in productivity,
3. the ability to acquire rapid feedback, and
4. an overall increase in the quality of software produced.

Although many old-fashioned software engineers argue that the very foundations of CSE contradict what we traditionally have thought about software quality. In particular, long drawn-out software life cycles (e.g. waterfall) were thought to help iron out bugs before any potential customer receives the software. High-performing organizations have been shown to effectively and efficiently follow CSE principles and ultimately produce higher-quality software.

1.5 Shared Understanding

Much like NFRs, shared understanding is also a critical success factor in achieving high-quality software that meets stakeholders' needs [19]. Shared understanding refers to a form of communal knowledge or belief [1], and is a critical factor when working in a collaborative environment as different people may have diverging understandings that are not mutually shared. Shared understanding also fosters a more effective, motivated, and collaborative team, more efficient use of resources, and less conflict [20]. On the other hand, a lack of shared understanding could contribute to a lower-quality product. There are two aspects of shared understanding to consider. First, is whether an organization has a shared understanding of what the customer wants in

the product, if this shared understanding does not exist then there is the potential for the product to be entirely derailed. Second, once a shared understanding of the product has been created and the organization is ready to begin developing the product, it is important to build an *internal* shared understanding amongst product managers, developers, testers, support specialists, and almost any other applicable employee.

However, empirical research on the shared understanding in software engineering projects is scarce [1], let alone on the shared understanding of NFRs. The shared understanding of NFRs is even harder to study, due to the cross-cutting nature of NFRs, and has not been studied in the rapidly changing environments of agile or CSE, yet remains a critical success factor to software and merits further attention.

1.6 Theory Building in Software Engineering

The development of software engineering-focused theories that describe real-world phenomena and context based on empirical evidence has received little attention [21], until more recently [22]. However, theories can help researchers and practitioners understand and make sense of relatively complex software engineering topics, such as the shared understanding of NFRs. For example, Herbsleb and Roberts's theory of coordination on collaboration in software engineering [23] models and predicts how decisions are made, and the effect the decisions have on development time and developer productivity in software engineering projects. Given the practical relevance of the topic of interest in this dissertation, a shared understanding of NFRs, I argue that a theory that describes the key mechanisms may be of considerable interest.

1.7 Research Goal & Questions

The goal of the research in this dissertation is to develop a deep and rich understanding of how an organization practising CSE may build a shared understanding of NFRs. The following research questions guide the research in this dissertation:

RQ1: What is the interplay between a shared understanding of NFRs and CSE?

RQ2: How does an organization build, manage, and maintain a shared understanding of NFRs in CSE?

In this dissertation a multi-case study [24] was executed using a mixed-methods approach, including the triangulation of data from quantitative, qualitative [25], and immersive techniques [26] on four industrial organizations. The data were analyzed utilizing thematic analysis [27] to answer RQ1 and RQ2. The findings and insights from the answers to RQ1 and RQ2 were used to develop this theory on the shared understanding of NFRs in CSE, henceforth referred to as the Iceberg Theory.

1.7.1 Scope

An important aspect of any dissertation is to accurately describe the scope and definitions surrounding the underlying research. The scope of this dissertation considers the shared understanding only from an internal software development perspective, namely the scope is limited to employees of a development organization. For example, this research examines the interactions from developers to other developers, and developers to as many other internal organizations, including quality assurance, management, support, operations, marketing, etc. Notably absent from this dissertation is the concept of external stakeholders, or customers, including both the client who is *paying* for the software or the end-user who will ultimately *using* the software. An exception may be the case where the customer requesting the software *may be* internal to the company.

1.7.2 Underlying Conceptual Framework

The underlying conceptual framework for this research employs Glinz and Fricker's [1] forms and categories of a shared understanding, see Figure 1.1, which are described in this section and used throughout this dissertation. At the highest level, Glinz and Fricker identify information as either relevant or irrelevant with respect, i.e. does the information matter? Furthermore, a shared understanding can either be *true* or *false*. A true shared understanding indicates a group of individuals has an equal level of shared understanding of a particular topic without any misunderstandings. Finally, a false shared understanding implies that a group of individuals believes they have a shared understanding of a particular topic, but in fact, there is actually a *misunderstanding*. Additionally, there are two forms of shared understanding between a group of individuals: *implicit* and *explicit*; independent of whether what they share is true or false. Explicit shared understanding occurs when a group of individuals has the same interpretation of an explicit specification (e.g. requirements and design docu-

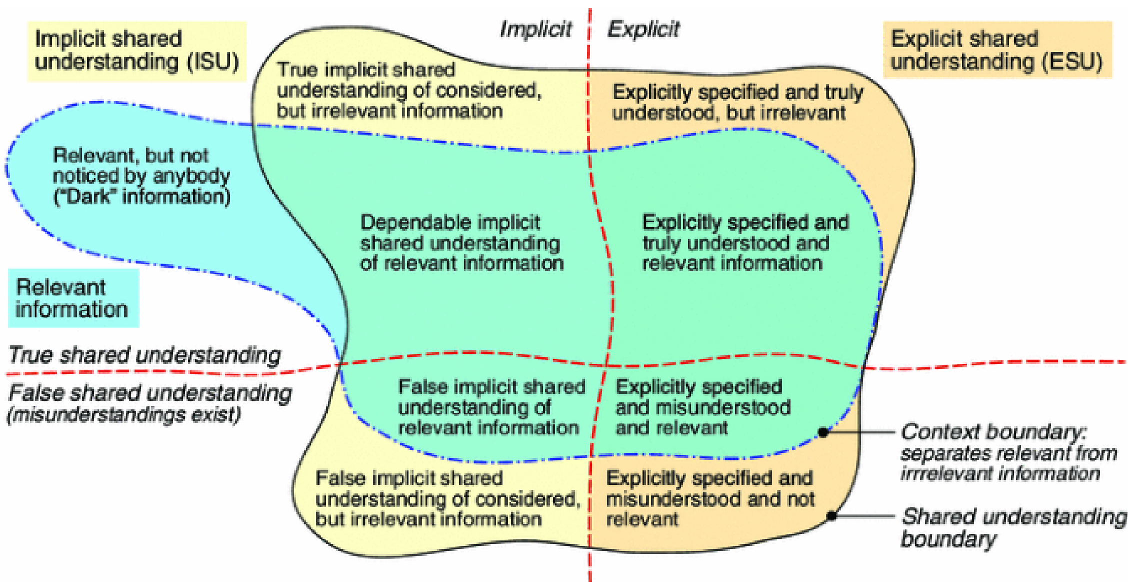

Figure 1.1: Glinz and Fricker's Forms and Categories of Shared Understanding [1]

ments, manuals, etc.). Implicit shared understanding is when a group of individuals has the same interpretation of some non-specified knowledge.

Of particular interest to this dissertation is the shared understanding of relevant information. Relevant information could be understood by no one, dark information; understood by everyone, true shared understanding; or it could be understood by some number of people, but not everyone, which is not necessarily a false shared understanding. The subtle difference between a lack of shared understanding and a false shared understanding is that according to Glinz and Fricker [1] a false shared understanding implies that there is a misunderstanding — which by definition implies the inability to correctly understand something. Alternatively, the lack of shared understanding does not necessarily imply a misunderstanding, it simply means there is some information that everyone *should* equally understand, but has not yet achieved the status of a true shared understanding. The lack of shared understanding is an important concept discussed throughout this dissertation.

1.7.3 Strategy for Generating the Theory

In developing the Iceberg Theory, I adopted Ralph's guidelines for developing process theories in software engineering [28]. In this section, I describe the strategy adopted, including an overview of the research method employed, as well as a discussion of

trade-offs and choices during the research. A more in-depth description can be found in Chapter 3.

The first step to generating a theory is to select an appropriate strategy. In this dissertation, I adopted a multi-case study approach that draws on multiple data sources, including both qualitative and quantitative data, facilitating triangulation to increase the trustworthiness of the findings [24]. Case study research is suitable for studying socio-technical phenomena and to develop empirically-driven theories in software engineering [29]. In my research, I sought to advance the understanding of how developers address the shared understanding of NFRs in software organizations, so a case study approach grounded in real-world organizational settings was most appropriate. This approach afforded me a prolonged engagement lasting several weeks or months onsite to conduct observations, interviews, and data analysis, as well as a level of protection from "missing instances, or over-simplifying and over-rationalizing process" [28].

Initially, I performed case studies at three organizations, Alpha, Beta, and Gamma, to answer RQ1 and RQ2. The resulting findings and insights from these three case studies were used to develop the first iteration of the Iceberg Theory. As part of research and resulting Iceberg theory, I uncovered *challenges* that organizations face and result in a lack of shared understanding of NFRs, *triggers* that later prompt organizations to adopt a number *strategies to manage NFRs*, more specifically, I uncovered four of the nine practices from these three organization, and that the *lack of shared understanding* that organizations face can be *accidental* or *essential*. I also developed the relationships between these concepts in the theory. From the three organizations I initially studied, all four of the initial *practices in managing NFRs* in the first iteration of the theory were found to be prompted by a *trigger*, as the organizations *reacted* to a particular stimulus to implement a practice; although I also hypothesized that organizations may or at least should be more proactive in building a shared understanding and not simply respond to a trigger.

After the initial Iceberg Theory was formed, an opportunity to collaborate with a fellow researcher, Laura Okpara, who was researching a similar topic arose. In particular, Laura was performing a related case on a fourth organization, Delta, that provided the opportunity to reinforce and add to the Iceberg Theory. Laura's case study followed a homogeneous approach to my initial three case studies, except that during data analysis findings were compared from the fourth case study with the elements and relationships in my theory. In addition, the data collection was *not*

influenced by the Iceberg Theory, so any resulting construct was unbiased. The intention was to independently confirm, refute, or add to the elements and relationships in the initial version of my theory. Laura's case study developed five additional practices that were not part of the first incarnation of theory. Together, we were also able to identify that organizations may proactively implement a practice to build a shared understanding unprompted, without the need for a particular trigger.

Thus, in this dissertation, I present a process theory, which "explain[s] the behaviour of intelligent agents taking actions to reach goals, but the agent chooses its own sequence of actions" [28] in building, maintaining, and managing a shared understanding of NFRs in CSE.

1.8 Contributions

The contributions of this dissertation include answering RQ1, *What is the interplay between a shared understanding of NFRs and CSE?* and RQ2 *How does an organization build, manage, and maintain a shared understanding of NFRs in CSE?* Using the results and insights developed from answering these two RQs I constructed the Iceberg Theory, aimed at both practitioners and academics alike, to describe how an organization manages the shared understanding of NFRs in CSE,

Three significant contributions were the result of my research on RQ1 and the interplay between a shared understanding of NFRs and CSE:

1. I add to the scarce, yet much-needed, empirical evidence on the shared understanding of NFRs in software engineering, including contributing factors and practices towards avoiding it,

2. I raise awareness, based on empirical findings, to challenges that CSE brings to the shared understanding of NFRs, and

3. I provide insight for practitioners on documenting and communicating NFRs to avoid a lack of understanding in CSE.

The contributions resulting from RQ2 on understanding how an organization builds, manages, and maintains a shared understanding of NFRs include the empirical evidence of the practices and challenges faced by organizations dealing with NFRs in CSE and the implications drawn for research and industry. In addition,

my research heightens the awareness of academics and practitioners of the unique relationship, opportunities, and trade-offs that CSE offers to NFRs. In particular,

1. how organizations using CSE can manage NFRs, for example by offloading subtasks of NFRs to third parties,

2. managing an NFR comes at the cost of certain trade-offs, such as the loss of control over an offloaded NFR, or a decrease in the shared understanding of the NFR, and

3. the importance of configurability as an NFR and the amount of investment required to manage it.

Finally, the insights from my research on RQ1 and RQ2 were used to develop the final, and ultimate, contribution: The Iceberg Theory, which describes the unique, complex, and intricate relationship between the shared understanding of NFRs and CSE, including details on how CSE affects the shared understanding of NFRs. The Iceberg Theory also specifies best practices to build, manage, and maintain a shared understanding in CSE, including the relationship between CSE and a *lack* of shared understanding, in addition to how an organization may identify and recognize *when* to build a shared understanding.

1.8.1 Publications

The findings and contributions from this dissertation have been published, in chronological order, in:

[1] C. Werner, Z. S. Li, N. Ernst and D. Damian, “The Lack of Shared Understanding of Non-Functional Requirements in Continuous Software Engineering: Accidental or Essential?,” in *IEEE 28th International Requirements Engineering Conference (RE)*, Zurich, Switzerland, pp. 90-101, 2020.

[2] C. Werner, Z. S. Li, D. Lowlind, O. Elazhary, N. A. Ernst and D. Damian, “Continuously Managing NFRs: Opportunities and Challenges in Practice,” in *IEEE Transactions on Software Engineering*, pp. 2629-2642, no. 7, 2021.

[3] C. Werner, “Towards a Theory of Shared Understanding of Non-Functional Requirements in Continuous Software Engineering,” in *Proceedings of the ACM & IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 300-304, 2022.

[4] C. Werner, L. Okpara, K. J. Stol and D. Damian, "A Theory on the Shared Understanding of Non-Functional Requirements in Continuous Software Engineering," *Pending Review in ACM Transactions on Software Engineering and Methodology*, 2023.

1.9 Structure

The structure of this dissertation is as follows: Chapter 2 describes relevant background and related literature; Chapter 3 outlines the research method taken throughout this dissertation; Chapter 4 details the research undertaken to answer RQ1 chronicling the interplay between a shared understanding of NFRs and CSE is; Chapter 5 contains the in-depth analysis to understand and answer RQ2 on how an organization build, manage, and maintain a shared understanding of NFRs in CSE; Chapter 6 describes in great detail the Iceberg Theory and its components; Chapter 7 provides a discussion of the importance and implications of the Iceberg Theory, including concluding remarks; finally, publications in Appendix A, research artifacts in Appendix B and C, and the Bibliography complete the dissertation.

Chapter 2

Background and Related Work

Research into how CSE deals with NFRs is limited. In general, we have a reasonably clear understanding of NFRs, and how agile software development and requirements engineering affect one another. However, these studies tend to be focused on larger contexts, e.g., in distributed teams or multi-team agile initiatives [12], and make little mention of CSE. In particular, the relatively recent rise of CSE practices, such as automated verification and build-on-commit, have the potential to greatly impact how NFRs are managed, as CSE emphasizes automated verification, rapid iteration, and shared codebases. This chapter details related work in the areas of software quality, rework, shared understanding, requirements engineering, NFRs, CSE, and finally, work on NFRs and CSE.

2.1 Software Quality & Rework

Studies conducted on the general reliability of software date back to the 1970s. While these studies recognized the problems of software quality, that often manifested in the maintenance phase, the vast majority of these studies focused on the software development process. Before 2000, most software was developed using a waterfall model, consisting of a series of sequential phases, i.e. design, implementation, verification, and maintenance. The verification and maintenance phases, at the later stage of the long-drawn-out software life cycle, represent the most costly phases to fix software quality issues, also known as rework.

It wasn't until Swanson coined his three dimensions of maintenance, corrective, adaptive, and perfection, that the problem of software rework was directly studied

and classified [30]. After which several various definitions and classifications of rework have been proposed.

Regardless of the classification and the large costs associated with rework [31], there is consensus that some rework could *not* have been prevented [6] and is acceptable. A certain amount of rework is desirable [32], as no rework could indicate developers are not performing their jobs with due diligence, popularized in the ideas captured by Ries [33]. Some rework might be due to misunderstood NFRs, decreasing software quality, which can worsen and compound rework upon rework in later development phases [34]. False shared understanding [1] of NFRs can lead to substantial technical debt and ultimately force an organization to perform major rework [35].

2.2 Shared Understanding

Shared understanding refers to a form of communal knowledge or belief [1], and is a critical factor when working in a collaborative environment as different people may have diverging understandings that are not mutually shared. Shared understanding is a critical success factor in achieving high-quality software that meets stakeholders' needs [19]. Creating a common shared understanding is an important aspect and a challenge to consider when attempting to develop high-quality software [36], especially in requirements engineering [37]. Yet there has been a lack of "systematic treatment or classification of the different forms of shared understanding, neither in general nor in a software engineering context" [1].

Glinz and Fricker [1] developed a model describing four quadrants of shared understanding consisting of two forms, explicit and implicit shared understanding, and two categories, true and false understanding. Explicit shared understanding is captured in the form of documents or artifacts and represents how this specified knowledge is interpreted. On the other hand, implicit shared understanding represents how non-specified knowledge, e.g. assumptions or opinions, is understood. A false shared understanding, whether it be implicit or explicit, implies that there is some form of misunderstanding, or a lack of shared understanding, amongst a group of people, perhaps caused by a misinterpretation, misconception, misbelief, or some other form of confusion. Many times a lack of shared understanding may go unnoticed, adding further complexity and cost to the problem. Finally, a true shared understanding indicates that a group of people maintain a common perception of a particular subject.

The most important goal of RE is to create a shared understanding between the development team and its stakeholders [38]; moreover, the practice of creating and maintaining a shared understanding is not well-established [39]. Previous work has shown that a misunderstanding of requirements at the beginning of development resulted in substantial rework [40]. Unfortunately, shared understanding is passive, informal, and unstructured [1]; the result of which negatively affects software quality and necessitates rework [41], which is further compounded by the cross-cutting nature of NFRs. A recent survey [42] stipulates creating a common shared understanding of customer expectations is the top challenge in agile adoption and remains understudied. [1]. However, a shared understanding of NFRs is even harder to study, due to the cross-cutting nature of NFRs, and has not been studied in the rapidly changing environments of agile or CSE, yet it remains a critical success factor to software and merits further attention.

2.3 Requirements Engineering

Requirements engineering has always been an important aspect of software engineering, although it may not necessarily receive the attention it deserves, especially in a CSE context. It is now accepted that RE in agile organizations follows a just-in-time requirements engineering approach [43, 44]. Just-in-time RE practices deal with requirements as needed, rather than upfront, for example, by adding issues to the backlog and then delving into the requirements for that issue only once, or if, it becomes part of the iteration plan [45]. Frequently members of an organization's leadership team are the only people with detailed knowledge of the requirements [43], which aligns with philosophies such as Ries's lean startup approach [33]. The focus is to release often, gather feedback on the new features delivered, and prioritize work for the next release as needed. The implication of this just-in-time RE is that a) little upfront analysis is done and b) requirements analysis is taking place in the verification and experimentation phases, usually once the software is released to customers. This has implications for how an organization is measuring and analyzing its requirements. Given that NFRs are difficult to analyze and understand, even in highly planned, upfront RE processes, suggests an even bigger challenge in just-in-time settings. At the four organizations I studied, agile practices included weekly sprint planning meetings, cross-functional roles, short iterations, and rapid releases.

One problem with neglecting requirements is requirements-related rework: the extra work needed to fix problems in software due to poorly understood requirements. Rework is extremely costly [5] and accounts for 40-50% of the effort on software projects [6]. Some rework may be due to an avoidable lack of shared understanding, which I define as an accidental lack of shared understanding. In CSE, however, some amount of a lack of shared understanding is essential: inherent in dealing with the essential complexity of software [46], it captures the unknown unknowns [47] and represents desirable learning and feedback [33].

2.4 Non-Functional Requirements

Another critical success factor in software is NFRs [48, 9]. NFRs, such as privacy, have the potential to derail a software product, for example, if the system violates elements of Europe's General Data Protection Regulation (GDPR). However, in CSE, similar to agile software development, FRs are the primary focus, while NFRs are neglected [49]. CSE, especially in small, agile organizations, focuses on automated, rapid release of working software, and often leads to short-term functional prioritization [50].

On the surface, NFRs are often simplified as system qualities, such as the *-illities*: usability, reliability, maintainability, etc.; however, upon deeper analysis, NFRs may have a significant influence on a system's overall design and architecture [16, 51]. NFRs tend to receive a lot of attention in safety-critical systems, or at large organizations. However, for startups in web application settings, including the organizations I studied, NFRs, such as software reliability or system performance, are also critically important.

Despite their importance to success [1], NFRs are ambiguous, not well documented, shared in an informal matter, and not well-understood [52, 53]. What some organizations think are NFRs are FRs [13]. It is well known that NFRs are cross-cutting across multiple disciplines, especially in a short, fast-paced iterative software development context [16]. Thus, refining an NFR to a "level of detail that is measurable and valuable and then [finding] design fragments that can be completed within the release cadence" is difficult [16].

NFRs are also inherently difficult to verify or validate [54, 55]. Finally, the inability to appropriately decompose an NFR into manageable and digestible chunks further increases the difficulty in managing an NFR in CSE. The difficulty impedes the forming of a common ground of shared understanding of that NFR and can result

in costly rework. Many empirical studies have been performed on the challenges of NFRs in practice [56, 57, 58]; however, these studies do not focus on identifying a lack of shared understanding of NFRs in CSE.

Moreover, *how* and *who* verify an NFR is another important aspect to consider. Previous work found that NFRs are often difficult to verify and validate [14, 54]. Manual verification is often the most common choice to verify NFRs [10]. The study by Ramesh, Cao and, Baskerville [59] suggested that in agile RE, NFRs often get de-prioritized in small organization settings and NFRs are ill-defined and ignored, e.g. "We have no specific test of stability. We just test for functionality and see if it stays up" [59]. This makes dealing with NFRs at a later date more difficult, and it typically introduces technical debt.

Finally, NFRs for organizations in agile settings are often "informally stated, contradictory, difficult to enforce during development, and very hard to validate" [54]. For example, 75% of NFRs in a recent study from Eckhardt, Vogelsang, and Mendez were describing system behaviour [13]. Agile RE risks neglect of NFRs since user stories focus primarily on features [60]. Finally, the wide-ranging and extensive NaPiRE study [61] found that "unclear / unmeasurable non-functional requirements" were one of the top problems respondents had with requirements in their small organizations [62, p.11].

Most recently, research has examined how best to incorporate NFRs into agile settings. For example, Alsaqaf, Daneva, and Wieringa [12] examined the way large, multi-team projects managed NFRs. The main challenge was the way NFRs cross-cut teams in larger organizations. Whereas in smaller organizations, cross-cutting is not yet an issue. They also report on organizations that are not operating in continuous settings, which I do.

Finally, a recent systematic literature review found that environments practicing CSE are underutilized for NFR verification and there is a very low ratio of industrial studies in NFR verification in continuous practices compared to academic studies [63].

2.5 Continuous Software Engineering

Continuous Software Engineering has emerged as the next evolutionary step from agile methods. Agile methods emerged in the early 1990s, emphasizing iterative and incremental development, feedback, and responsiveness. Several practices such as Continuous Integration and later Continuous Deployment and Continuous Delivery,

originated as part of agile methods, but have shifted the tempo from agile sprints to a continuous, anytime pace. Fitzgerald and Stol labelled this phenomenon continuous *, which extends to all practices within the software development lifecycle, including continuous planning and continuous use [64].

CSE leverages automated tools and development practices to allow an organization to rapidly and efficiently, make automated and continuous changes to their software products. A major shift that differentiates CSE from agile methods is the continuous nature, versus the steady cadence of 2–3-week sprints common in agile methods such as Scrum. As in agile, CSE seeks to deliver new functionality and features rapidly, perhaps daily. This, however, has major consequences for non-functional requirements, which are typically crosscutting in a code base and cannot be easily fixed.

RE, including NFRs, is very different in a CSE context, whereby many of the common best practices are not followed and requirements may be informally captured just-in-time for development [45]. While there are other definitions of continuous software development, in this dissertation I use Martin Fowler's influential blog post on the topic [65]. Fowler's definition of continuous practices includes maintaining a single source code repository, automated verification, automated builds, fast builds, and automated deployment [65]. Fowler also specifies sound organizational practices, which include each developer creating a commit at least once a day and keeping the build process transparent for all stakeholders [65].

Given the prevalence of NFR challenges, irrespective of the software development context; the challenges are exacerbated in an agile context through the fast and incremental CSE methodology. Several studies [66, 67, 68, 69, 70, 71] have explicitly studied the management of NFRs in a CSE context; however, none of them focus on the associated lack of shared understanding. Feitelson describes how Facebook uses an open-source tool, Perflab, to provide metrics that Facebook monitors to evaluate the performance of its system [69]. While metrics are a key strategy to operationalize and create a shared understanding of an NFR in a CSE, Perflab is self-proclaimed as "not supported" and "not turn-key" [69].

For software security, Jaatun argues that proper attention to incident management, including the involvement and education of developers, can help alleviate security issues in CSE [66]; although this is strictly for security and is primarily focused on the Building Security In Maturity Model [72].

Other studies [67, 71, 68] examine how to architect a system to be used in CSE, primarily focusing on deploying as a serverless cloud-based platform. These studies

focus on NFRs, such as scalability and deployability, that have a significant influence on a system's architecture, including positives, negatives, and trade-offs. While these studies have published and disseminated information on *how* to architect a system in CSE, these studies do not look at how to create a true shared understanding of the explicit NFRs in CSE. Although the initial motivation behind CSE improved the shared understanding between development and operations [73], little is known about how CSE, and its associated emphasis on automation and rapid, frequent iterations, may impact shared understanding of NFRs. Furthermore, no empirical evidence exists that measures the effect of shared understanding of NFRs [1], let alone in CSE. Thus, the goal of this dissertation is to fill the gap in research by explicitly focusing on the state of the practice of shared understanding of NFRs in CSE organizations.

While continuous practices have been shown to enhance requirement traceability [74], the research that investigates NFRs has been limited as it mostly focuses on the verification of NFRs. NFRs are often neither comprehensively verified nor automated. In particular, a study on verification found that an organization may not provide sufficient time to verify NFRs [75] because NFRs may require more time to verify. Even when organizations do verify NFRs, the amount of automated tests for NFRs is limited, and an organization may require manual verification to verify them [76]. Similarly, in a study of large organizations that primarily relied on traditional forms of software development with some continuous integration, NFRs were found to be verified late in the development cycle.

Late verification of NFRs may cause severe side effects, such as re-factoring architecture at a stage when an organization is busy preparing for a release of software [77]. Finally, in a study at Facebook and OANDA [78], it was found that small iterations and rapid releases of software may have caused "resource and performance creep". In other words, unbeknownst to an organization, its software may degrade in performance, efficiency, and capacity, which represent three different NFRs. One aspect often missing in those prior works is a description of tools used for managing NFRs. For example, Savor et al. [78] mention that NFRs were watched by measurement tools but make no mention of what tools or how the tools were operated. Chen [79] wrote that the verification of NFRs in the context of continuous development was an important aspect for future study. This dissertation develops a theory to describe how organizations manage the shared understanding of NFRs in CSE.

2.6 Theory Building in Software Engineering

Until recently, little research has focused [22] on generating a theory to describe phenomena based on empirical evidence observed within the software engineering discipline [21]. Theories in software engineering are encouraged to help researchers answer relevant questions [22] that are useful to software practitioners [21]. Herbsleb and Roberts's theory of coordination on collaboration in software engineering [23] and Hoda's grounded theory describing how organizations transition to agile development [80] are two such examples. Given the practical relevance of the topic of interest in this dissertation, a shared understanding of NFRs, I argue that a theory that describes the key mechanisms may be of considerable interest.

As in other fields, the SE discipline has also struggled with the question of what exactly constitutes a theory. Theories can be categorized in different ways. First, Gregor proposed a taxonomy based on the goal of theories: analysis, explanation, prediction, explanation and prediction, and design and action [81]. Second, theories can also be distinguished in terms of their scope, ranging from *general* or *grand* theory [82], to mid-range theory [83], to *theory fragments* [84]. Third, theories can be distinguished as variance theories or process theories [28]. A variance theory is normally expressed in terms of constructs and relationships that can be quantitatively evaluated using, for example, structural equation modelling. A process theory tends to be more descriptive, explains and describes *how* a particular entity changes or happens [28]. In this chapter, I present a process theory, which "explain[s] the behaviour of intelligent agents taking actions to reach goals, but the agent chooses its own sequence of actions" [28] in building, maintaining, and managing a shared understanding of NFRs in CSE.

Chapter 3

Research Method

This dissertation is built upon a mixed-methods approach due to the complex and real-world nature of the phenomenon of shared understanding of NFRs in CSE, which hereinafter shall be referred to as shared understanding. To develop the foundations of a theory describing a rich and complex socio-technological phenomenon, such as shared understanding, there is a precondition that the context within which the phenomenon occurs *must* be considered to play an important role in the phenomenon itself. A case study represents a difficult technical challenge and requires a fine-tuned research lens to analyze a rich, contextual phenomenon. In particular, through empirical studies, this dissertation examines how each organization builds, manages, and maintains a shared understanding, and how it overcomes the associated lack of shared understanding. Thus, a case study research method is an appropriate method for studying such a phenomenon [85].

As recommended in literature [86, 24], I collected data from different sources, allowing for data triangulation to produce richer and more reliable results. I used several data collection methods (see Table 3.1): interviews with developers and managers, focus groups, observation, and technical artifact collection. In situ research, such as the research undertaken in this dissertation, produces a large amount of data, in particular, qualitative data, which may be challenging to make sense of [28]. A multitude of coding techniques were leveraged to make sense of large amounts of qualitative data [87], and hence turn the data into the resulting Iceberg Theory.

The research in this dissertation is divided into four stages: preparation, RQ1, RQ2, and developing the Iceberg Theory. The research is based on an empirical multi-case study [24] using a mixed-methods approach, including the triangulation of quantitative and qualitative [25] data, through a multi-case study, and immersive

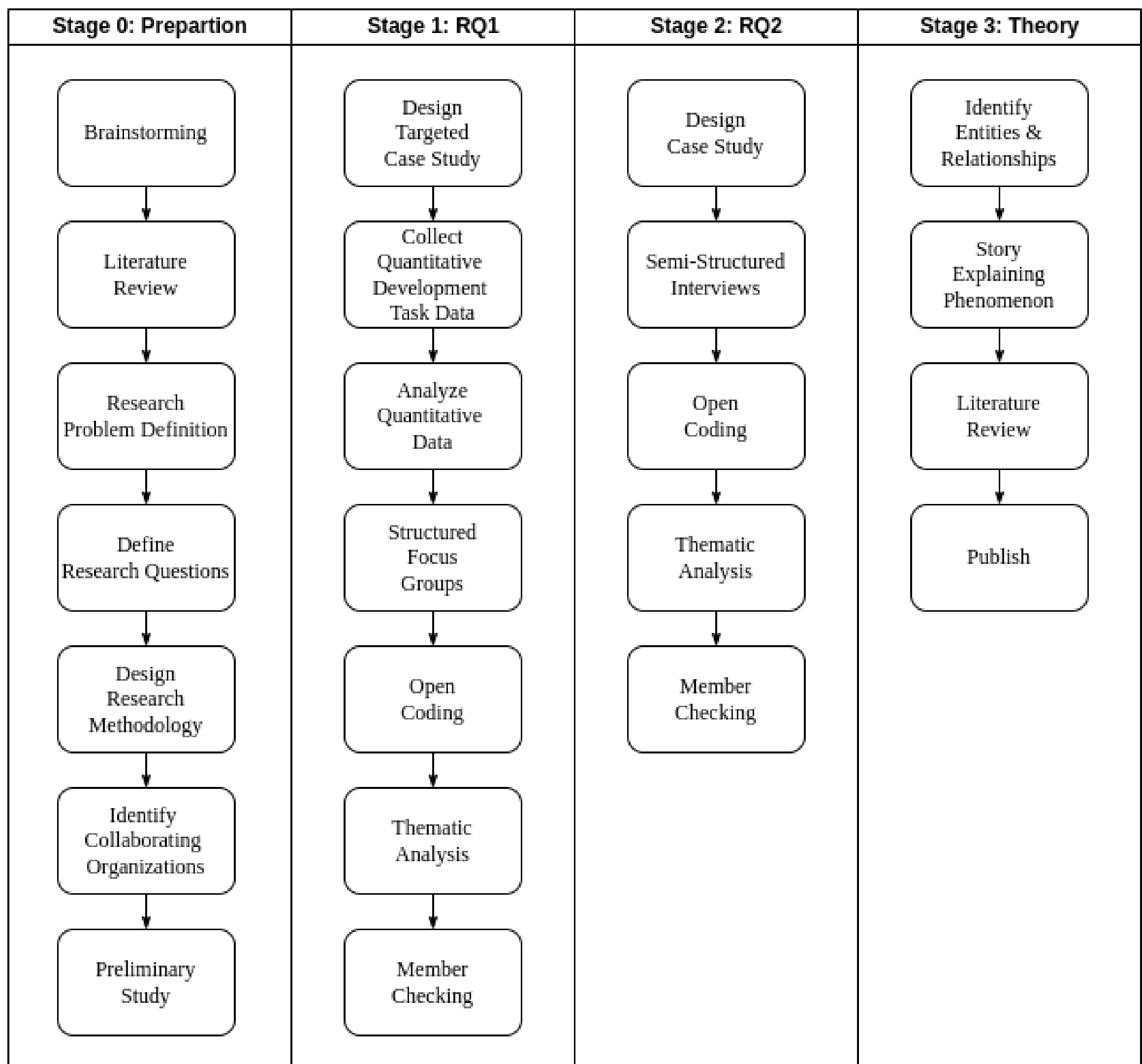

Figure 3.1: Summary of Research Methodology

Table 3.1: Data Collection Activities

Timeline	Activity
Feb–May 2019	Observations at Alpha, Beta, Gamma; 27 informal conversations at Alpha, 42 at Beta, 31 at Gamma
May–June 2019	5 interviews at Alpha, 8 at Beta, 5 at Gamma
Jan–Feb 2020	Analysis of 445 artifacts at Alpha, member checking of 49 tasks with managers and developers; Analysis of 357 artifacts at Beta, member checking of 36 tasks with managers and developers. Analysis of 1,720 artifacts at Gamma, member checking of 89 tasks with developers and managers
	2 Focus Groups (3h each) at Alpha to discuss 15 tasks; 2 Focus Groups (3h each) at Beta to discuss 15 tasks; 2 Focus Groups (3h each) at Gamma to discuss 11 tasks
Nov 2021–Jan 2022	Observations at Delta 34 informal conversations at Delta
Feb–May 2022	11 Interviews at Delta

techniques [26]. The empirical studies performed as part of this dissertation form the basis and initial evidence required to develop the Iceberg Theory on the shared understanding of NFRs in CSE, including the rich relationships between the components of the Iceberg Theory. I briefly describe my research method next.

3.1 Preparation

Before embarking on such a grandiose research project, a topic must first be chosen. As part of the topic selection, several brainstorming sessions were held with various members of the Software Engineering Global Interaction Lab[1] at the University of Victoria. After some initial high-level ideas were discussed, an intensive literature review was performed to identify a gap in the current research community. Namely, it was identified that there was a substantial gap focused on the intersection of NFRs,

[1] https://thesegalgroup.org

shared understanding, and CSE — hence the research problem was formally defined and adopted. Next two research questions, RQ1 and RQ2, were proposed, as previously discussed:

RQ1: What is the interplay between a shared understanding of NFRs and CSE?

RQ2: How does an organization build, manage, and maintain a shared understanding of NFRs in CSE?

3.1.1 Site Selection

A site is a single unit of study and may be represented by an organization, culture, group, artifact, or repository [28]. In this research, a site represents a single organization whose primary source of revenue is generated by developing and selling software solutions. A key criterion for selecting case study organizations is an organization's relevance to the research, the ability to produce rich data and believable explanations, and be practically accessible [88].

With a high-level research goal defined, the next step was to identify candidate organizations as potential sites. Local personal and industrial contacts and local trade shows were used to compile a list of potential candidate organizations. Every effort was made to ensure these prospective organizations represented a broad range of software domains (e.g. e-commerce, content management), the number of employees, the age of the organization, and the maturity of CSE practices. The collaborating organizations were selected based on availability and willingness to grant access to and develop research artifacts, all the while each organization represents a different software domain and exhibits mature CSE practices for this study.

As part of the research protocol approved by the University of Victoria and NDAs, the names of each organization and interviewee are anonymized. I selected four local organizations using CSE: Alpha, Beta, and Gamma to answer RQ1 and RQ2, and the Delta case study was included during the theory development stage; each representing a site in this study. Alpha works in the crowdsourcing industry collecting large amounts of data on a daily basis. Beta provides an e-commerce platform for customers distributed worldwide. Gamma is an online content provider, including advertisements. Delta develops loyalty, rewards, and referral programs for other organizations. Table 3.2 summarizes the selected case companies.

Each organization uses CSE and represents leading organizations conducting best practices with respect to CSE [89] and NFR knowledge management. However, as

Table 3.2: Selected Site Descriptions

Org.	Year	Staff	Domain	Key NFRs
Alpha	2013	44	Data collection and analytics	Configurability, security, scalability
Beta	2010	56	E-commerce platform provider	Usability, performance, reliability
Gamma	2011	36	Content provider including online advertisement management	Performance, reproducibility, configurability
Delta	2013	29	Loyalty, rewards, and referral programs	Usability, maintainability, performance

part of this research, I collaborated with a fellow researcher to determine that each organization has a slightly different definition of what *exactly* CSE means to each respective organization [89].

Each organization represented an independent case in this multi-case study, each with real-life contexts, feeding into the empirical study to develop the theory on a shared understanding. As part of each case study, access to development artifacts, such as code repositories and issue trackers, was granted. In addition, each organization granted access to employees to participate in casual interactions, interviews, and focus groups. Each organization has been in business for 8-10 years and has 40-100 employees. Each organization implements CSE, including automated builds and testing and varying levels of automated deployment. One important trait is that the bulk of their respective development teams are co-located, with few exceptions. This distinction is important, as it eliminates the plethora of problems associated with globally distributed software development, in particular, RE [90, 91, 92].

3.1.2 Preliminary Study

Studying the intricate and complex relationship between shared understanding and CSE represents a significant and difficult technical challenge. An important aspect of studying a socio-technological problem, such as shared understanding, is the context under which the phenomenon is observed. Thus as part of the multi-case study, multiple periods of time were spent embedded at each organization, learning about each organization's respective products, processes, and business [26]. This immersive

technique consisted solely of participant observation; while embedded at each organization, there was no active contribution to any organizational activities. However, it was more than a fly-on-the-wall, as there were interactions between researchers and employees at each organization. The insights gained from the immersive observations provided the necessary context and constructs to understand and evaluate the relevant NFRs to each organization and what processes and tools were employed to handle these NFRs. These immersive visits were a key component to the successful evaluation of the shared understanding at each organization.

3.2 RQ1: Shared Understanding and CSE

The first stage of research in this dissertation consisted of collecting data from quantitative and qualitative sources to investigate and describe a rich and deep understanding with respect to RQ1: What is the interplay between a shared understanding of NFRs and CSE? To conceptualize the lack of shared understanding of NFRs, rework development tasks were used to trace how the lack of shared understanding manifests. To analyze the lack of shared understanding of NFRs as it materialized in rework, an in-depth analysis of development artifacts was performed at Alpha, Beta, and Gamma. The artifacts were drawn from the organizations' source control systems; each artifact represented work on either a bug, feature, story, or epic, depending on the context of each organization.

A total of 19,060 artifacts were collected: 9859 from Alpha, 2629 from Beta, and 6572 from Gamma. Of the 19,060 artifacts, 2,522 development tasks were analyzed whereby 348 were identified as exhibiting a lack of shared understanding of an NFR. Each development task was then analyzed and systematically coded with respect to three criteria: 1) NFRs, 2) lack of shared understanding, and 3) rework. This coding was developed and validated in an iterative and collaborative process with the collaborating organizations over two full months. A round of member-checking was performed with each respective organization.

Of these 348 tasks, 174 were discussed with each respective organization through member checking to validate that these tasks did indeed exhibit a lack of shared understanding of an NFR. Overall, a total of 41 tasks were confirmed to exhibit a lack of shared understanding of NFRs. These 41 tasks were used as the basis for the in-depth analysis through further focus group sessions at each organization.

Focus group sessions were scheduled at Alpha, Beta and Gamma to discuss each organization's share of the 41 development tasks exhibiting a lack of shared understanding of an NFR. Each focus group lasted three hours and involved two researchers, one developer, and one manager. I clarified the meaning and definition of key terms, including NFRs, shared understanding, avoidability, and their mutual relationship; clarification of these key terms was essential to establish a common foundation for the focus group. A number of topics were discussed for each development task, including the factors contributing to a lack of shared understanding, which NFRs were more prone to a lack of shared understanding, and what amount of lack of shared understanding could be avoided. The recordings from the focus groups were transcribed and verified. The qualitative data from the in-depth, focus group sessions was used in the triangulation of contextual data from the preliminary study and quantitative data collected from each organization's task management tool.

Thematic analysis [27] was performed on the qualitative data obtained from the transcripts of the focus groups, and employed other researchers to avoid researcher bias. An open coding [93] approach is employed to develop a purely inductive codebook [27], which minimizes a coder's ability to force a bias of any particular hypothesis. The resulting codebook is then further scrutinized to form higher-order themes, which were member-checked with respective organizations. The higher-order themes were used to answer RQ1.

3.3 RQ2: Challenges and Practices

The second stage of research in this dissertation involved in-depth semi-structured, technical interviews to answer RQ2: How does an organization effectively build, manage, and maintain a shared understanding of NFRs in CSE? Given the intricate nature of this understudied research domain a qualitative research method was employed, which is suitable to study non-technical aspects, including socio-technical, that complement traditional quantitative software engineering research methods [25] and to develop empirically-driven theories in software engineering [29].

The second stage was conducted through semi-structured interviews with a broad set of individuals representing various departments at each organization. The recordings from the interviews were transcribed and verified. These interviews form the base qualitative data and were examined using an established data analysis method,

thematic analysis, to identify themes and patterns in the data [27]. Thematic analysis was performed by coding the sentences transcribed from the interviews.

The analysis involved inductively developing codes from the raw transcripts and then further identifying themes that describe the rich relationship between shared understanding and CSE. Additional trained researchers from the University of Victoria aided with the coding and thematic analysis to avoid researcher bias by a single coder. The open coding approach [93] was used to minimize the bias any one particular coder may have. Throughout the coding the constant comparison method was employed, whereby codes were added, removed, and merged based on the discussions between the coders. Similarities and differences between codes were discussed to group codes into clusters [94], whereby each cluster has a distinct higher-order theme. Finally, to increase the credibility and validity of the research findings, member-checking was performed [95] with the study participants to verify and validate whether the findings resonate with the context of their organization.

3.4 Developing the Theory

The final stage of this dissertation is to develop, document, and describe the Iceberg Theory on the shared understanding of NFRs in CSE. The approach to developing the theory was based on results and insights developed from RQ1 and RQ2. The empirical investigations, while focused on industrial software practices, develop the necessary components towards a theory on the shared understanding of NFRs. In particular, Ralph's guidelines for developing a process theory in software engineering were followed [28].

Thematic synthesis [27] was employed to uncover the various components and relationships of the theory, see Table 3.3 for explicit examples. Themes were constructed by contemplating how different codes may be combined to form overarching themes. The theory that was developed through this research defines the entities and relationships, including a detailed story describing the entities and their relationships. These entities and relationships were established through the long-lasting multi-case industrial study focusing on four local organizations, representing the most substantial effort in this research. Together the results of the results and insights from RQ1 and RQ2 manifest as either entities or relationships towards the theory and ultimately build the underlying constructs describing the Iceberg Theory.

In addition, during the development of the theory, I had an opportunity to collaborate with a fellow researcher, Laura Okpara, who was conducting similar research based on RQ1 and RQ2 from my research. Laura's research followed a homogeneous approach to my RQ2 on Delta, a similar organization to Alpha, Beta, and Gamma. The opportunity to collaborate began by assisting her with her data analysis and comparing her findings from Delta with the elements and relationships in the initial renditions of the Iceberg Theory. Our collaboration provided the opportunity to confirm, refute, and add to the existing elements and relationships in the Iceberg Theory.

3.5 Ethics

As per the University of Victoria regulations, the research method had been documented and approved by the Human Research Ethics Board (Certificate 18-203 and 21-0656) to ensure that all aspects of this research involving human participants meet the ethical standards required by Canadian universities and national regulatory bodies.[2]

3.6 Risks and Limitations

Due to the interpretative nature of qualitative research, many of the typical limitations found in quantitative research do not necessarily apply, namely internal validity, reliability, objectivity, and external validity. The advantage of using qualitative research offers a more holistic view to obtain much richer and deeper data [96]. As such, Lincoln and Guba's quality criteria are used to conceptually evaluate the proposed theory, including their recommended techniques to overcome the specified limitations. In particular, credibility, dependability, confirmability, transferability, and authenticity will be discussed in sequence [97, 98].

3.6.1 Credibility

Credibility evaluates how trustworthy and believable the results are given that a qualitative researcher is subjectively interpreting the perspectives of participants, as opposed to objectively documenting actual reality. To ensure the credibility of this

[2] https://www.uvic.ca/research/conduct/home/regapproval/humanethics/index.php

research prolonged engagement, persistent observation, peer debriefing, triangulation, and member-checking were used.

Prolonged engagement ensures that a sufficient amount of time is spent at each organization to provide important contextual information, avoid misinterpretation, and deduce the difference between what is useful information and simply misinformation. Furthermore, a prolonged engagement builds trust between members of the organization and researchers, adding to the credibility of the researchers' interpretations of the data. Through prolonged and immersive visits a substantial knowledge base of context, culture, and a repertoire enables well-informed and accurate interpretations of case study data.

Persistent observation ensures sufficient in-depth studies are performed to uncover sufficient details. To address this threat, a significant amount of time was spent at each organization, amounting to roughly 4 months at each organization. In addition, a representative and diverse set of employees from each organization were engaged. Finally, more than one researcher was present to mitigate researcher bias.

Peer debriefing validates that a particular researcher's interpretations are not biased. Peer debriefing was utilized throughout each interview and each focus group was attended by at least two interviewers. In addition, multiple researchers were used in the open-coding approach throughout the thematic analysis, to ensure the analysis was fair, consistent, and neutral.

Triangulation of data uses more than a single source of data, or interpretation, to optimistically draw the same conclusions from the same data. The two forms of triangulation used were: investigator triangulation, whereby more than one researcher was involved in the data collection and analysis, and various sources of data, including quantitative and qualitative through various methods including immersive techniques and multi-case studies.

Member-checking validates that the interpretation of the data is a real representation of the participant's actual perspectives. Member-checking was performed through ordinal feedback from participants and any finding that did not receive positive support was noted and subsequently dropped.

3.6.2 Dependability

Dependability assesses whether the findings would be the same if the study was replicated in a similar context [99]. Thick descriptions are used to describe the research

method, circumstances, setting, context, and findings. In addition, detailed notes are taken, including audio recordings, with candidate permission, where possible through all in situ observations and interviews. Furthermore, each finding is triangulated from at least two different data sources.

3.6.3 Confirmability

Confirmability evaluates the levels and variety of biases that prejudice research findings. Research is susceptible to sampling, non-response, response, confirmation, and researcher bias.

Sampling bias represents the biggest threat, especially when sampling a specific group, such as software organizations. Performing a multi-case study of no less than four distinct, unique organizations establishes some level of confirmability. Furthermore, each prospective organization is surveyed to assess their respective level of NFR understanding. At each organization, a diverse sample population of employees is selected; although due to the subject being investigated the sample must be primarily technical. Finally, definitions and examples are presented to each participant to create a shared understanding of particular terms.

Response bias was mitigated by anonymizing interviews and focus groups, developing rich relationships through in-depth knowledge and trust, and assuring participants confidentially and the ability to speak freely. More than one researcher was employed to promote discussion between and amongst interviewees.

Confirmation bias was overcome by using open-ended questions encouraging respondents to provide their own points of view. Interviews were conducted in a semi-structured manner to allow extemporaneous questions. As interviews progressed, questions were refined based on prior interviews.

Researcher bias was mitigated by involving multiple researchers through each step of the research. In addition, open coding [93] was used to develop a purely inductive codebook [27], thus minimizing a coder's ability to enforce a bias towards any particular hypothesis. Finally, triangulation, peer debriefings, and member-checking were all utilized to minimize any particular bias.

3.6.4 Transferability

Transferability assesses to what extent the results could be applied to other contexts. Although a case study lacks explicit transferability, a case study remains the most

appropriate method to study a contemporary, real-life context phenomenon [24]. To maximize transparency two techniques are used: thick descriptions [100] and purposeful sampling. Detailed thick descriptions are used to narrate the circumstances, setting, and findings used throughout this research, which are possible due to detailed notes and audio recordings. A purposeful sampling of participants ensures the prospective participants offer the best-in-class practices to provide highly relevant and meaningful data points for subsequent analysis.

Table 3.3: Theory Concept Development

Raw Interview Text	Research Observation Notes	Theory Concept
"We have a metric to monitor [the NFR]. On a deployment, it is actually part of my responsibility to monitor over the day and see how those numbers are relative to the previous day."	The developer discusses how they assign a metric associated with a particular NFR and how they track the metric over time.	*Practice: Put a number on the NFR*
"Availability is actually an interesting one, do you spin up Amazon services in Virginia, or do you avoid that because that's their biggest region and so things sometimes go wrong there?"	Manager talks about how they become reliant on a cloud provider to provide an NFR, but they are also at the mercy of the cloud provider.	Let someone else manage the NFR is *susceptible* to loss of control
"I can't imagine a scenario when that would become a high enough priority to ever implement [reproducibility]."	Developer indicates that reproducibility was not a high enough priority to invest in until it was too late.	*Challenge: Deprioritization of NFRs*
"It's a scalability issue, but it's not prioritized, so we need to build more domain knowledge so it doesn't fall by the wayside."	Developer and manager indicate that scalability was an issue, but was not a high enough priority due to a lack of shared understanding.	Deprioritization *results in* lack of shared understanding
"The developers working on the code didn't know the impact causing an outage, this is one of those things that people don't pay too much attention to until they need to."	A lack of shared understanding of transparency resulted in a loss of service for a customer.	Lack of shared understanding *manifests as* a service disruption
"Big problem for one user, one very big user, because for [redacted customer's name] everything that company says is critical."	A usability issue was misunderstood until a very large customer complained to the CEO causing the organization to put the NFR in source control	A customer demand *prompts* putting the NFR in source control.

Chapter 4

Interplay Between a Shared Understanding and CSE

To answer RQ1: what is the interplay between a shared understanding of NFRs and CSE, empirical evidence was gathered from Alpha, Beta, and Gamma to explore and investigate the relationship between a shared understanding of NFRs and CSE. Four sub-research questions were developed to further deepen the understanding of the unique relationship between shared understanding of NFRs and CSE.

RQ1.1: What contributes to a lack of shared understanding of NFRs?

RQ1.2: Which NFRs are most associated with a lack of shared understanding?

RQ1.3: What amount of a lack of shared understanding of NFRs is accidental versus essential?

RQ1.4: What practices can be employed to avoid a lack of shared understanding?

4.1 Research Methodology

To answer these research questions, I collaborated with Alpha, Beta, and Gamma to investigate the lack of shared understanding of NFRs in relation to rework. In particular, the in-depth examination of forty-one NFR-related development tasks identified as rework formed the basis of the understanding portion of this research. I was granted access to these tasks through a collaborative process with Alpha, Beta, and Gamma, to identify reasons for a lack of shared understanding, including whether it

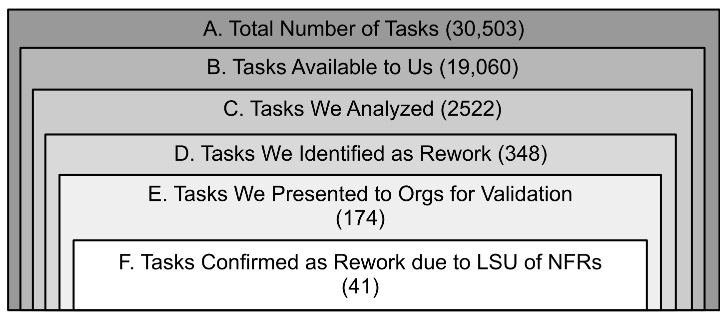

Figure 4.1: Sampling of rework tasks due to lack of shared understanding of NFRs (Total # of tasks - # of tasks at Alpha (A); Beta (B); and Gamma (G))

was accidental or essential, and which practices to employ to avoid a lack of shared understanding. This conceptualization of the lack of shared understanding as rework development tasks was used to shed light on the complex relationship between shared understanding, rework, and CSE.

The selection process and key characteristics of Alpha, Beta, and Gamma are described in Chapter 3.1.1 and the preliminary study is discussed in 3.1.2.

4.1.1 Data Collection

To analyze the lack of shared understanding of NFRs as it materialized in rework, an in-depth analysis of development tasks was performed at each organization, where each task represented either a bug, feature, story, or epic, depending on the context at each organization. Each development task was analyzed and systematically coded in relation to three elements: 1) NFRs, 2) lack of shared understanding, and 3) rework. This coding was developed and validated in an iterative and collaborative process with these organizations over two full months.

Identification of NFR-related Development Tasks that Represented Rework

Each organization granted us access to their respective task management tool (Jira) under a non-disclosure agreement. All available tasks were exported, and associated attributes through a script over one week and spent another week understanding the data. The resulting data contained 30,503 tasks across these three organizations (Figure 4.1 Box A). Due to the large number of tasks and limited time with each organization, the task sampling was performed over a period of four weeks to achieve

a more practical and manageable number of tasks for the analysis, as summarized in Figure 4.1.

With the help of each organization, tasks associated with projects where employees were *unavailable* for the focus groups were removed, reducing the number of tasks to 19,060 (Figure 4.1 Box B). Next, the most recently closed tasks in the last 12 months were analyzed, totalling 2522 tasks (Figure 4.1 Box C), to ensure that the validation of NFRs and rework with employees was feasible. The vast majority of the four weeks was spent on identifying which of these 2522 tasks represented rework, I 1) analyzed each attribute of a task, title, summary, description, comments, assignee, date, estimated time, and identified keywords, such as refactor, rework, and improve, and 2) examined references to previous tasks. This analysis resulted in 348 tasks that *are believed* to represent rework (Figure 4.1 Box D).

1st Member Checking and Validation of Rework Tasks

Each task was then confirmed if it represented rework and, more importantly, if that rework was due to a lack of shared understanding of NFRs. A number of meetings were held at each organization to discuss the selected tasks; these meetings were always attended by 1) two researchers, 2) one developer, and 3) one manager, where each employee had a considerable amount of breadth and depth of technical knowledge. Each interview started by clarifying the definition of NFRs, shared understanding, rework, and the relationship between the three to ensure consistency. Due to partner time constraints, only 174 of the 348 tasks identified as rework were discussed (Figure 4.1 Box E). Overall, the organizations confirmed 41 tasks as rework due to the lack of shared understanding of NFRs (Figure 4.1 Box F). These 41 tasks were used as the basis for the subsequent in-depth analysis through further focus group sessions at each organization.

Focus Group Sessions on Rework Associated with Lack of Shared Understanding of NFRs

Scheduled focus group sessions were then scheduled at each organization to discuss each organization's share of the 41 rework tasks. The factors contributing to the lack of shared understanding (RQ1.1) were examined, including which NFRs were more prone to lack of shared understanding (RQ1.2), what amount of lack of shared understanding could be avoided (accidental) versus essential in the complex relation-

ship with rework (RQ1.3), and what practices could be employed to avoid a lack of shared understanding (RQ1.4). Each focus group lasted three hours and, similar to the validation of rework tasks, involved 1) two researchers, 2) one developer, and 3) one manager. Two focus group sessions were performed at each organization. The base set of questions for each of the 41 tasks can be found in Appendix B.1. The definition of NFRs, shared understanding, rework, avoidability, and the relationship between the four were defined and clarified similarly to the 1st member checking phase. For example, I ensured that each participant equally understood avoidable in the sense that the organization could have prevented the lack of shared understanding by taking action before the rework task arose. These sessions were, with permission, audio-recorded.

4.1.2 Data Analysis

To answer the research questions, the data analysis triangulated contextual data from the preliminary study, quantitative data collected from each organization's task management tool, and qualitative data from the in-depth, focus-group sessions on the 41 rework tasks due to a lack of shared understanding of NFRs. The key to the analysis of the rich, rework task data was the contextual, organizational knowledge acquired during the preliminary study at each organization.

To analyze the qualitative data, the recordings of the 41 focus-group sessions were transcribed, taking one week. Thematic analysis [27] was then performed on these transcripts, spanning a duration of 4 weeks. An open coding [93] approach was used to develop a purely inductive codebook [27], which minimizes a coder's ability to force a bias of any particular hypothesis. In the initial coding phase, a transcript was independently coded by two coders, after which an agreement session was held to discuss the codes, consolidate the codebook, and calculate Cohen's kappa coefficient. This continued until the kappa value stabilized above 0.6, or substantial inter-rater reliability agreement [101], which occurred at the 20th transcript. Each of the remaining 21 transcripts was coded by a single coder and reviewed by a different coder. Throughout all coding the constant comparison method was used; codes were added, removed, and merged based on the discussions between the coders. The final codebook took two weeks to compile and had 48 codes, where 85% of the codes were used in the first 10 transcripts. The complete codebook is in Appendix B.2.

Table 4.1: Summary of Factors Contributing to LSU of an NFR (RQ1.1)

Theme	Associated Codes
Fast Pace of Change	EducationalRework JustGetItToWork LackOfResources LackOfTests ThirdPartyIntegration TradeOff
Lack of Domain Knowledge	BusinessContext DifferentPriorityScale ChangeHereBreaksThere ChangingRequirements LackOfKnowledge
Inadequate Communication	Ambiguous Communication InformationOverload

To answer RQ1.1, RQ1.3, and RQ1.4 thematic synthesis [27] was employed over a two-week period to abstract themes from the derived codes. Themes were constructed by contemplating how different codes may be combined to form overarching themes. Three themes emerged as factors contributing to the lack of shared understanding of NFRs (RQ1.1). Similarly, several themes emerged related to the accidental versus essential nature of shared understanding (RQ1.3), including practices to avoid a lack of shared understanding (RQ1.4). A 2nd round of member checking was performed with the three organizations to validate these themes. For RQ1.2, the number of references to particular NFRs for each task was counted.

4.2 Findings

4.2.1 What Contributes to Lack of Shared Understanding of NFRs? (RQ1.1)

Participants were asked if they knew why there was a lack of shared understanding of NFRs for each of the 41 tasks (Q4). Through the data analysis, three themes emerged: fast pace of change, lack of domain knowledge, and inadequate communication. The themes and associated codes are in Table 4.1.

Fast Pace of Change (33 of 41 tasks)

When an organization is moving rapidly there is little emphasis on NFRs due to the immense pressure to release a product, e.g. a manager at Beta discussing why usability is not on the forefront *"time constraints, had to be out. When it was originally written we just needed it to work."* Similarly, when considering deployability, a manager at Alpha said *"when you're first starting out it's move fast and break things; get things out the door. There's fallout, could be minimal fallout, but something you take a chance on."* Both Beta and Gamma suffered from extensibility issues, recently performed a rewrite, and optimism surrounds the ability to increase shared understanding of the new system, e.g. *"our new [redacted] so a lot of our debugging like live to on the server is going to go away so that will change if you'd come and interview me in two weeks and see a completely different conversation" (Developer at Beta).*

Lack of testing was also a result of moving too fast, as these three organizations admitted to not performing adequate testing, of either FRs or NFRs due to insufficient resources, time *or* people, e.g. *"we weren't doing any kind of testing, a/b testing or anything like that to assess performance or whether these changes were having a positive or negative impact" (Developer at Gamma).*

Lack of Domain Knowledge (29 of 41 tasks)

Domain knowledge is the business-specific context required to compete in a particular domain. Glinz and Fricker's paper on shared understanding [1] mentions domain knowledge as a problem to building shared understanding. The participants were clear that understanding the market you are developing for is key to continued success for the entire organization, e.g., *"it's important to have that business context of how people are going to use it and that's probably the biggest stride being made, is just trying to expand internally what we know about business context so that new people coming in [are aware]" (Developer at Alpha).* Lack of domain knowledge can cause a lack of shared understanding of NFRs if an organization is entering a new, unfamiliar market where even a basic understanding would be beneficial, e.g. *"if we had known more about the industry, which I mean again even just a user of the industry" (Developer at Beta)* with respect to having a narrow view and not consider extensibility. Alternatively, lack of shared understanding could occur due to the unknown unknowns [47], e.g. *"lack of scope, we never considered that this would be a thing"*

(Manager at Beta) concerning the scalability of how one of their customers would utilize a particular feature.

It was observed that an organization's difficulty in adapting to new horizontal markets with respect to localization techniques created maintainability issues, e.g. *"start an app early on for North American companies requiring one [format]. As your app becomes more popular you start getting requests for new countries that use [other] formats" (Manager at Beta).* Even if domain knowledge is well known, the level of comprehension or understanding may change, which may bring new knowledge about a particular NFR, e.g. *"they changed because of increased comprehension of the problem space we were trying to solve" (Developer at Gamma)* in regards to acknowledging they did not grasp a extensibility requirement.

Another insight from this research is that the priority of an NFR is a result of the level of shared understanding of that NFR. Reaching consensus on the priority of an NFR may be difficult due to a discrepancy in perceived importance between different units within an organization, as a developer at Gamma stated *"definitely NFRs tend to be lower priority, or at least perceived as by the business as low priority"*. Additionally, NFR were assessed as low importance *until* the NFR becomes such a problem that the system no longer functions, e.g. *"I would say performance is never a high priority until somebody says it doesn't work for me in which case you've graduated from an NFR to it is not functioning for me" (Developer at Beta).*

Finally, organizations also face difficulty when dealing with complicated NFRs, such as privacy [102]. These three organizations are particularly concerned about privacy as they are collecting and managing a lot of customer data. Privacy can be challenging as a developer may not have the legal expertise to determine the correct course of action to comply with privacy law. Hence, legal consultants are frequently relied upon to provide guidance. However, despite legal consultation, Gamma's compliance with GDPR was hindered due to the inexperience of the developers. Even if a developer acquired knowledge of the GDPR, there was no systematic method to develop shared understanding.

Inadequate Communication (15 of 41 tasks)

Developers told us that, due to the implicit and cross-cutting nature of NFRs, they were not aware of the value of communicating NFRs until it was too late, e.g. a manager at Alpha explained *"nobody explained to him what reliability was"* and *"then*

somebody gave them a code review or whatever and said, oh you should have done it this way." Developers explicitly reported communication problems. For example, developers working on the same problem, almost simultaneously, in isolation of each other but not communicating the *right* information, which created maintainability issues e.g. *"two people [developing] independently, they talk to each other well, but you're still going to approach problems differently and hit different areas of the system and not know everything that's going on. So there was no notion of or expectation of maintainability" (Manager at Beta).*

Communication is the key to ensuring all developers have a shared understanding of NFRs; however, communication is often lacking, e.g. *"it's just a performance matter, but yeah, just the people involved were unfamiliar with what they're using" (Developer at Beta).* Communication is also time-sensitive, it might *"take a little while before they sort of enter the common knowledge of the company" (Developer at Alpha)* according to a developer at Alpha in reference to extensibility and maintainability.

In other instances, a developer makes a false assumption that everyone else has the *same* understanding of terminology, knowledge, or perception of an NFR and the need to communicate the NFR is overlooked. For example, the definition of quality and releasable is different between *two* developers, *"it's a lack of understanding of like what is quality software and what does releasable mean?" (Manager at Gamma).*

Communication breakdowns can also lead to challenges in disseminating information. For example, extensibility and performance suffered at one organization due to *"two different developers building the front-end and the back-end at different times. Only like a week or two apart, but they weren't communicating well enough to each other and it wasn't enough like top-down planning of that" (Manager at Gamma).* As a result, high-priority rework occurred to rectify the effects of the lack of shared understanding.

4.2.2 Which NFRs are Most Associated with Lack of Shared Understanding? (RQ1.2)

Each organization was asked which NFRs were associated with each task (Q3). The results of NFRs with at least six or more associated NFR tasks are in Figure 4.2. Each task could be associated with multiple NFRs. The top NFRs: reliability/availability, maintainability, usability, and extensibility, respectively occurred in 17, 16, 15, and 14 tasks.

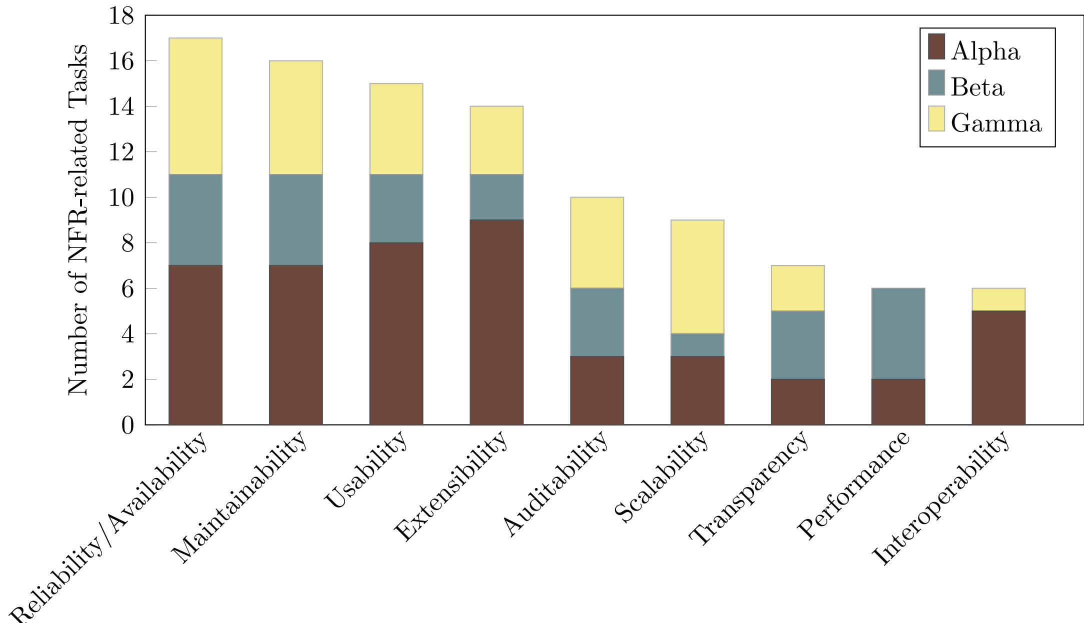

Figure 4.2: NFRs with ≥ 6 NFR-related tasks (RQ1.2)

4.2.3 What Amount of a Lack of Shared Understanding is Accidental versus Essential? (RQ1.3)

Recall participants were asked whether the lack of shared understanding was avoidable (Q7). Participants were also asked how the lack of shared understanding could be avoided (Q5) and why this action was not taken (Q6).

Overall, the responses show 78% of the lack of shared understanding of NFRs was considered accidental across these three organizations. The remaining 22% represents an essential lack of shared understanding.

4.2.4 What practices can be employed to avoid a lack of shared understanding? (RQ1.4)

In response to Q5 and Q6, two primary practices emerged to avoid a lack of shared understanding: standards, and communication and documentation. However, in many cases, the focus groups with the research team were the first time the organization realized the true cost of a lack of shared understanding.

Shared Development Standards (22 of 41 tasks)

These three organizations believed that further adopting more development standards could build shared understanding, particularly for maintainability. This would prevent rework, e.g. *"any time you needed a place to put something we used to toss them on that big stack of unorganized classes until it became a giant object" (Developer at Beta)* The lack of shared understanding means the giant object must be refactored and manifested maintainability issues ultimately created functional problems, as noted by a developer at Beta, *"the rework on it was not only to refactor it [...] but essentially to handle the type of structure used and [redacted] caused problems throughout the app."*

Standards can be applied to multiple facets of an organization, not limited to coding standards. These three organizations are actively standardizing their development methodology. Gamma mandates a developer must create a task to record work exceeding one hour, to help foster a shared understanding. This can help a new employee assigned to improve the deployability of the infrastructure, who may not have the context of the original infrastructure. If the original developer is unavailable, the assigned employee has no shared understanding (e.g. why was Jenkins modified?).

Similarly, a manager at Beta maintains that creating a standardized development process can *"create more shared understanding in more advanced design up-front"* to help avoid usability issues.

Adequate Communication and Documentation (19 of 41 tasks)

Unsurprisingly, inadequate communication was one of the main contributors to the lack of shared understanding, an obvious solution is to have an adequate amount of communication; however, more surprisingly is that this was not occurring often enough. Communication issues were observed between developers and a) other developers, b) support analysts, c) testers, d) product managers, and e) managers. For example, Gamma experienced a situation where *"[developer] was off on his own and he wasn't interacting with the development team or the project management team [and was short-sighted]"*, causing major maintainability challenges. A manager at Gamma said the maintainability issue could be resolved by *"better coordination and communication"*, as the developer was a contractor and isolated from the rest of the development team and *"there was no documentation about it."* Additional documentation, code review, or walk-through was all that was needed to avoid the lack of shared understanding.

Communication across organizational roles is essential to reduce a lack of shared understanding of NFRs. For example, Alpha had a lack of shared understanding of deployability due to miscommunication between management and developers, *"[we are] handing them off to a specific account manager and teaching them how to work through the system."* Additionally, a manager at Gamma indicates *"more collaboration to begin with"* may be necessary to alleviate issues associated with maintainability, extensibility, and scalability. In Gamma's case, the issue arose because there was a lack of communication between a developer working on a feature and the rest of the development team. In consequence, the developer was unaware of the expectations in conjunction with the rest of the software. This issue could be mitigated if more upfront communication was established and the developer had a shared understanding of the development team's practices. Documentation is communication that takes the form of explicit shared understanding. For example, a developer from Alpha believes *"documentation [is desired] because [it gives] you [an] understanding [of] what's going on during that time frame"*. Without documentation, tracking decisions and

actions may be difficult. Moreover, onboarding and training are challenging without documentation serving as a guide.

4.2.5 What Triggered These Organizations to Build a Shared Understanding?

These three organizations were not actively trying to build a shared understanding. In the CSE approach, they try something and assess the viability once it is deployed in the CSE pipeline. If it was not viable (e.g., did not increase revenue) they simply pivot to other features and tasks. As discussed in the preceding section, these organizations acknowledge that rework due to an accidental lack of shared understanding of NFRs is a problem. However, the question of *when* to build a shared understanding of NFRs is tricky. Rather than proactively doing this, these three organizations relied on reactive stimuli. They used different triggers to identify the need for building a shared understanding of NFRs. These triggers were regulatory requirements, accumulating technical debt, the needs of important customers, and disruption of service.

Regulatory Requirements

A core part of each organization's business was to collect and use customer data. The incentive for building a shared understanding of regulatory NFRs, such as privacy and security, is to reduce the potential liability. These laws or regulations, such as the GDPR, are important and comprehensive, as non-compliance could result in crippling financial or legal penalties [102]. Explicit regulatory policies, such as privacy, can be extracted, visualized, and re-published to ensure a higher quality of shared understanding between various stakeholders [103].

For Gamma, the GDPR represented making substantial adjustments not only to the organization's business but also raising the level of shared understanding of privacy. Instead of privacy being an important, but possibly unconsidered quality when faced with time constraints, Gamma needs to ensure that everyone (i.e. not only developers) in the organization has a shared understanding of the GDPR. For instance, third-party services and libraries must be verified for GDPR compliance [102]. Employees need to understand the risk of finding and using an external library without first vetting the library for privacy considerations.

Accumulating Technical Debt

At these three organizations, another trigger was to build a shared understanding of NFRs when the organization had incurred significant technical debt as a result of an accidental lack of shared understanding. This often occurs when technical debt is ignored, potentially exacerbated by the fast pace of CSE, and the belated response by technical staff to recognize the need to change [104]. Refactoring software has many benefits, in particular for NFRs, such as increased reliability, maintainability, and reusability [105, 106]. However, the benefit is usually technical in nature (e.g. the system becomes more reliable) and the impact of refactoring on shared understanding is yet unknown.

Due to the amount and constraints of technical debt, two organizations (Alpha and Beta) rearchitected their entire systems from scratch. The effect of this re-architecture was two-pronged. First, each organization had the opportunity to build an entirely *new* architecture. The new architecture encouraged developers to openly discuss and evaluate NFRs, such as availability, scalability, deployability, scalability, and maintainability, and developed a well-understood, shared understanding of these NFRs. Second, each organization was able to build a shared understanding of not-well-understood, legacy NFRs from the *previous* system in achieving feature parity between the two. Thus, refactoring can potentially *increase* the shared understanding of NFRs.

Needs of Important Customer

As these organizations have a broad set of customers, each of which must be satisfied, it is vitally important to identify and prioritize prospective customers [107]; however, often the focus is on FRs. Two of the three organizations (Alpha and Gamma) were found to build a shared understanding as a result of some input from an important customer or client, usually in the form of a complaint. At Alpha, an important customer was using a very large dataset, which Alpha had not considered or tested. This large dataset caused performance issues due to a lack of scalability in the architecture and resulted in a high-priority rework task. As a result of the high priority, a shared understanding was developed for both performance and scalability across a large number of developers. The shared understanding made developers aware that the original implementation lacked performance measures, and how to fix them, and brought attention to the various scales with which customers were using their system.

Had this customer not been as important Alpha may not have focused the necessary resources to build this shared understanding.

Disruption of Service

Disruptions of service are never a desired trait, as a recent disruption at Amazon was estimated to cost $66,240 per minute of downtime [108]. Disruption of service usually involves a loss of functionality, and subsequent rework to patch the problem. The silver lining in a disruption of service is that these three organizations focused on the lack of shared understanding of NFRs when a disruption of their service occurred. Service disruptions are usually tightly related to an NFR, availability or scalability issues. If the effect of the disruption was small (i.e. one small customer or one small rarely used component) then the shared understanding was not widespread. Participants in this chapter claimed that incident post-mortems as an excellent example of how service outages can somewhat painfully force organizations to confront gaps in shared understanding. For example, Alpha uses post-mortems to increase understanding to focus on the most important aspects of the service.

4.3 Discussion

The goal of this chapter was to investigate the relationship between shared understanding and CSE through empirical evidence. In particular, the lack of shared understanding of NFRs and their effects as traced in the form of rework in software projects at Alpha, Beta, and Gamma were investigated. Some NFRs, reliability/availability, maintainability, usability, and extensibility, were found to be more prone to lack of shared understanding in these organizations (RQ1.2), and the majority (78%) of this lack of shared understanding was accidental, i.e. avoidable, whereas 22% represented essential, i.e. not avoidable, lack of shared understanding (RQ1.3).

From the analysis, three factors contributing to the lack of shared understanding (RQ1.1) emerged: fast pace of change, lack of domain knowledge, and inadequate communication. In addition, these were agile organizations practicing CSE, which this research indicates may pose additional challenges to the management of the shared understanding of NFRs. Two practices shared development standards and adequate communication and documentation, were also identified whereby organizations may benefit in building a shared understanding and awareness of NFRs (RQ1.4). However,

organizations have a fixed set of resources and thus need to strategically prioritize when to build a shared understanding against developing new features. The next discussion point will be on *when* and *how* organizations could employ these strategies to build shared understanding in the context of CSE.

Effectively building and managing a shared understanding of requirements is key to successful software projects. NFRs pose additional challenges given their cross-cutting nature which makes them more difficult to handle, particularly in CSE deployments. The findings suggest that CSE negatively affects the shared understanding of NFRs, resulting in important implications for research, including focusing on the complex relationship of the shared understanding of NFRs in CSE.

Concerning the practical implications, CSE does not absolve an organization's responsibility to maintain a shared understanding of NFRs, including tracking and evaluating how much rework occurs due to an accidental lack of shared understanding. An organization may also proactively engage employees in building a shared understanding of NFRs. To a large extent, the effort that organizations expend on building shared understanding or mitigating the effects of a lack of it will depend on an organization's ability to weigh the cost of the accidental lack of shared understanding of NFRs relative to its need for functional delivery. Development of methods for such assessment, and to address the other research implications mentioned above are worthy of future work.

4.4 Threats to Validity

To ensure the validity of this qualitative research, a replication package[1], including statistics of the analyzed tasks and inter-rater agreement sessions. Unfortunately, due to non-disclosure and ethics agreements, full transcripts are not able to be publicly accessible.

First, there are threats present in data gathering, data analysis, and the results. When gathering data, the effects of respondent bias (the observer effect) were mitigated through two steps. First, the participants were assured that the researchers' presence was not to critique them, but rather to acquire insight into a lack of shared understanding of NFRs in their respective organizations. Second, discussions were held with two participants simultaneously for each focus group and encouraged the

[1] `https://doi.org/10.5281/zenodo.3671463`

participants to not only talk to us but also to each other. To ensure the participants had a shared understanding of particular terms (e.g. NFR, shared understanding, avoidable) the definitions were discussed before the questions. For construct validity, while interviewing participants the term lack of shared understanding was defined and used; however, in the theoretical conceptualization the creative analogy of accidental or essential was used. Transcription tools were used to help transcribe the audio of the discussion. To ensure that the transcription was accurate, a human listened and verified each audio file. One limitation may be the task sampling, as not every task was analyzed. This was due to the limited time I had with each organization and restricted access to the data. However, the organization participants felt the selected tasks were representative of tasks in their context.

With respect to credibility in data analysis, pair coding was conducted until a point was reached where a moderate to substantial level of inter-rater agreement was consistently achieved. Second, two rounds of member checking with these three organizations and participants were conducted. The first round of member checking involved the organizations confirming the 41 tasks were rework as a result of a lack of shared understanding of an NFR. The second round of member checking verified the findings and themes, including factors contributing to and practices to avoid a lack of shared understanding of an NFR. Ordinal feedback (Strongly Disagree-Strongly Agree) on each of the themes and practices was elicited. For all three themes and three practices, the respondents had a median score of Agree. Another threat is the relatively small sample size (41 tasks), partly due to the considerable amount of work required for each task, and the limited time with the three organizations. Finally, for conclusion validity the research implications are phrased as hypotheses, *not* factual conclusions, requiring further research investigations.

Chapter 5

Practices & Challenges with a Shared Understanding

In this chapter, empirically derived insights are used to answer RQ2: How does an organization build, manage, and maintain a shared understanding of NFRs in CSE? Through the empirical evidence gathered from Alpha, Beta, and Gamma I describe the practices and challenges faced by organizations dealing with NFRs in CSE and the implications drawn for research and industry. As part of answering RQ2, two more specific, sub-research questions were formed to explore the practices and challenges faced by an organization building, managing, and maintaining a shared understanding:

RQ2.1: How do CSE organizations manage the shared understanding of NFRs?

RQ2.2: What challenges does CSE introduce when managing the shared understanding of NFRs?

5.1 Research Methodology

Given the exploratory nature of RQ2, an exploratory approach was employed to investigate the practices in managing NFRs at Alpha, Beta, and Gamma. Due to the intricate nature of these RQs, a qualitative research method was used, which is suitable to study non-technical aspects, including socio-technical, that complement traditional quantitative software engineering research methods [25] and to develop empirically-driven theories in software engineering [29].

The selection process and key characteristics of Alpha, Beta, and Gamma are described in Chapter 3.1.1.

5.1.1 Data Collection

Once a confident understanding of each organization was constructed, qualitative data was collected through distinct and semi-directed interviews with eighteen employees in development and managerial roles from these organizations A summary of the interviewees is in Table 5.1. The semi-structured, open-ended interviews were conducted by two researchers with each participant and lasted between 45-90 minutes in person at an organization's office. The semi-structured interviews consisted of a base set of fourteen questions, which can be found in Appendix C.1. The interviews included questions on NFRs and CSE, such as 1) organization definition of an NFR, e.g. *Do you define NFRs? How do you define an NFR? Which NFRs are important to your organization?* 2) organization definition and implementation of CSE, e.g. *Are you familiar with the terms continuous integration, delivery, and deployment? If familiar, how do you define these terms? Does your organization practice CSE?*, as well as how their organization managed NFRs: 3) treatment of NFRs in the context of CSE, e.g. *How do you trace an NFR through continuous deployment? What happens when an NFR fails; is there a feedback loop from continuous development?*

Each interview was structured by going through the base set of questions. Additional probing questions were asked on more in-depth, specific subjects, depending on an interviewee's role or primary work. For example, an interview with a DevOps engineer involved extended questioning on tools or processes used by the engineer's organization to facilitate their CSE. An interview with a front-end developer working extensively on the visual aesthetics of an application often included questions regarding prioritizing and verifying NFRs. Interviews were, with permission, recorded.

5.1.2 Data Analysis and Results Validation

Each recorded interview was transcribed using an automated transcription service and verified each transcription by a human. Subsequently, an established data analysis method, thematic analysis, was employed to identify themes and patterns in the data [27] with other researchers to aid in the coding. The analysis involved inductively developing codes from the raw transcripts and then identifying themes related to practices and challenges in handling NFRs in the three studied organizations. The

Table 5.1: Participant Information and Demographics

Org.	**Participant Number**	**Role**	**Gender**	**Experience at Org.**	**Overall Experience**
Alpha	P1	Dev.	Male	< 2y	< 20y
	P2	Dev.	Male	< 10y	< 20y
	P3	Mgr.	Male	< 10y	< 20y
	P4	Mgr.	Male	< 5y	< 10y
	P5	Mgr.	Male	< 10y	< 20y
Beta	P6	Dev.	Male	< 2y	< 20y
	P7	Mgr.	Female	< 5y	< 20y
	P8	Mgr.	Female	< 10y	< 10y
	P9	Dev.	Male	< 5y	< 5y
	P10	Dev.	Male	< 5y	< 20y
	P11	Dev.	Male	< 2y	< 20y
	P12	Dev.	Female	< 2y	< 2y
	P13	Dev.	Male	< 2y	< 5y
Gamma	P14	Dev.	Male	< 2y	< 2y
	P15	Mgr.	Female	< 2y	< 20y
	P16	Dev.	Male	< 2y	< 5y
	P17	Dev.	Male	< 5y	> 20y
	P18	Dev.	Female	< 2y	< 5y

Table 5.2: Progression of Thematic Analysis

From Raw Interview Text...	*"We have a metric that tracks it to monitor it. So what happens is on a deployment I would monitor it and it is actually part of my responsibility to monitor over the day and see how those numbers are relative to the previous day."*
...To Codes...	Metrics, Implicit, Deployment, Organizational, Manual, NFRPerception, DevOpsPerception, Testing
...To Theme	*Put a Number on the NFR*

open coding approach [93] was used to minimize the bias any one particular coder could have. Throughout coding the constant comparison method was used, whereby codes were added, removed, and merged based on the discussions between the coders.

After each coding session, all coders would meet to discuss each item, which codes were applied, and debate the reasoning behind a particular code and whether that code applied to that item. In the initial coding phase, the four researchers coded every dialogue segment, where a dialogue segment was the unit of analysis, and encompassed the interviewee's answer to a question, of the same transcript (P5) independently before calibrating with the other coders. As part of calibration, the four researchers conducted a mini-workshop to discuss their understanding of the codes after coding each transcript. Coder agreement was defined as any dialogue segment where all 4 coders had at least one code for the segment in common, where synonymous terms were merged into one single code. Since the mini workshops involved extensive discussion on the meaning and use of the codes in the codebook, the coders evolved a shared understanding of each code.

Following the first phase of coding, an additional seven interview transcripts distributed across the three organizations (P3, P4, P9, P12, P13, P16, and P18) were coded. The intention was to further develop codes and consistency in coding. In this stage, each interview was individually coded by two separate researchers; inter-rater agreement levels varied from 64% to 93%, with an average agreement of 85%.

The last phase of coding was divided among the remaining interviews and assigned a single coder for each interview. The number of codes created in the last 10 interviews accounted for only 4 additional codes, thus the codebook was saturated. To ensure that a coder did not miss any important codes during individual coding, an additional sanity check was included where another coder would check the first coder's results. Table 5.3 shows a sample of six codes from the codebook, while the entire codebook (61 codes) can be found in Appendix C.2.

To answer RQ2.1 and RQ2.2 themes were developed based on thematic synthesis [27] of the coded data. Similarities and differences between codes were discussed to group codes into clusters [94], whereby each cluster had a distinct higher-order theme. Each theme represented either a practice (RQ2.1) or a challenge (RQ2.2). Table 5.2 shows an example of relating a raw transcript quote to an eventual theme. Finally, to increase the credibility and validity of the research findings, member checking was performed [95] with the study participants to verify whether the findings resonated

Table 5.3: Codebook Samples

Code Name	**Description**
ConfigurabilityNFR	NFR: Configurability
NFROffloading	Relinquishing technical control / responsibility of said NFR to an external entity
NFRPerception	Individual's perception of NFRs
PerformanceNFR	Performance of the outcome (ie the output) usually measured by time
Metrics	Creating or monitoring of a quantitative value
Tooling	Off the shelf (Kubernetes, Docker, etc)

with the context of their organization. The member-checking feedback was used to revise the findings.

5.2 Findings

Alpha, Beta, and Gamma each deeply care about the shared understanding of NFRs in their software development. Alpha, in the data business, is most concerned about configurability, security, and scalability. These three NFRs are vital to Alpha's business as Alpha processes and stores data from large numbers of users per day on a third-party hosted infrastructure. Beta, a leading e-commerce platform, is primarily concerned with usability, performance, and stability/reliability, as their applications handle millions of commercial transactions per day. Finally, Gamma, an online advertisement content provider, requires performance, revenue, and configurability to ensure that Gamma's infrastructure can effectively deliver content to a wide audience.

Table 5.4 shows the overall ranking and individual ranking for each organization based on the frequency that the code representing the NFR appeared in the data with at least 30 occurrences. The entire codebook can be found in the research artifacts. Gamma's number 2 ranking, revenue, is notably absent from the table due to less than 30 overall occurrences, despite the high number of occurrences at Gamma.

Throughout this chapter, and in Table 5.4, each code or particular finding is lined to the evidence (interview text) as much as possible using quotes as well as spark-histograms. First introduced by Ying and Robillard [109], these histograms represent a compact and quick form of assessment of the findings: each histogram captures the 18 participants interviewed on the x-axis, while the y-axis shows the number of times the given code was mentioned by each participant, relative to the total number of mentions of that code (i.e. normalized). The total number of mentions is also shown following the histogram.

The x-axis is ordered to match Table 5.1. For example, consider the histogram for the code configurability (, 103 mentions), which shows participants 3-6 mentioned this code more often than the other participants, and across all transcripts, this code occurred 103 times.

This section describes the themes, comprised of 4 practices and 3 challenges, derived from the analysis. These practices and challenges are summarized in Table 5.5.

Table 5.5: Practices and Challenges with Shared Understanding of NFRs with CSE

Practices	Put a number on the NFR Let someone else manage the NFR Write your own tool to check the NFR Put the NFR in source control
Challenges	Not all NFRs are easy to automate Functional requirements get prioritized over NFRs Lack of shared understanding of an NFR

Table 5.4: Ranking of NFRs

Rank	NFR	Histo.	Alpha	Beta	Gamma
1	Configurability		1	T12	3
2	Performance		T4	2	1
3	Security		2	T5	6
4	Scalability		3	T5	5
5	Usability		T14	1	T9
6	Reproducibility		T4	T5	4
7	Testability		T6	4	8
8	Stab./Reli.		10	3	T9
9	Availability		T6	T5	T12
10	Maintainability		8	T5	T17
11	Readability		9	T15	T14

5.2.1 Practices For Handling Non-Functional Requirements in Continuous Software Engineering (RQ2.1)

The first research question examines how the shared understanding of NFRs is managed within each organization — the practices they use to handle NFRs in the context of agile teams following CSE. Four practices were identified from the analysis. These were: *Put a number on the NFR, Let someone else manage the NFR, Write your own*

tool to check the NFR, and *Put the NFR in source control.* Each practice is discussed in turn.

Put a Number on the NFR

The first practice for dealing with a shared understanding of NFRs is establishing metrics to help validate and assess a particular NFR. The concept of NFR metrics was frequently discussed during the interviews (, 154 mentions). When an organization tracks pertinent metrics for NFRs, metric indicators collected by the organization's deployment pipeline provide valuable insight into NFRs.

For Gamma, performance is the most important NFR and is primarily tracked through a *"[caching ratio] that we have from [redacted] our caching provider goes [on a dashboard] because that's usually a pretty good indication that something has gone wrong. It also drastically affects performance"* (P14). At Alpha, a drop in response times below a specific metric will cause a decline in performance. As a result, they *"have a certain amount of monitoring set up [...] You're also defining which alarms are set. Therefore, if requests [drop] below [redacted] milliseconds and that's what you want to hold it to. Then that would be codified in the alarm"* (P5). Without setting a quantitative metric, Alpha may not receive a warning that its software experienced a dramatic drop in performance.

Similarly, Beta provides a platform for many retailers and usability is important for those retailers. As part of managing usability, Beta tracks the number of customer actions required to complete a transaction through the use of usability metrics: *"I can tell you that 90% of our customers have less than [redacted] items [...] They'll [say] we know that each [order] requires [redacted] page loads"* (P7). While this may only represent a subset of tasks for usability, Beta views it as part of satisficing usability, in particular by bringing awareness to other teams. While at Gamma metrics are a key component in managing usability, across cross-functional teams, including development and product management by *"showing how many users are using a specific feature, where the feedback benefits developers but our product team in their ability to make their decisions"* (P16).

A critical success factor in putting a number on the NFR is the feedback loop (, 46 mentions) to enable the continuous monitoring of metrics. The feedback loop is an integral part of CSE [64] and is one of the earliest perceived benefits of adopting CSE. For Alpha, which deals with a lot of user data, effectively managing security

awareness is important for its business, e.g. *"at least people who need to be aware of IAM changes are automatically notified"* (P3). At Beta, the feedback loop provides the ability to quickly identify bugs that crash the software: *"I think quick feedback is the core of DevOps so that we will see what broke"* (P6) and *"it reduces risk because things are integrated more frequently"* (P6). In particular, the feedback loop is most useful if it is a *quick* feedback loop and many organizations strive to reduce the time required for a feedback loop to complete, e.g. *"developers really want [the] feedback loop to be tight"* (P13).

Let Someone Else Manage the NFR

The most popular approach to managing shared understanding of NFRs found was to let someone else do it by offloading it (, 155 mentions), where that someone else is typically a large cloud-service provider. Offloading an NFR means that an organization allows another platform or tool (, 262 mentions) to realize, at least part of, the NFR on the organization's behalf.

An organization's ability to focus on core functionalities and behaviour of their software is heightened by offloading the brunt of the work to realize NFRs such as scalability, reliability, etc. However, offloading an NFR is not as simple as flipping a switch to realize an NFR. In many cases some form of configuration is required; furthermore, ensuring the system is ready for configuration, i.e., is configurable, requires a specialized skill set.

Configurability was the most referenced NFR (, 103 mentions). The findings revealed that the three organizations studied made frequent use of configurability tooling such as Docker, Terraform, and Kubernetes. These tools help Beta maintain three separate software stacks through dependency management, which allows developers and testers to easily create or re-create an environment and application, e.g. *"it automatically does it in the Docker compose files. So we've automated a lot of dependency updates and stuff like that"* (P11). Furthermore, cloud providers are favoured by DevOps engineers as they have access to a cloud provider support team to assist with issues, *"cloud providers are awesome. I love being able to just file a support ticket"* (P12), as opposed to dealing with an issue on their own. Finally, configurability tools also help manage reproducibility (, 52 mentions), e.g. *"the most important part is that the builds and the environment they run on and deploy to are defined in a repeatable way or state"* (P5).

security (, 68 mentions) was also offloaded to a third-party service. Security and privacy checks may be intrinsically supported by a third-party service, relieving an organization of the obligation of laboriously maintaining and provisioning these checks on its resources. Beta maintains an extensive customer database and is keenly interested in security and privacy of the database. Security aspects can be offloaded, e.g. *"I believe [security]'s all codified in like the Docker and Kubernetes world"* (P6). Gamma utilizes a security key management system directly from one of the cloud providers.

Cloud providers are heavily relied upon to manage performance (, 88 mentions), availability (, 39 mentions), stability/reliability (, 41 mentions), and scalability (, 56 mentions). For example, at Alpha, *"[your web application is] immediately spread across however many availability zones and nodes as you want"* (P5).

Finally, NFR offloading provides the capability to ratchet NFRs [16], as all three organizations highlighted their ability to increase the amount of resources they consume from their cloud providers if they hit a particular NFR bottleneck. For example, Gamma and Alpha both noted that they can simply pay more to the cloud provider, e.g., *"We just put money in the machine and made it better"* (P5).

Write Your Own Tool to Check the NFR

As opposed to offloading an NFR to a third-party service or tool, organizations also wrote custom, in-house source code, custom tooling, scripts, or manifests (, 78 mentions). Beta and Gamma both codified some usability parameters of their user-facing front-end to manage a portion of usability according to their individual definition and satisfaction, which may not be broadly applicable. This codification was done as part of their source code. They submit the source code, wait for the source to process through the CSE pipeline, and finally observe and verify the result, *"this is how we define usability and make sure that it's there. If you want to change our usability parameters or whatever we change it in source code and then we can test it and verify that it still meets our needs"* (P10).

For a resource-constrained organization, a custom tool is usually a last resort, where the existing off-the-shelf solutions either do not exist or do not sufficiently meet particular requirements. A custom tool may be based on an augmented third-party tool that requires significant modification to meet the specified needs, e.g. *"so*

we used to have [name redacted] dashboards out there ... [but that] didn't give us any application-specific information" (P14).

Some custom tools were used to handle NFRs from an operations perspective to determine availability (, 39 mentions) or stability/reliability (, 41 mentions) of the infrastructure. Other custom tools were developed to help automatically enforce or validate security (, 68 mentions) within a CSE pipeline: *"so there's a lambda [function] that runs when you make a bucket, it triggers and goes 'you didn't encrypt', it turns on encryption, tells you, 'you are an idiot'. Security non-functional requirement!"* (Manager at Alpha).

Put the NFR in Source Control

In each of the three organizations studied, the workforce was constantly changing, typically growing, and the products were experiencing a rapid pace of change. The constant change requires that developers gain a clear understanding of the NFRs of the product so they know how their changes might impact these NFRs. Typically, an NFR is not effectively communicated via natural language documents: *"with respect to codifying something vs documenting it: it's not precise enough it's written in English. It's open to interpretation."* (P17)

The results show developers captured NFRs directly in source control. Codification refers to using code and related artifacts, such as version-controlled JSON configuration files, to capture NFR knowledge (, 92 mentions). For example, automated dashboards monitor health indicators, such as availability, and these explicit rules or triggers that represent metrics of an NFR are in source control. Codification helps set an objective metric for developers who might not have had enough time to acquire the tacit knowledge about what constitutes an acceptable NFR threshold. P17 notes *"my understanding is that we should be able to see our tests in source control in terms of nicely capturing results in a way that I and the other developers can see and say some data is not captured yet"* (P17).

5.2.2 Challenges (RQ2.2)

While Alpha, Beta, and Gamma each have concrete practices for managing NFRs, they still face challenges (, 71 mentions). While it has been shown that CSE further enables an organization to realize NFRs, from the findings the most notable challenges were *loss of control*, *difficulty prioritizing NFRs*, *difficulty using tools and*

tests to automate NFRs, and *challenges with shared understanding*. For each challenge, the findings are discussed in relation to relevant existing literature and highlight areas worthy of further research.

Offloading NFRs to Third-Party Providers Results in Losing Control Over an Offloaded NFR

The emergence of cloud providers, such as AWS, Google Cloud, and Microsoft Azure, offers significant advantages to software development organizations. First, it has encouraged and facilitated small organizations to realize NFRs, perhaps only partially through sub-tasks, that would otherwise not be within their reach, such as scalability. Second, offloading sub-tasks of NFRs, such as scalability and performance, enables an organization to devote additional resources to enhancing the core product [110] and is key to a small organization's business success. Often, finding money to pay for NFR offloading is easier than finding staff or time, especially for small, resource-constrained organizations. Furthermore, there is some notion that some NFRs may be realized and guaranteed through certified quality of service agreements [111], allowing an organization to focus on the core of their business. Third, the utilization of cloud platforms [112], or even simulators [113], allows an organization to easily build otherwise costly environments solely to verify reliability, availability, performance, and scalability.

At the same time, it must recognize that offloading an NFR may not imply the NFR is realized across all aspects of the organization. This is even more important with the prevalence of distributed or micro-service architectures, as offloading an NFR for one particular component does not satisfy that NFR for the entire system. Given the cross-cutting nature of NFRs, one must recognize the limitation of offloading an NFR and carefully plan to ensure that offloading will achieve the desired result. Organizations must be cognizant of the limitations of offloading an NFR.

Two costs associated with offloading an NFR have been identified, 1) the loss of control of the NFR and 2) the potential for vendor lock-in. First, the offloading organization will be at the mercy of the organization taking over that NFR [114]. If an NFR is realized by decomposing that NFR into a series of sub-tasks, then an organization might lose control of those sub-tasks, or the assigned priority of those sub-tasks. In particular, if an organization has offloaded a portion of an NFR, such as availability, to a cloud provider and that cloud provider experiences an issue, such as

an outage, the organization will also experience an outage and hence the availability of the organization's product is now entirely out of their control [115].

Second, with offloading, there is also the risk of vendor lock-in, which occurs when a customer is overly dependent on a vendor, such as a cloud provider, and is unable to switch to another vendor without substantial re-work. Vendor lock-in has long been a problem in the software industry [116]; interestingly, none of the three organizations mentioned anything about vendor lock-in from any of the interviewees. However, neither cloud provider tools nor standards are widely adopted (see Section 7.3) so the potential for vendor lock-in remains and is an area of active research [117, 118, 119].

Fast Pace of CSE Deprioritizes NFRs

Agile methodologies have been shown to risk the overemphasis of FRs at the expense of NFRs [49]. The findings corroborate previous research [78, 50] indicating that the fast pace of CSE increases the risk of deprioritizing NFRs. These results also suggest that neither frameworks nor models produced from research are adopted in practice to assist in prioritizing NFRs.

Since the three collaborating organizations are still growing rapidly, employees balance many other responsibilities, among other potential limits to resources. In resource-constrained environments, NFRs are an easy target to bump from the sprint plan or milestone due to a lack of clarity around how to verify or define an NFR. At Alpha, one aspect of their system is the need for CPU power to process large volumes of data. However, if not enough developers are available to maintain the system, efficiency and performance may degrade over time: *"nobody's looked at this last six months maybe someone should check it out [...] It's a resource management issue and a lot of times you have too many things for too few people."* (P4). An organization may be obliged to make NFR trade-offs with the hope that immediate, short-term success will lead to the ability to remedy the trade-offs, i.e., potential technical debt, in the future. 14 out of 18 interviewees (, 41 mentions) acknowledged the existence or previous existence of trade-offs: *"the sort of startup code today some of it you could consider clever but you do too many clever things then rack up a lot of technical debt"* (P5).

An organization, even an early-stage organization, needs to be aware of these NFR trade-offs so that it can ensure that an important NFR such as scalability is improved when the NFR reaches a low point: *"Some of the core pieces of the system again get*

more love or more time to knock that [technical debt] number down or we just pay more close attention" (P5).

While the developers interviewed indicated that NFRs are important to developers, the perceived importance of NFRs differs for product managers, among others. As such, NFRs were largely left to the developers to self-manage in an ad-hoc manner. Despite the numerous frameworks and models developed through research [120, 121, 122, 123], there exists a gap between industry and practice on whether they can be used to solve this issue of NFR prioritization. This is a significant empirical finding adding to the scarce evidence on how NFR prioritization, or the lack thereof, is handled in industrial versus research settings.

Not All NFRs are Easy to Automate

Some NFRs, such as usability, are intrinsically difficult to verify through automated means. Unfortunately, adopting CSE approaches to development, which means committing and deploying at an extremely rapid rate, appears to make this an even greater challenge. For example, Beta relies on some manual usability acceptance verification for any customer-facing software deployment. As a result, the people tasked with manual verification in our findings suggest that usability becomes a bottleneck in CSE: *"[User acceptance testing is] easier to chunk into one deployment rather than 12 a day."* (P7).

In addition to usability, it may be difficult to write automated tests for other NFRs: *"There's a lot of variability and it's hard to write really good thresholds of what is working. What is not working"* (P14). Although some NFRs may not require much work to automate, based on the interviewees' sentiment, producing the *right* automated tests may require more than one test creation iteration.

Furthermore, purely increasing the number of tests does not equate to higher quality tests: *"I think that testing itself [is a] quality metric. So we're looking at coverage as a possible metric but we're trying to determine a more accurate form because we don't believe that [increasing] code coverage [will] provide us the value [...] if you test all cases that you have 100% coverage, it's not really scalable"* (P16).

Under normal circumstances, it can be difficult to predict a sudden spike in user activity. If Gamma suddenly experiences overwhelming levels of traffic, prior tests for performance and scalability may not suffice: *"I feel like [tests] always [had] a bias toward the happy cases [...] When something does go wrong, it's something horribly*

out of left field [...] How do you prepare for those? How do you think about what left field is?" (P18). Determining the parameters and conditions that would effectively verify the entire problem space of an NFR is difficult.

Lack of Shared Understanding of an NFR

A shared understanding of an NFR implies that everyone involved with the NFR is in agreement with the meaning of that NFR, and its various components. This shared understanding relies on knowledge transfer from the people—such as the product manager or CEO—who created the NFR, to those whose work might affect it. Shared understanding is a challenge for the subject organizations. They faced problems with inconsistency in what gets explicitly stored in source control; with tacit knowledge and a low (bad) *circus factor*[1] [124]; and problems with role siloing.

The three organizations relied inconsistently on documentation for knowledge transfer of NFRs (, 78 mentions). Explicit NFR knowledge transfer (, 55 mentions) occurs when a developer relies on a formal metric or artifact, such as documentation, to frame their understanding of whether an NFR is being met. For instance, in reference to having their infrastructure run by Terraform scripts to deploy using Kubernetes, one of the respondents mentioned: *"a big incentive for me is the idea that I don't become a linchpin and at the same time a bottleneck being the one person specialized in this"* (P3).

The subject organizations do not consistently invest effort or resources in documenting NFRs. While some NFRs are being actively monitored (see Section 5.2.1) or are documented as code within a source control system (see Section 5.2.1), others are not. For example, in reference to the fact that the performance of a feature requires processing to finish within 2.5 hours: *"No, I don't think I have that specifically labelled. I don't think I outlined any specific requirement like that"* (P2).

Implicit knowledge transfer of NFRs (, 29 mentions) occurs when someone on the development team attains a personal understanding of an NFR without relying on an established metric or artifact. For example: *"at the moment it is tacit knowledge and unfortunately when a new developer comes in and starts working on stuff [they struggle]"* (P10). The *circus factor* [124] captures a major problem with implicit knowledge: *"I try not to get hit by a bus. So does [redacted]. Certain parts of the*

[1] To avoid the ugly implications of bus factor in favour of circus factor: the number of people who have to run away to join the circus to hurt the project.

system are maintainable because certain people know how they work versus it being [explicitly] documented" (P5).

Implicit knowledge is often obtained by one or two people who have the overarching view, typically early employees or founders. *"I know a lot of the team leads have a ton of non-functional requirements in their head and how things should work and they're kind of the ones gating what goes out based on those undocumented non-functional requirements. If we were to document those we could get those ideas into the heads of the people actually writing the code, and better the development experience"* (P13). Without transferring the knowledge of NFRs to front-line developers, awareness of the importance of particular NFRs is lost.

While explicit knowledge may be in source control (see Section 5.2.1), role siloing makes understanding the artifacts difficult (, 51 mentions). For example, capturing NFRs in the deployment scripting language Terraform is likely highly useful for team members working closely with deployment and DevOps roles, e.g. *"For anything that I do, [infrastructure as code] ends up in Terraform as a form of documentation"* (P3). However, the value of documentation provided solely by code can vary depending on the developer's context. Developers less familiar with Terraform may have different interpretations of what the script is doing if they can understand the Terraform language in the first place.

5.3 Discussion

The goal of this chapter was to unveil the state of practice in managing the shared understanding of NFRs in organizations that use CSE. The findings presented suggest that an organization may manage the shared understanding through four main practices. These practices are believed to be best practices for managing the shared understanding, as the respondents, overall[2], are very satisfied with how their respective organization manages the shared understanding. While these practices may not be specific to CSE, a special relationship may exist for each that is unique in a CSE context.

While the use of metrics is well-established in the industry, CSE enables an organization to better automate and deploy metrics in a rapid feedback loop. Furthermore, while NFRs are typically more difficult to automate, CSE brings a heightened focus,

[2]11/12 respondents believe shared understanding is well managed

attention, and importance to automating important NFRs. While the lack of shared understanding of NFRs has been commonplace, evidence was found to suggest the CSE has led to a decrease in shared understanding in Chapter 4. Finally, while FRs may be often prioritized over NFRs, the fast-paced environment of CSE exacerbates the deprioritization of NFRs.

The results uncover surprising opportunities that CSE practices offer to manage shared understanding, as well as associated challenges and trade-offs. The importance of configurability as an NFR in organizations practicing CSE is also discussed. The findings represent valuable empirical insights that add to the nascent empirical evidence on utilizing CSE to manage shared understanding of NFRs [63]. Finally, various research directions are proposed for the treatment of shared understanding in CSE.

The effective management of shared understanding is key to successful, high-quality software projects. NFRs themselves are well-known to be difficult to express, let alone manage, partly due to their cross-cutting nature. Since NFRs in the context of CSE have not been sufficiently explored in literature, a qualitative investigation was conducted to gather empirical evidence on how CSE organizations handle the shared understanding of NFRs. Contrary to previous research, the investigation described in this chapter brings insights from three organizations that do manage NFRs using a variety of practices, yet continue to face important challenges and make trade-offs.

While NFRs are difficult, ambiguous, and tough to verify in normal circumstances, following the four practices will allow an organization to NFRs in CSE. In particular, the empirical evidence indicates that a key to reining in shared understanding is to leverage CSE practices, such as the quick feedback loop or the capability to offload NFRs to third parties. However, the peril of realizing an NFR by leveraging CSE is that an organization may lose control of an offloaded NFR, leaving them at the mercy of the third party, or incur a decrease in the shared understanding of an NFR.

5.4 Threats to Validity

Threats to validity in qualitative research typically concern the reliability of the results. The interviewees in this chapter were representative of their organizations with respect to role, gender, and experience. The unbalanced distribution of roles, 12 developers versus 6 managers, and genders, 13 males versus 5 females, is representative of the three organizations' demographics, and unfortunately, there were no other man-

agers or females to interview. The analysis did not reveal differences due to gender, role, or experience; this represents a worthwhile direction for future study.

To mitigate the threat of construct validity, each interview commenced by examining the respondent's knowledge of NFRs. Then explained the NFR concept was explained with examples so that each respondent had a similar level of understanding about NFRs. All participants were proactive and valuable in offering details commensurate to their respective roles and experience with their organization's practices.

As far as the credibility associated with data analysis, the primary threat is in the coding approach, which was described in Section 5.1. Best practices were followed for thematic coding and used the inter-rater agreement process frequently to align coding schemes. Due to the number of coders, multiple codes were allowed to be applied to a unit, thus limiting the ability to apply an inter-rater agreement that would resolve a chance agreement. Finally, this research may be susceptible to researcher-participant interactions, since these were in-person interviews.

As for *analyzability*, a computer-aided transcription service was used, but each transcript was checked against the audio where the transcription was unclear. The open coding approach was utilized to remove the potential bias from coders. Peer debriefings were also performed, including the inspection of deviant codes to verify the analysis and ensure the results were consistent and neutral.

With respect to *transparency*, histograms are used to enhance the thick descriptions of the responses. For reliability, a member checking exercise was conducted to validate the findings with the subjects, where 12 of 18 interviewees responded. Ordinal feedback (Strongly Disagree-Strongly Agree) was elicited on each of the practices and challenges. For all 4 practices and 3 challenges, the 12 respondents had a median score of Agree. One challenge that was originally included had a median score of Neutral, with 5/12 voting Disagree or Strongly Disagree. As a result, this particular challenge was dropped pending further investigation. Additional insight gathered from the member checking (from one Director of Technology at Gamma) was integrated into the discussion of findings. The codebook and analysis scripts are available as a replication package in the research artifacts repository but due to NDA, raw transcripts cannot be shared.

The *usefulness* of these findings is geared towards bridging the gap between research and practice of handling shared understanding of NFRs for CSE organizations. In particular, awareness is raised of areas for researchers to focus on with respect to the current and emerging trends that can enable organizations to realize NFRs. I

recognize that NFRs cannot and should not be grouped together, as the differences between individual NFRs can be as great, if not greater, than the difference between an FR and NFR; I believe that further in-depth studies should be focused on individual NFRs. While the usefulness to practitioners is to help bring focus to *how* an NFR can be realized and the associated pitfalls to each.

Chapter 6

An In-Depth Description of the Iceberg Theory

The goal of the research in this dissertation is to develop a deep and rich understanding of how an organization practising CSE may build a shared understanding of NFRs. The results and insights from the case studies at Alpha, Beta, and Gamma to answer RQ1 on the relationship between a shared understanding of NFRs in CSE and RQ2 on how an organization builds, manages, and maintains a shared understanding of NFRs in CSE ultimately led to the development of the initial Iceberg Theory, that provides a cohesive and comprehensive story on the shared understanding of NFRs in CSE.

A process theory, such as the Iceberg Theory, documents and describes how an entity or concept, in this case, shared understanding, undergoes some change and perhaps persists over time [125]. The initial Iceberg Theory contained the following concepts: all six triggers, all five challenges, the classification of a lack of shared understanding, and four practices: put a number on the NFR, let someone else manage the NFR, write your own tool, and put the NFR in source control. In addition, all of the relationships, except the *unprompted* practices, came from Alpha, Beta, and Gamma. I describe these concepts and their respective relationships to enable an organization to assess its individual situation and provide a path forward to building a shared understanding of NFRs. A visual representation of the Iceberg Theory on the shared understanding of NFRs is depicted in Figure 6.1.

As discussed in Chapter 3.4, as I was developing the initial incarnation of the Iceberg Theory, an opportunity presented itself to collaborate with a fellow researcher,

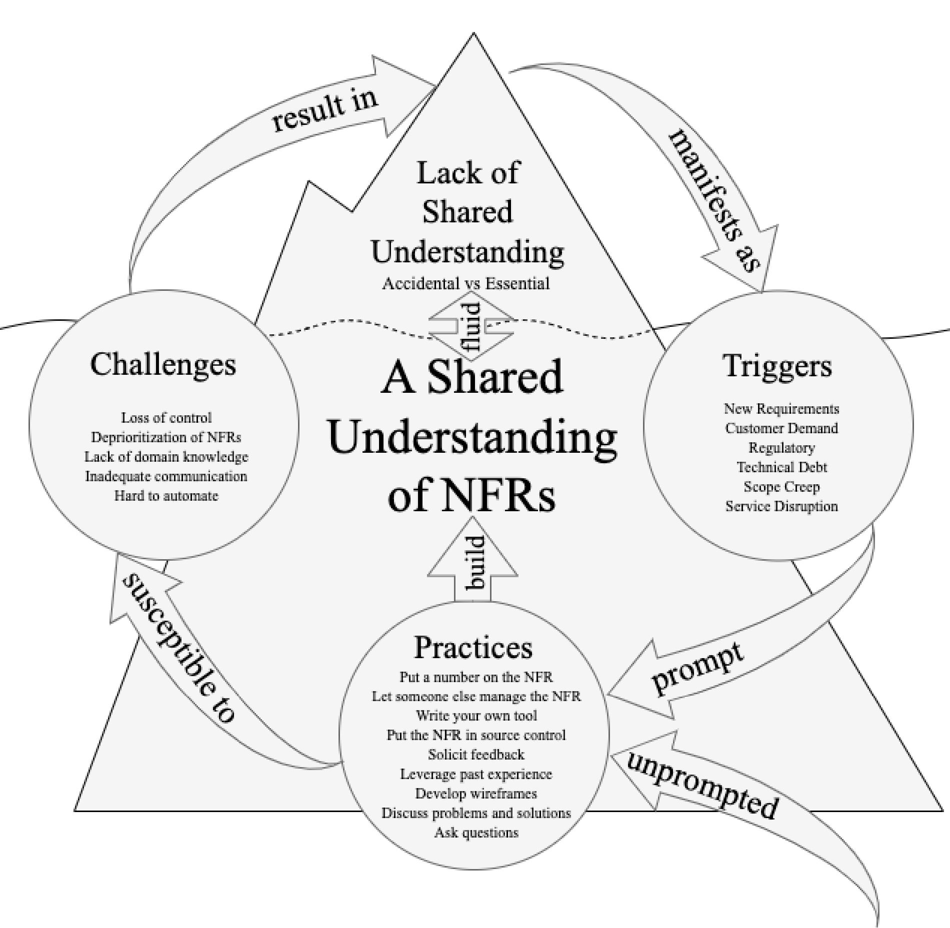

Figure 6.1: The Iceberg Theory

Laura Okpara, who was conducting a homogeneous study. Laura's case study at Delta followed a similar approach to mine, except that during data analysis Laura and I compared the findings from Delta with the elements and relationships already established in the Iceberg Theory from Alpha, Beta, and Gamma. The intention was to confirm, refute, or add to the elements and relationships in the initial version of our theory. The case study at Delta led us to uncover five additional practices; namely, solicit feedback, leverage past experience, develop wireframes, discuss problems and solutions, and ask questions. In addition, we also identified that an organization may proactively implement a practice to build a shared understanding unprompted, without the need for a particular trigger.

The Iceberg Theory has five theoretical concepts: the shared understanding of NFRs, the lack of shared understanding of NFRs, triggers, practices, and challenges; and seven relationships between them: (1) the *lack of shared understanding* manifests as *triggers*, (2) a *trigger* prompts an organization to implement a *practice*, (3) an organization may also implement a practice *unprompted*, (4) a *practice* is susceptible to *challenges*, (5) a *practice* builds a *shared understanding*, (6) a *challenge* may result in a *lack of shared understanding*, and (7) the state of a *shared understanding* may be fluid between a *shared understanding* and a *lack of shared understanding* as an organization continues to build a shared understanding. Together, these concepts and their respective relationships describe a cycle, although there is no specific start or end point. I now describe the key mechanisms of the Iceberg Theory of NFRs.

The visualization of the lack of shared understanding of NFRs is well suited as the tip of an iceberg, as the lack of shared understanding could potentially cause an organization to sink. The lack of shared understanding is a likely starting point where an organization could potentially be encouraged to start building a shared understanding. As ice is less dense than water causing ice to float in water, the tip of an iceberg represents roughly 10% of the total mass of an iceberg. The vast majority of the iceberg is underwater, much like most of the shared understanding of NFRs is hidden behind triggers or challenges. There is a lot more shared understanding to build than meets the eye, or the tip of the iceberg that can be seen. To be truly successful an organization needs to build and maintain an immense amount of shared understanding over time, representing the underwater portion of the iceberg. This immense amount of desired shared understanding needs to be surfaced and disseminated throughout an organization. Finally, there is a constantly fluctuating and fluid line between a

true shared understanding and a lack of shared understanding—hence the wavy line between the lack of shared understanding and a true shared understanding.

Of particular interest is that there is no single state representing an obvious starting point, which enforces the cyclical nature of developing a shared understanding. At any point in time, an organization could, perhaps unconsciously, find itself in any single state, or even simultaneously within multiple states if an organization has developed a mature shared understanding of pertinent NFRs and still building a shared understanding of other NFRs. The goal of this theory is to provide a map describing where to go next and what to expect.

The state of shared understanding of an NFR is not static, but rather dynamic because it can evolve over time, which means there is once again a lack of shared understanding. It is therefore important for an organization to be constantly proactive. In addition, other, potentially undiscovered, NFRs could benefit from the, potentially further, development of a shared understanding. Alternatively, the environment in which the software resides *will* ultimately change, potentially requiring the organization to continue developing its shared understanding of NFRs.

I now present each component of the Iceberg Theory in more detail.

6.1 Lack of Shared Understanding

The lack of shared understanding of an NFR can manifest in a multitude of ways. Glinz and Fricker [1] bisect shared understanding in two ways: 1) implicit and explicit shared understanding and 2) true and false shared understanding. While a false shared understanding certainly fits under the definition of a lack of shared understanding, the definition of a lack of shared understanding is broader. A lack of shared understanding could manifest as a single person, perhaps the founder or a key architect has certain knowledge but it is not shared with others. While a shared understanding amongst one person may work in an organization of one, it does not scale up and suffers from the *circus factor* [124]. Finally, a lack of shared understanding could be something entirely unknown, relevant but unnoticed knowledge; such unknown unknown knowledge can still be relevant and emerge at a later point during a project as an important issue. For example, a manager at Beta noted that *"lack of scope, we never considered that this would be a thing,"* commenting on the scalability of how one of their customers would utilize a particular feature.

This theory defines a lack of shared understanding as either *accidental* or *essential*. In CSE, the *essential* lack of shared understanding represents the desirable learning and feedback that can help uncover some of the relevant but not noticed knowledge. On the other side, an *accidental* lack of shared understanding that is not desired and deemed avoidable. Some amount of lack of shared understanding, *the essential*, is inevitable; however, evidence suggests that the *accidental* lack of shared understanding accounts for 78% of a lack of shared understanding [126]. Obviously, this disproportionately large amount of accidental shared understanding is not desirable, and understanding how an organization can prevent this is a key motivation for the development of this theory on shared understanding.

Continuous software engineering approaches emphasize speed of delivery, and like agile methods, de-emphasize comprehensive documentation that could capture systems knowledge; a manager at Alpha described: *"when you're first starting out it's move fast and break things; get things out the door. There's fallout, could be minimal fallout, but something you take a chance on" (Developer at Beta).*

In CSE, an organization relies heavily on an implicit shared understanding [1], reinforced by this study: *"at the moment it is tacit knowledge and unfortunately when a new developer comes in and starts working on stuff [they struggle]" (Developer at Beta).* However, an implicit shared understanding may be harder to achieve in CSE, due to the fast-paced iterations [126], as a Manager at Gamma shared: *"you build an MVP and there's always going to be room for a reorganization later on as you better understand your problem space."*

Very often key NFR knowledge is held by a small number of people at an organization, usually the founder or employees hired very early on, which increases the *circus factor* and if this knowledge is not transferred then the importance and awareness of the NFRs may be lost. For example, *"I know a lot of the team leads have a ton of non-functional requirements in their head and how things should work and they're kind of the ones gating*[1] *what goes out based on those undocumented non-functional requirements. If we were to document those we could get those ideas into the heads of the people actually writing the code, and better the development experience" (Developer at Beta).* Similarly, a Developer at Delta explained: *"The thing is, non-functional requirements are not always communicated as much, trying to figure out what or not, understanding or communicating expectations around some of those*

[1] A manual process whereby those with the NFR knowledge are required to approve a particular change prior to deployment.

NFRs from stakeholders." In addition, a manager at Alpha said *"I try not to get hit by a bus. So does [redacted colleague's name]. Certain parts of the system are maintainable because certain people know how they work versus it being [explicitly] documented."*

A lack of shared understanding may not be obvious or have any prevalent signs. In many instances, a lack of shared understanding may never be uncovered, but that does not mean it should not be addressed. Other times, lack of shared understanding might surface and manifest in the form of a *trigger*, although perhaps not initially obvious to anyone, in hindsight, they may be overlooked, for example: *"something out of our control changed, or other things have changed since that was put into place due to privacy regulation acts" (Developer at Gamma)*.

Triggers are another key concept of the theory, which I discuss next.

6.2 Triggers to Build a Shared Understanding

Many an organization do not proactively invest in building a shared understanding of NFRs, primarily to focus on functional requirements [127] and simply wait for a *trigger* and respond as appropriate, just-in-time. The realization that there is a lack of shared understanding of an NFR could manifest itself as a trigger, which is the most likely starting point for an organization. An organization should be prudent and prioritize creating such shared understanding for those NFRs that are most critical to their products. Although they do understand the benefits, for example, *"create more shared understanding in more advanced design up-front" (Developer at Gamma)* to help avoid usability issues.

My analysis revealed six triggers as part of this theory: new requirements, customer demand, regulatory, technical debt, scope creep, and service disruption. This list is not exhaustive, and I expect that further research in this area may reveal others. I discuss these six in detail.

6.2.1 New Requirements

New requirements may emerge throughout the lifecycle of a software product that is delivered using a continuous software engineering approach. However, as with any requirement, a lack of shared understanding may not be obvious at first and could be uncovered through prototyping or subsequent rounds of elicitation [128]. For

example, one developer at Delta said: *"You might get halfway through implementation and realize there are a whole bunch of questions that you don't know the answers to. And those questions should have been answered but weren't, either because no one understood at the time or because you've discovered new requirements."*

New requirements can appear as time continually shifts forward, and the environment where the software is operating may change, causing new requirements to surface. Some of these new requirements may have a drastic effect on NFRs, causing substantial rework, for example, *"so they wanted to extend it and could not figure out how without doing all this rework to start with" (Developer at Alpha).*

6.2.2 Customer Demands

Another type of trigger that may prompt the need to build shared understanding is customer demands. Customers may insist, sometimes quite forcefully, that certain behaviour is delivered for a software product that would be unique to that customer. Such a customer demand could require an organization to build a shared understanding, especially if it concerns a major client that was deemed very important. Alternatively, many customers could have a similar request: *"if a lot of customers had this setup, then [priority] would have been high" (Manager at Alpha).*

Frequently, however, a customer demand translates into new functional requirements, rather than NFRs, and the subtleties of NFRs are ignored, often exacerbated by the fact that relevant knowledge is tacit in nature, *"big problem for one user, one very big user, because for [redacted customer's name] everything that company says is critical" (Developer at Beta).* Or perhaps the software was being used in a slightly different environment than intended: *"if your requirement is still the same and your system is still working as it was but it's the actual environment that has changed" (Developer at Gamma).* These cases need to be handled in a similar fashion, first by building a shared understanding with the customer, and then ensuring that the entire organization is able to develop the same shared understanding.

6.2.3 Regulation

Regulatory requirements are stipulated through laws enacted by governing bodies, *"putting into place are privacy regulation acts" (Developer at Beta).* Regulatory requirements, such as the GDPR, can carry crippling financial penalties for non-compliance. Therefore there is a serious incentive for an organization to invest in

building a shared understanding of privacy, which also applies to third-party services or vendors [102]. Rectifying a privacy violation for a system, or sub-system, that was not designed with privacy up front may not be possible, which encourages the creation of movements such as Privacy by Design [129].

6.2.4 Technical Debt

A fourth type of trigger is technical debt, which could accumulate to the point that an organization must invest in creating a shared understanding of NFRs. Technical debt may not be a main concern for an organization, especially in fast-paced CSE environments, and the longer technical debt is ignored the greater the chance of a degrading lack of shared understanding. One developer at Beta commented: *"any time you needed a place to put something we used to toss them on that big stack of unorganized classes until it became a giant object."*

Technical debt may result in a need to refactor, or entirely rewrite, a software system. Refactoring has the positive side-effect of encouraging developers to openly discuss important NFRs for a particular architectural design and ensuring that the underlying cause of the need to refactor, in this case, a lack of shared understanding of an NFR, does not happen again. A manager at Beta illustrated this: *"two people [developing] independently, they talk to each other well, but you're still going to approach problems differently and hit different areas of the system and not know everything that's going on."*

6.2.5 Scope Creep

A fifth type of trigger is scope creep which can be loosely defined as "the growth of the scope of a project's requirements beyond those originally intended" [130]. Scope creep is quite common in CSE, as a large feature is typically broken down into manageable segments that can be completed within the time frame of a relatively short one to two-week sprint; however, often the entire scope may not necessarily be captured in a segment. Scope creep could *"represent the developer who worked on this didn't fully appreciate the full scope" (Developer at Gamma)* or due to the *"lack of scope, [as] we never considered that this would be a thing" (Manager at Alpha).* While CSE is praised for the ability to rapidly and continuously ship [78], the early lack of shared understanding of a particular NFR could be disastrous, as software architecture is often chosen based on the shared understanding of a particular set of NFRs and may be

difficult to re-architecture post-deployment. For example, *"start an app early on for North American companies requiring one [format]. As your app becomes more popular you start getting requests for new countries that use [other] formats" (Manager at Gamma).* Scope creep could potentially alter that set of NFRs and spell disaster if an organization cannot adequately handle the new changes. An organization has to decide on how much and how often scope creep can be accepted, as illustrated by a developer at Delta: *"they had made the assumption that they would be able to go in and update these prints after launch. And we had made the interpretation that everything was just sort of hard-coded, write once and forget it. And so, in the end, it was something that we actually had to push back on the client and say that, making that something that's editable, would significantly increase the scope."*

6.2.6 Service Disruptions

A service disruption can wreak havoc or debilitate an organization's ability to succeed. Service disruptions may occur either due to a series of small incremental changes that have a combined detrimental effect on an NFR, or it could be a single change that has major implications. Disruptions of service may also simply cause a degradation of service, which could drive away customers, jeopardizing future business. Post-mortems are an excellent example of how service outages can force an organization to confront gaps in shared understanding. Said one developer at Alpha: *"The rework on it was not only to refactor it [...] but essentially to handle the type of structure used [in the application] that caused problems throughout the app."*

6.2.7 Triggers Prompt Action

Some number of triggers manifested by a lack of shared understanding are inevitable in large complex systems that are subject to continuous evolution. Triggers prompt an organization to invest in creating a shared understanding, to avoid that trigger from occurring again; however, responding to every single trigger may not be feasible, so an organization must be prudent to focus on appropriately responding to the triggers that represent critical success factors. Triggers often have negative connotations, although a trigger should not only be considered negative. A trigger could have severe consequences: a service disruption could lead to a tarnished public image or grave financial penalties that may spell the end of that organization. However, an organization may recover from a serious trigger that caused a major outage, so long as

the organization learns that particular trigger and invests in building a shared understanding. Creating a shared understanding of an NFR in direct response to a trigger would ideally limit the damage that a recurrence of a lack of shared understanding of that NFR could cause.

6.3 Practices in Building a Shared Understanding

I propose in this theory that developers and architects may either be prompted by a trigger (reactive) or unprompted (proactive) to address the lack of shared understanding. My analysis revealed nine practices for managing NFRs that develop a shared understanding of NFR. I discuss each in turn.

6.3.1 Put a Number on the NFR

Putting a number on the NFR refers to the quantification of NFRs, which in turn allows an organization to validate, assess, and monitor the NFR in question. A major benefit of monitoring and continuous evaluation is the feedback loop that this creates, which is essential in continuous software engineering [64]. At Beta, the feedback loop provides the ability to quickly identify bugs that would lead to system crashes: *"I think quick feedback is the core of DevOps so that we will see what broke [and] it reduces risk because things are integrated more frequently" (Developer).* The feedback loop is most useful if it is *fast*, and many an organization strives to reduce the time required for a feedback loop to complete, as a Developer at Beta indicated: *"developers really want [the] feedback loop to be tight."* At Delta, a fast feedback loop was also seen as positive: *"the good thing about developing at Delta is that our loops are pretty quick; we have a lot of communication, so it becomes quite clear when something isn't being met."*

This practice often comes with dashboards which in turn help with the proliferation of shared understanding of those monitored NFRs. Altruistically, a dashboard may be shared amongst an entire organization, stretching beyond the barriers of the development team, as the whole organization, from finance to marketing, may embrace the Continuous * approach [64].

Many an organization use metrics to monitor common NFRs, such as availability, performance, scalability, availability, and reliability [127]. For Gamma, performance is a vital NFR and is tracked through a *"[caching ratio] that we have from [redacted] our*

*caching provider goes [on a dashboard] because that's usually a pretty good indication that something has gone wrong. It also drastically affects performance." (A)*lpha has *"a certain amount of monitoring set up [...] You're also defining which alarms are set. Therefore, if requests [drop] below [redacted] milliseconds and that's what you want to hold it to. Then that would be codified in the alarm" (Manager)* to enable warnings or alarms based on thresholds. At Delta, performance is a valuable NFR and is primarily monitored through Datadog [131]; metrics like error rates, average response times for API calls, and count of application instances are indicative measures of the state of their systems and applications.

However, metrics and the quick feedback loop allow an organization to monitor NFRs that may be more difficult to quantify, such as usability. Usability, among other NFRs, is often harder to validate through feedback channels like code reviews. However, an organization may continuously validate usability through iterative end-user testing or behavioural testing with stakeholders, by leveraging tools that support behaviour-driven methodology. Beta is able to determine *"that 90% of our customers have less than [redacted] items [...] They'll [say] we know that each [order] requires [redacted] page loads" (Manager).* While at Gamma metrics are a key component in managing usability, across cross-functional teams, including development and product management by *"showing how many users are using a specific feature, where the feedback benefits developers but our product team in their ability to make their decisions" (Developer).*

Additionally, monitoring usability may be accomplished by embedding components in the user interface that report statistics back to the organization, enabling the organization to understand how their software is being used, what features are and are not being used, and how efficient their user interface is to use. Consequently, the organization leverages the quick feedback loop and blue-green deployments to evaluate how commits affect the organization's revenue—all in real-time.

6.3.2 Let Someone Else Manage the NFR

An NFR that is not necessarily part of the organization's core business may be offloaded, especially for a resource-constrained organization. Offloading an NFR allows an organization to focus on those NFRs that are critical to success which is pertinent to resource-constrained startups. Many NFRs, such as availability, scalability, and reliability, can be offloaded, either entirely or partially, to a cloud provider, provid-

ing an organization the best chance of success, as they are able to focus on what is important to their goal while paying little attention to availability, scalability, or reliability. For example: *"[your web application is] immediately spread across however many availability zones and nodes as you want" (Manager At Alpha).*

Obviously, offloading an NFR does not necessarily mean the organization is entirely devoid of any responsibility for that NFR, as perhaps only a portion of that NFR may be offloaded. However, cloud providers are favoured by DevOps engineers as they have access to support when issues arise as opposed to dealing with an issue in-house. One developer remarked: *"cloud providers are awesome. I love being able to just file a support ticket" (Developer at Beta).* Again, CSE practices are crucial to letting someone else manage the NFR, as an organization needs to be able to assess relatively quickly whether the NFR is being satisfied or not.

Maintaining Infrastructure as Code (IaC) enables an organization to configure its software infrastructure and environments [132], also commonly known as the NFR configurability, which is an enabler of many CSE benefits [133]. *"For anything that I do, [infrastructure as code] ends up in Terraform as a form of documentation" (Manager at Alpha).*

IaC enables an organization to achieve configurability, but it also enables the organization to codify other NFRs captured as IaC, e.g. *"it automatically does it in the Docker compose files. So we've automated a lot of dependency updates and stuff like that" (Developer at Beta).* IaC has grown to capture more quality aspects of software, including build, staging, and deployment infrastructure, with very little input, aside from the IaC, by humans [134]. IaC standards also require a specialized, highly-touted skillset, usually in the form of a DevOps or site reliability engineer. Furthermore, IaC helps manage reproducibility, as a Manager at Alpha pointed out: *"the most important part is that the builds and the environment they run on and deploy to are defined in a repeatable way or state."* However, the value of documentation provided solely by code can vary depending on the developer's context. Developers less familiar with a particular flavour of IaC may have different interpretations of what that particular component is doing, especially if they cannot understand the syntax of the language in the first place.

Offloading an NFR also allows an organization to effectively ratchet [16], *"we just put money in the machine and made it better" (Manager at Gamma).* Ratcheting allows an organization to consume more cloud provider resources, or access locked features to handle a particular NFR bottleneck.

6.3.3 Write Your Own Tool

Writing your own tool implies that the organization themselves are writing custom, in-house source code, tooling, scripts, or manifests. The investment required for a custom tool could be more than a resource-constrained organization could absorb, especially if an off-the-shelf or open-source solution exists. However, off-the-shelf applications may not satisfy an organization's needs and are worthy of some heavy modifications, as one developer pointed out: *"so we used to have [name redacted] dashboards out there ... [but that] didn't give us any application-specific information" (Developer).*

A custom tool could be developed, or an existing tool could be extended, to enforce a particular NFR, especially if embedded as part of the CSE pipeline. Beta and Gamma each codified some usability parameters as part of their source code. They submit the source code, wait for the source to process through the CSE pipeline, and finally observe and verify the result: *"if you want to change our usability parameters or whatever we change it in the source code and then we can test it and verify that it still meets our needs" (Developer at Beta).* At Delta, developers define and share development standards within packages and custom tools to define how some NFRs are managed: *"I introduced a number of standards and built some packages to try and enforce those standards. So there's the standard for how we, on the server side, get configuration from the environment" (Manager at Delta).*

Writing your own tool is related to another practice, *Put the NFR in Source Control*, discussed below, as very often the entities that represent the tool are maintained in source control. Having the tool in source control helps spread the shared understanding of the NFR that the particular tool is representing. For example: *"so there's a lambda [function] that runs when you make a bucket, it triggers and goes 'you didn't encrypt', it turns on encryption, tells you, 'you are an idiot'. Security non-functional requirement!" (Manager at Alpha).* In addition, writing your own tool may be related to Put a Number on the NFR, as many times the result of the tool is some form of metric that can be measured and monitored. An organization may codify an NFR as part of their source code, submit the code, wait for the source to proceed through the CSE pipeline, then observe, or monitor the result, and finally choose to react to the metric in some form.

6.3.4 Put the NFR in Source Control

Codifying NFRs that can be stored in source control in some form or another is another popular practice. Codification refers to using code and related artifacts, such as version-controlled IaC configuration files, to capture NFR knowledge. Configuration for tools (e.g. static analysis, code linters, and formatters) can be stored in source control alongside the source code itself. The potential configuration of an NFR could include rules or triggers that represent thresholds for certain metrics that could potentially cause alarms or failures through a dashboard or the CSE pipeline.

Codification is especially important in sharing NFR knowledge for a small organization (e.g. startup) that is constantly changing (typically growing), both in terms of the number of staff and the rapid pace of change a product may experience. Codification helps set objectives for a developer who might not have had enough time to acquire the tacit knowledge about what constitutes an acceptable definition of an NFR. NFR knowledge may not be easy to communicate via natural language: *"with respect to codifying something vs documenting it: it's not precise enough it's written in English. It's open to interpretation" (Developer at Gamma)* This open interpretation is also relevant to NFRs which should not be defined with relative versus absolute terms, e.g. the response time must be fast versus the response time must sub-100ms. Thus, codifying an NFR in a precise and absolute manner, such as source control, removes some possibility that the NFR may be misinterpreted.

6.3.5 Solicit Feedback

Soliciting feedback describes how an organization continuously validates the shared understanding of an NFR through feedback channels, such as code reviews and deep-dives, as a part of their CSE process, e.g. *"there are things like code review, review and feedback from the managers that sort of catches the NFRs" (Developer at Delta).*

A code review is a common and proactive way to build a shared understanding of an NFR and, also solicit feedback that some given code satisfies the expectations for some NFRs, such as maintainability, testability and scalability. A good code review may reveal when the expectation for the NFR has not been met; for example, testability, when written code cannot be unit tested, or when unit tests do not provide complete code coverage. Continuously soliciting feedback on NFRs allows an organization to share and build knowledge regarding NFRs with the stakeholders and ensures a common understanding of the NFRs relevant to the software product.

In addition, feedback meetings such as deep-dive meetings or shaping meetings create opportunities for the software team to proactively build a shared understanding of NFRs. A software developer can discover and initiate the shared understanding of NFRs, such as extensibility, through other feedback methods such as a project deep-dive or shaping meetings: *"our [deep-dive] during the development process and the project engineering role really helped with that. Because when the shaping is happening, we're starting to think about those non-functional requirements: how it might work for other use cases, how something should be developed from a quality perspective and what tools we are using" (Developer at Delta).*

A shaping meeting is a meeting that occurs at the start of a project where the project stakeholders come together to decide on the initial high-level requirements of the intended product; whereas a deep-dive meeting occurs after the project implementation begins and usually only includes the software developers or managers directly involved in implementing the intended product. During deep-dive meetings, the software team can demonstrate a product or its design and subsequently discuss and verify that the product is aligned with both the low-level requirements and high-level requirements.

6.3.6 Leverage Past Experience

Past experience is one way to build a shared understanding of NFRs. A software developer may deepen their understanding of how NFRs were handled by reviewing previous projects and improving their understanding of those NFRs: *"I think the biggest tool that has helped me learn NFRs has really just been reviewing previous work, specs, and integrations." (Developer at Delta).* Leveraging past experience was found to be primarily reactive, due to triggers such as technical debt or new requirements, which encouraged a developer to increase their knowledge about the affected NFRs.

Past experience also relates to how an organization values and prioritizes NFRs. Occasionally, an NFR may not be explicitly defined or separated from the business logic of a software project: *"there's a process that isn't formally presented, it's picked up over time through training. I think that the majority of NFRs have been picked up that way" (Developer at Delta).* However, an organization may proactively incorporate processes that encourage developers to learn and understand the valuable NFRs for their software projects. For instance, during onboarding, a developer may

learn about the expectations for code quality as code quality affects the usability and maintainability of the software or application.

6.3.7 Develop Wireframes

Developing wireframes is using a visualization tool to create designs, mock-ups, demos, or sketches to describe NFRs and improve the developer's understanding of NFRs. An organization may proactively visualize an NFR, such as usability, maintainability and interoperability, using third-party software solutions such as Figma: *"something like Figma or Figjam to quickly visualize what we're talking about." (Developer at Delta)*. Sharing a wireframe or mock-up that describes an NFR is a valuable way to spread the shared understanding of an NFR and also elicit feedback from stakeholders.

An organization may share mock-ups and minimalistic designs of software with stakeholders during meetings to receive feedback to determine the expectations for usability: *"that's where things like 'shaping', wire-frames, starting to try to think through that user experience really helps. Design drawings, I find are really helpful. Early mock-ups help people because I find that a lot of the team does well with actually experiencing something. They quickly start to see this does the thing, but it's totally unusable for other reasons" (Manager at Delta)*. A wireframe may serve as a common reference for the expectations of an NFR, such as usability, for developers working on a project throughout the software development process. Due to the fast pace of change in an organization that adopts CSE, developing a wireframe and mock-up may help project stakeholders build and maintain a shared understanding of an evolving NFR.

6.3.8 Discuss Problems and Solutions

Another practice is to frequently communicate and discuss an NFR through specialized training, guild meetings, or another regularly scheduled meeting within an organization. An organization can create a skill-focused group, such as a guild, to centralize individuals working on specific aspects of projects and encourage knowledge sharing amongst and within these groups.

A guild may include individuals from diverse technical and non-technical teams within an organization, enabling cross-functional communication and collaboration between project stakeholders. Discussions through a guild allow an organization to

proactively focus on a specific important NFR, e.g. a release management guild may organize exercises or training for communicating configuration and software deployment expectations that may affect the availability or portability of the software: *"it's been helpful for talking about how we build things and non-functional requirements in specific areas of our jobs" (Developer at Delta).*

In addition, discussing a problem or solution through a guild allows an organization to create and share development standards for NFRs and to establish an alignment for some NFRs amongst individuals who independently work on different projects: *"Great user experience is one of those things that's difficult. When it comes to user experience, we've centralized the people who make user experience decisions into a group and then we have standards in that group" (Manager at Delta).* Sharing development standards for an NFR, such as testability, removes ambiguity and ensures that each software team member understands what constitutes good test coverage and ease of unit or regression testing.

6.3.9 Ask Questions

Asking questions is a popular approach for building a shared understanding of NFRs during the early stages of the software development process. A developer may use question-asking to clarify expectations for an NFR and ensure that each stakeholder has a good understanding of a particular NFR. A software developer or manager may ask a client questions to elicit the NFRs that are important during the early stages of a project: *"you want to ask a lot of questions, you want to poke a lot of holes and things and, really go to the very edge of, you know, when you're sketching it out." (Manager at Delta).*

The effectiveness of gaining knowledge about an NFR through asking questions depends on the richness of the medium of communication. When a software developer asks questions about an NFR via text, e.g. when using Slack or other asynchronous communication, the response may not be rich enough to sufficiently satisfy the developer's expectations. Hence, a richer communication medium, e.g. video call or in-person conversation, may provide a more effective way to build a shared understanding of an NFR.

During a meeting or conversation, a software developer and other project stakeholders may develop a common understanding of an NFR even with a minimal set of questions being asked: *"when we go into the deep-dive, we know when the non-*

functional requirements have been met when we as a group meet, and there aren't gaps in our shared understanding. We come out of a review meeting, and there's not a lot of feedback" (Developer at Delta). Frequently asking questions during the software development process enables building a shared understanding of NFRs among project stakeholders.

6.4 Challenges to Building a Shared Understanding

Although the aforementioned practices were found to be useful in developing and maintaining a shared understanding, I also found that each is susceptible to challenges. Through the research on this theory, five challenges were found to be a result of the nine practices and resulted in a lack of shared understanding. I will discuss these challenges next.

6.4.1 Loss of Control

Loss of control of an NFR may be experienced when an NFR is offloaded to a third-party provider, such as AWS or Google Cloud. Third-party providers facilitate an organization of any size in achieving a multitude of NFRs, that would otherwise be too expensive or difficult to achieve in a timely manner. However, while offloading can help in satisfying the NFR, it also comes with a loss of control of an NFR and consequently negatively affects the shared understanding of that NFR within the organization. A certain level of shared understanding must be maintained to keep the NFR satisfied in the future. If something were to happen to an offloaded NFR, then the knowledge of how to rectify the issue must be present.

6.4.2 Deprioritization of NFRs

Deprioritization occurs quite often in CSE, as quick iterations favour functional requirements over NFRs: *"we tend to deprioritize ones that aren't like hard functional problems" (Developer at Beta).*

Agile methodologies have been shown to risk the overemphasis of FRs at the expense of NFRs [49], whereby the fast pace of CSE increases the risk of deprioritizing NFRs [78, 50]. The emphasis on FRs is even more prevalent when an organization is

resource-constrained (e.g., a startup) and removing an unclear, hard-to-define NFR from the sprint or milestone is an easy choice; for example *"the sort of startup code today some of it you could consider clever but you do too many clever things then rack up a lot of technical debt" (Manager at Alpha).* Frequently, an organization may trade-off some amount of technical debt to ensure that a new system-defining feature can be developed and released: *"Some of the core pieces of the system again get more love or more time to knock that [technical debt] number down or we just pay closer attention" (Manager at Alpha)* However, over time this trade-off may produce an insurmountable amount of technical debt and should not be ignored for too long: *"nobody's looked at this last six months maybe someone should check it out [...] It's a resource management issue and a lot of times you have too many things for too few people" (Manager at Alpha).*

6.4.3 Lack of Domain Knowledge

Lack of domain knowledge can cause a lack of shared understanding of NFRs if the domain is new or entirely unknown to an organization. With a new domain, there is some expected essential lack of shared understanding to be present; however, at the very least an organization should have a basic understanding. Often, an organization may be an expert in their respective domain and attempt to apply its technology to a similar domain: *"the scoping and the understanding of how broken down you need to have a ticket with descriptions, to understand the non-functional requirements that need to happen" (Developer at Beta).* However, an organization may struggle in adapting to a new domain, as there may be unintended effects that increase the lack of shared understanding or drastically change the shared understanding, but perhaps not equally.

Some NFRs may be captured by governing laws, often written in specialized legal language that only a lawyer could understand (see Section 6.2.3). Therefore, dissecting and disseminating such information requires careful coordination between staff who can understand the jargon and staff who need to implement the NFRs described in a language they cannot comprehend.

The lack of domain knowledge can also negatively affect the priority that should be placed on developing the shared understanding of an NFR. An organization may exhibit some disparity in the priority placed on NFRs, or even a single NFR, especially between two units within an organization. While a seasoned developer may place

heavy emphasis on particular NFRs, these may be perceived as having low priority to non-technical teams, such as marketing or sales.

An NFR may have such a low priority that it receives no attention at all, until it causes a partial or total loss of functionality, at which point the organization has no choice but to elevate the priority of the NFR and assign some dedicated resources. However, these resources often are season-veterans, who may have experience in dealing with NFRs; thus it is important to ensure all team members maintain a shared understanding of the NFRs.

6.4.4 Inadequate Communication

Lack of communication may inhibit an organization from adequately building a shared understanding, as communication is paramount to the often implicit and cross-cutting nature of NFRs. Unfortunately, the required amount of communication is usually deficient until it is too late: *"two different developers building the front-end and the back-end at different times. Only like a week or two apart, but they weren't communicating well enough to each other and it wasn't enough like top-down planning of that" (Manager at Gamma)* Developers have been found to be simultaneously working on the same problem, independently and in isolation, but not communicating the right information. Inadequate communication causes divergent paths of thought and increases the lack of shared understanding.

Communication is key to ensuring all developers have a shared understanding; however, communication is time-sensitive. Often it takes time for some information to be common knowledge and develop into a shared understanding, as very often *"there was no documentation about it" (Developer at Gamma).* Alternatively, a developer could make false assumptions that everyone else has the *same* understanding of an NFR: *"there have been times where I've had a conversation with [person] about what a product needs to do and how he has envisioned it. And I guess some things just maybe got lost in translation. And I end up working on something that is different or works differently than what he had imagined. And vice versa" (Developer at Delta).* In addition, the need to communicate the NFR may be overlooked by a single person: *"the root of it is we built one half, the front-end, without building the back-end" (Developer at Beta).* The definition of quality and releasable could vary quite drastically between just two developers. Ultimately, as an organization matures the organization and its people will develop procedures and methods to foster *"better coordination and*

communication" (Developer at Gamma) that will create an environment conducive to developing a shared understanding.

6.4.5 NFRs are Hard to Automate

Some NFRs are notoriously hard to automate, which is in direct conflict with CSE [135], which encourages practices such as *automate the build* and *make the build self-testing*. If an organization is committed to adopting CSE and commits and deploys code at a rapid pace then focus must be spent on how to automate the validation and verification of NFRs, as they may become a bottleneck to releasing software. Often it is easier to write test cases that test the positive cases and with NFRs it may be harder to predict what could go wrong and ultimately write an automated test for that prediction. For example: *"I feel like [tests] always [had] a bias toward the happy cases [...] When something does go wrong, it's something horribly out of left field [...] How do you prepare for those? How do you think about what left field is?" (Developer at Gamma)*. However, like any task in software development, it need not be perfect, as in CSE change is a certainty; writing a test case that meets the bare minimum definition of done is sufficient—assuming subsequent rounds or iterations still consider building upon the initial effort.

6.5 A Shared Understanding of NFRs

The purpose of this theory is to educate and inform a software organization on the intricate cycle of how to recognize the need to build and maintain a shared understanding, some of whom have been educated through this study: *"I do want to complain about our [tasks] though as the discussion revolving around the task actually created a shared understanding and let the developer to actually add a little description" (Developer at Alpha)*. However, at times the line between shared understanding and a lack of shared understanding may be fluid and ambiguous. Maintaining a shared understanding is as arduous, perhaps even more than creating the shared understanding, especially as an organization grows. Many contextual factors affect the ability of an organization to build a shared understanding, including the complexity of the software, the environment that the software ultimately resides in, and the size of the organization to name a few. Once a shared understanding has been initially con-

structed, the onus lies within the organization to maintain that shared understanding to avoid having the shared understanding melt into a lack of shared understanding.

Chapter 7

Discussion & Conclusion

Throughout a multi-year multi-case study my empirical research sought to uncover and develop a deep and rich understanding of the state of practice of a shared understanding of NFRs in CSE. In particular, using four local organizations as sites, I was able to provide considerable insight into RQ1, concerning the interplay between a shared understanding of NFRs and CSE, and RQ, describing how an organization builds, manages, and maintains a shared understanding of NFRs in CSE. Ultimately, the insights provided by answering RQ1 and RQ2 were used to develop the Iceberg Theory. The research performed in this dissertation has several implications to discuss with respect to the shared understanding of NFRs that will be discussed next. I first discuss the intricate relationship between shared understanding and CSE, followed by the relationship between NFRs and CSE, the importance of configurability as an NFR, practical details on how an organization can build a shared understanding in CSE, and concluding remarks of the importance and implications of the Iceberg Theory.

7.1 Shared Understanding and CSE

At first sight, CSE advocates for ongoing customer feedback and rapid iteration cycles [64] that should result in a heightened understanding of customer requirements. In this deeper empirical investigation, this study reveals the potential for a much more complex relationship between the lack of shared understanding and practices of CSE. The findings from this dissertation suggest that *CSE may hurt the* shared understanding of NFRs and lead to rework.

One expected benefit of CSE is an improved validation of requirements, due to rapid customer feedback [136]. A quick feedback loop is associated with lowering the impact of lack of shared understanding [1]. However, as NFRs are cross-cutting and difficult to decompose into sub-components that can be completed within a short iteration [16], an NFR may take days, weeks, or even months to evaluate. For example, at Alpha, there was rework due to a lack of shared understanding of privacy that was untestable, *"we never really know what we're testing because it requires a host of real users hitting it to get an accurate picture of your hits over the last couple of weeks, developers won't have weeks of data."* Beta had a similar experience where *"a new FR came in to integrate a third-party service, which we dutifully did, but it cratered system performance. Embarrassingly, there were so many other changes going on at the same time (some our own and some seasonal) that it took us a few weeks to discover and then track down the problem."* For these four organizations, it is hypothesized that CSE hurts the shared understanding of NFRs, much like how agile RE practices increase the risk of overemphasis of FRs [49]. A notable complication is CSE has a broader scope than agile, as not only do you need to slice NFRs [137], but you need to subsequently ship them in rapid, frequent deployments [78].

NFRs were found to be de-prioritized in CSE, in part due to the frequent release of FRs being a top priority and a lack of shared understanding, around either domain or technical knowledge. For example, Alpha, Beta, and Gamma each described how role siloing between DevOps and developers creates a lack of shared understanding of NFRs. NFRs may not get prioritized until the right (profitable) customer complains, e.g. *"flagship user of this product, deemed very important" (Manager at Beta).* At these four organizations, product managers are often tasked with creating, prioritizing, and validating tasks. Tasks were typically only created for FRs, and the NFRs are usually implicit, and in some cases as what Glinz and Fricker call *dark information*, i.e., relevant but unnoticed knowledge [1]. Another claimed benefit of CSE is the removal of intra-organizational barriers, such as between product managers, DevOps, and developers [64]. These barriers are prohibitive to both adequate communication and acquiring domain knowledge.

By tearing down these barriers, CSE should help alleviate the difficulty associated with cross-cutting NFRs that require input from multiple individuals and roles across an organization. However, a lack of shared understanding existed across organizational departments and roles and NFRs remain difficult for non-developers to grasp. Product managers in particular ceded control of NFRs to developers, relying on in-

tuition to verify the NFR was achieved: *"we're going to see this thing before it goes out the door and we'll know something feels wrong in terms of an NFR" (Manager at Beta)*. In particular, performance and scalability cannot be validated by a single product manager *"knowing when something feels wrong"*, when they actually require thousands of customers to verify.

The findings presented also suggest a disconnect between development and product management at these organizations practicing CSE, where a product owner would typically bridge the knowledge gap. Unlike agile development, CSE does not define a product owner role, who would have substantial technical *and* customer domain knowledge. At these four organizations, the product manager role was responsible for writing the tasks and deciding each task's priority. However, due to this disconnect NFRs requiring substantial technical *and* customer domain knowledge, such as security and privacy [138], were often neglected or underspecified [139] leading to a lack of shared understanding. At these organizations, the primary difference between an agile product owner and these four organizations' product managers is the insufficient level of technical knowledge; therefore, CSE may be left without the benefit of a true product owner.

On the positive side, CSE relies heavily on writing and maintaining numerous test cases, a form of shared understanding, that is part of the automated CSE build [1]. However, these organizations did not heavily invest in substantial testing efforts, even for FRs. Furthermore, NFRs are difficult to test [54, 55] and may require specific architectural considerations to be automatically tested [140], therefore NFRs are largely left as a manual task in CSE [10]. The lack of NFR tests could be due to a lack of shared understanding, i.e. they don't know which ones to test; alternatively, it could contribute to a lack of shared understanding of NFRs, i.e. tests themselves represent shared understanding anyone can read, modify, or execute. For example, interoperability may be degraded due to inadequate tests *"the root of it is we built one half, the front-end, without building the other, the back-end" (Developer at Beta)*. Even if an NFR is not fully tested as part of the CSE pipeline, the organizations are not realizing the full benefits of the CSE pipeline due to a lack of tests. One key to building a shared understanding of NFRs through tests in CSE is to, at the very least, have imperfect tests [135]; these imperfect tests are the first step in acknowledging and bringing awareness to the value of the NFR and start fostering a shared understanding.

7.2 NFRs in CSE: A Silver Lining

NFRs often do not get the appropriate attention they deserve. NFRs are cross-cutting in nature as they impact many aspects of the system and may be difficult to decompose into fragments that can be realized in a short, rapid CSE iteration [16], which further complicates the ability of an organization to manage an NFR. However, the evidence suggests that it might be easier, for organizations that shift to CSE, to manage through one of the nine practices identified.

Typically managing an NFR encompasses a number of steps, including elicitation, analysis, negotiation, implementation, verification, and validation. However, these practices at these four organizations suggest that an NFR may be realized *without direct* implementation, i.e. actions have been taken to satisfy that the conditions of the NFR have been met, although it may not necessarily be implemented or verified by the organization. Hence the term realization is used in the broader sense, as opposed to the traditional implementation. The realization of an NFR is composed of a number of sub-tasks and in this thesis, realization is used to indicate when an organization has reached a satisfactory level of an NFR to have realized it, whether or not it is completely satisficed or not. However, during the process of realizing an NFR, it is vital to *put a number on the NFR*, as a realized NFR may be affected, perhaps negatively, unbeknownst to developers, e.g. the implementation of a new feature causes performance to crater.

While the study discussed in this dissertation did not directly observe the elicitation of NFRs at the four organizations, the practices identified were concrete actions these organizations took to support NFR realization and verification. While the recent comprehensive SLR [63] confirmed the ability to *verify* NFRs by leveraging CI, this is just one aspect of NFR management. NFR verification may confirm *if* an NFR has been realized; however, realizing an NFR doesn't necessarily mean the NFR is verified.

For example, an organization may realize, potentially only a part of, resiliency by offloading it to Amazon Web Services and potentially verify resiliency with some form of chaos engineering [141]; however, note that realization and verification do not necessarily go hand-in-hand. As part of NFR management, an organization was able to realize an NFR, for example, availability, resiliency, or scalability by offloading to a third-party, ultimately resulting in very little overhead, aside from the associated cost to the organization.

While the SLR by Yu et al. [63] is the closest work to ours, they found leveraging CI is underutilized to verify NFRs and that the ratio of industrial to theoretical studies is low—thus highlighting the importance of this study bringing substantial empirical evidence to support that CSE is an enabler in, not only testing but, realizing NFRs.

Furthermore, through this study of the practices and challenges at these organizations, 30 NFRs were uncovered that they found relevant, which are clearly not an exhaustive list for *all* organizations. Seven NFRs are in common with the findings from Yu et al. [63], with latency and productivity being the exceptions. Notably absent from their list are 3 of the top-5 NFRs from this study, namely configurability, security, and usability.

Although Yu et al. specifically mention usability as hard to verify, this study provides evidence to suggest that some organizations are satisfied with their level of realizing usability; of course, this distinction is relative and may not apply to other organizations in such a black and white manner. In particular, it was previously discussed how Beta was able to leverage CSE to realize usability in real-time (see Section 5.2.1). In addition, Gamma is able to track usability metrics through their CSE practice, including user events, such as button clicks, page views, and navigation traces, and runs large-scale A/B experiments [142]. While, this distinction certainly merits further investigation to *exactly* how this organization satisfactorily realizes usability and how this realization can be applied to other domains and organizations, the exact details are outside the scope of this study. By leveraging metrics, the feedback loop, and continuous monitoring, Beta and Gamma have a near-constant realization of usability—an otherwise difficult-to-realize NFR.

Metrics, of any kind, are the starting point that allows an organization to set goals, track progress, and monitor the state of the system in a reliable fashion [143]. However, metrics are not without problems, as assigning a desired threshold to an NFR is not trivial. Fixed or static thresholds may be problematic for complex NFRs, requiring alternative solutions such as desired, minimum, dynamic thresholds [144], or even the use of artificial intelligence to adapt the thresholds.

Therefore it is believed that continuous, rapid iterations using metrics, the feedback loop, and continuous monitoring bring an increase to transparency and traceability of NFRs. Transparency and traceability are afforded by allowing anyone in the organization to easily track changes to a particular NFR metric, back to a localized source commit in the code [145]. Traceability in CSE has been previously studied in the Eiffel approach [74]; however, the Eiffel approach is aimed at improving the CSE

pipeline, not necessarily the resulting software. Stahl et al. [74] note that further work includes extending Eiffel to consider development activity, which would ideally include NFR activities as well.

The ability to realize and continuously monitor, track, and audit NFRs in real-time throughout the entire life cycle of a software project is immensely powerful [74]. Alternatively, an organization may hire consultants to assess the satisfaction of a particular NFR, such as security; however, this is often a one-time assessment and does not help with *continuous*, ongoing satisfaction of the NFR in question, which is usually the key from an operational point of view.

While assigning metrics to NFRs is not a new idea and has always been a recommended practice to ensure proper verification of NFRs [132], consideration of the metric often only happens during the initial design and architecture phases. Once the NFR has been defined, measured, and perhaps satisfied, and the organization is deeply entrenched in actual coding, the NFR may no longer be tracked [10]. The key novelty with CSE is that it facilitates realizing NFRs and the constant and continuous ability to monitor and satisfy NFRs through the quick feedback loop. An organization is able to look at its CSE pipeline and determine the gap between NFR objectives, and actual level of performance, usability, or configurability, among others.

7.3 The Importance of Configurability as an NFR

The importance of configurability as an NFR grows as an organization relies more heavily on CSE practices. Configurability is an attribute of the software system that refers to how easily an organization can configure its software infrastructure and environments [132], including IaC. Configurability is itself a prime example of how an organization can build a shared understanding of NFRs, by putting IaC in source code. However, configurability is *more* than just a system quality, as it also encompasses overarching *process quality*. The data shows the importance of the non-functional quality of the system's configurability—the source code, build scripts, infrastructure and deployment configuration, and associated hardware. Like maintainability, configurability refers to an internal quality that supports the goal of rapid deployment and re-configuration. To realize this NFR, one might use tactics such as rollbacks, keeping production and development environments in sync, or applying infrastructure as code tools such as Puppet.

Comprehensive configurability has long been considered to be an enabler of the many perceived benefits of CSE by Humble et al. [133]. However, configurability is largely underrepresented. The concept of configurability has further grown to encompass the configuration of build, staging, and deployment infrastructure, ideally with little to no human intervention [134], other than to develop a shared understanding of how to configure and manage this infrastructure. The research in this dissertation brings clear evidence that configurability should be considered an extremely important and high-priority NFR in CSE that deserves focus on building a shared understanding.

As software systems increasingly exist as a service running in the cloud, application code is no longer the only important source code. Infrastructure configuration and code are as vital to software business goals as application code.

At Alpha, customers are in part paying for Alpha to host reports and data for them. Thus, their infrastructure configuration and code must also exhibit NFRs such as reliability and availability. In contrast, these NFRs are entirely the customer's responsibility for an on-premise offering. An organization must now invest in building a shared understanding of these NFRs, including their configurability, in parallel with other NFRs and features.

As the technology director at Gamma commented during the member checking phase, *"[configurability], and associated IaC and automation is the enabler that allows organizations like ours to essentially offload other NFRs such as availability, scalability, and security to cloud providers [...] without [configurability] other NFRs would suffer, such as reliability, maintainability, repeatability, and even availability due to more human error during deployments."*

The increased reliance on configurability results in a trade-off: the organization needs to now spend significant additional resources on configuration, developer training, and avoiding potential vulnerabilities associated with configurability, including the lack of shared understanding.

First, configurability now has its own set of distinct code smells [146]. There are early efforts to mitigate these code smells, such as Rahman et al. [147] to identify code smells of configurability code in open source software. Second, while configurability may benefit from standard coding practices it is not yet done in practice [117]. Conversely, configurability is actually associated with a wide variety of disparate languages and tools. Most organizations use three or more different tools and no single tool is used by the majority of organizations [117]. Existing standards,

such as Topology and Orchestration Specification for Cloud Applications (TOSCA) and Open Cloud Computing Interface have been proposed; although the adoption amongst DevOps engineers is low (18%) [117]. Third, the standards themselves do not contain a complete set of NFRs, as they require extensions to include security [148, 149] and privacy [150], among others.

7.4 How an Organization Can Build a Shared Understanding in CSE

In addition to researchers, this study has wide-reaching implications for practitioners. The observed practices and challenges for handling NFRs in CSE demonstrate that organizations are both successfully treating NFRs and encountering difficulties. For practitioners, there are three noteworthy implications. First, organizations must be aware of the ability to realize NFRs using the nine practices that leverage CSE, but also be mindful of the associated challenges. Second, while offloading NFRs to a third-party provider has many potential benefits, practitioners should be aware of the potential consequences of offloading. Furthermore, practitioners should monitor any offloaded NFR to ensure the NFR is treated as expected. Fourth, practitioners need to recognize the importance of configurability and dedicate time to educate, elicit, analyze, and verify configurability. Finally, and most importantly, organizations need to learn exactly how to build a shared understanding of CSE. Upon encountering a trigger, e.g. regulatory requirements, rectifying technical debt, important customer complaints, and disruptions of service, for building a shared understanding, the next step is actually initializing a response. However, these four organizations are largely reactive to rework due to an accidental lack of shared understanding, e.g., responding to a disruption or customer complaint, as opposed to being proactive in building a shared understanding. RE research has shown that a certain amount of proactive RE is valuable [41]. Furthermore, the participants in this study suggested two practices that could help them build a shared understanding: shared development standards and documentation and communication. Practical implications are discussed as suggested by the empirical findings and which would enable an organization to achieve a more proactive RE approach to their software projects.

While shared standards for documenting NFRs across an organization are necessary to build a shared understanding [151], this documentation does not need to

be exhaustive or complete. Much like agile encourages *working software over comprehensive documentation*, there is a minimum level of NFR documentation required to achieve a shared understanding in CSE. An organization must strike a balance to meet this minimum, and sufficient, level of NFR documentation [152]. While determining what is considered a sufficient level of documentation might be different at each organization [153], a minimum level could be as simple as writing the NFR as part of the acceptance criteria in a task to a more involved characterization [15].

An organization should first select a single, vitally important NFR, such as performance or reliability based on their context. In the most simplistic manner possible, the NFR must be documented to ensure cross-functional visibility. All four organizations maintain some form of a dashboard that helps increase cross-functional visibility, thus these dashboards can act as a form of documentation and be beneficial in increasing the cross-functional visibility of the NFR. However, documentation on its own is not sufficient for building a shared understanding.

Communication is a well-known and studied knowledge management practice [154, 155], thus the next logical step is to share this minimal NFR documentation across roles and departments in a CSE practising organization, which may require cognitive understanding [156]. Establishing a cognitive shared understanding involves identifying insufficiency in an organization's understanding and rectifying misunderstandings [156]. The method of communication to establish this cognitive understanding could materialize in a variety of forms, including face-to-face conversations, emails, or corporate branding.

Once a standard level of NFR documentation has been developed and disseminated across all roles and organizational departments, then the value and awareness of the NFR must be incentivized to achieve buy-in from all roles and organizational departments. This may be achieved through executive sponsorship through a top-down approach. In contrast, documentation can also be achieved from the grassroots level, where developers initialize and progress towards disseminating knowledge between roles in the interest of self-help.

Finally, the NFR itself should be verified as part of the CSE pipeline. A well-known solution [157, 16] is to ensure NFRs can be quantified, for example as response measures or acceptance criteria. As part of their efforts for auditability, the three collaborating organizations actively add metrics to catch errors and increase insight into their system. Some NFRs are more difficult to verify than others; although even privacy has been managed through custom tools that automatically test

for software deficiencies [102]. Once an NFR can be verified, the results should be publicly displayed throughout the organization, to continue and maintain a high level of awareness. Furthermore, if a particular NFR is not satisfied an organization can adopt a similar practice from lean software development whereby all employees must halt production to fix an issue [158], which will help excite shared understanding across roles and departments. This ensures that shared understanding is consistent and maintained across the organization, including new employees. Ultimately, building awareness and educating the organization regarding the value behind building a shared understanding of NFRs, including priority, is paramount to software success.

The key constructs of the Iceberg Theory are shared understanding, or lack thereof, triggers, practices, and the challenges linked to those practices. In essence, my process theory describes what happens when organizations seek to address a lack of shared understanding of the NFRs that are important to them. As a process theory grounded in extensive empirical data, I posit that it offers a useful foundation for future studies. All theories have limitations and boundaries beyond which their validity could be challenged. The Iceberg Theory presented here defines a number of *triggers*, *practices*, and *challenges*. While I identified several of each through the empirical work, it is clear that these are not exhaustive, and that other elements could be revealed in further studies.

Quite often NFRs are not given the attention they deserve. NFRs affect every aspect of a software system, while some NFRs may be more superficial, others are pertinent to the success of the software. A large all-encompassing NFR may be difficult to slice into pieces that can be implemented in a relatively short, rapid CSE cycle [16], thus further complicating actually building a shared understanding of said NFR. However, the practices, and the associated challenges, discussed in this theory leverage common CSE concepts to help build a shared understanding, thus highlighting the importance of this theory, and the underlying empirical evidence, to support that CSE is an enabler in managing the shared understanding of NFRs.

7.5 Final Conclusion

Despite the well-understood importance of NFRs to a software project, exactly how an organization builds a shared understanding of NFRs in CSE has not been well-understood. The motivation behind the Iceberg Theory presented in this dissertation is to collect, extend, and build upon the current state-of-the-art knowledge of a shared

understanding of NFRs in CSE. The Iceberg Theory is based on extensive empirical evidence gathered from four case studies and is presented in such a manner as to have real implications for both researchers and practitioners. The theory provides researchers with the building block for future studies on the shared understanding of NFRs and offers practitioners a way to recognize the importance and build a shared understanding of NFRs.

The theory's entities are the shared understanding, the lack of shared understanding, triggers, practices, and challenges, and their respective relationships. The lack of shared understanding could manifest as any number of triggers, which is the most likely starting point for an organization. These triggers typically prompt an organization to invest in one or more of the nine practices that could help build a shared understanding in response to these; however, the practices are not a silver bullet and are susceptible to challenges that could result in a further lack of shared understanding. The theory acknowledges that some lack of shared understanding is the unknown unknowns [47] and may be essential to the software development lifecycle; while other forms of a lack of shared understanding may be accidental, and could have been prevented. Finally, there is a fluidity between the lack of shared understanding and a shared understanding, as an organization strives to reduce the lack of shared understanding of a particular NFR into a true shared understanding.

A number of open-ended questions and unexplored avenues merit future attention. First, further research is needed to develop and evaluate strategies to overcome the associated challenges, while some observations and ideas have been presented as part of this theory, they remain out of the scope of the theory's primary goal. Does the associated challenge exacerbate the problem? How can a particular challenge be mitigated? Are there modifications to a practice that should lessen the impact that a challenge has on a shared understanding? Second, while this theory helps bring awareness to organizations, some were surprised by their own lack of shared understanding, attention is required to research and develop a method, practice, or tool to assist an organization to identify the potential lack of shared understanding in CSE is needed. Triggers aside, how does an organization know when they have a lack of shared understanding? Is there a way to categorize the lack of shared understanding as accidental or essential? Finally, one reason an organization may be more reactive, as opposed to proactive, in building a shared understanding is due to the fast-paced environment associated with CSE, more empirical evidence and examples of triggers, both reactive and proactive, and methods and tools to help

identify triggers as opportunities to build shared understanding. How effective are triggers in building a shared understanding? Can an organization proactively build a shared understanding as opposed to reacting to a trigger? Is the resulting shared understanding as a result of reacting to a trigger less efficacious than proactively building?

The Iceberg Theory highlights the importance of NFRs and provides a blueprint to guide an organization to build a shared understanding of NFRs. Not only does it highlight important NFRs, but it also calls attention to the difficult problem of building a shared understanding of NFRs in CSE. The entities of the theory are presented in an easy-to-understand graphical representation of an iceberg (see Figure 6.1), as discussed in Section 6, allowing an organization to comprehend the theory from a practical point of view. In addition, all of the entities are described using terminology that should be familiar to an organization, as they were all derived from practice.

For example, if an organization can identify some lack of shared understanding through technical debt, the theory can help direct them toward a particular practice providing a suitable response, in addition to potential pitfalls. Future extensions to the theory could help study exactly which practice is best suited for any particular trigger. Ultimately, I believe the Iceberg Theory can help raise awareness and educate an organization on how to identify the lack of shared understanding through several triggers and reveal the value behind building a shared understanding of NFRs, including priority, which is paramount to promoting successful software projects.

Bibliography

[1] M. Glinz and S. A. Fricker, "On shared understanding in software engineering: An essay," *Comput. Sci.*, vol. 30, no. 3-4, pp. 363–376, Aug. 2015.

[2] E. W. Dijkstra, "The humble programmer," *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, 1972.

[3] M. Jackson, "Problem frames and software engineering," *Information and Software Technology*, vol. 47, no. 14, pp. 903–912, 2005.

[4] (2015) ISO/IEC TR 19759:2015 Software Engineering: Guide to the software engineering body of knowledge (SWEBOK). Accessed: February 11, 2021. [Online]. Available: https://www.iso.org/standard/67604.html

[5] S. Wagner, "A literature survey of the quality economics of defect-detection techniques," in *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ser. ISESE '06. Rio de Janeiro, Brazil: Association for Computing Machinery, Sep. 2006, pp. 194–203.

[6] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Foundations of empirical software engineering: the legacy of Victor R. Basili*, vol. 426, no. 37, pp. 426–431, 2005.

[7] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques*, 1st ed. Wiley Publishing, 1998.

[8] I. B. Jacobson and G. R. Booch, "The unified software development process," 1999.

[9] M. Glinz, "On Non-Functional Requirements," in *15th IEEE International Requirements Engineering Conference (RE 2007)*, Oct. 2007, pp. 21–26.

[10] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "How do software architects specify and validate quality requirements?" in *European Conference on Software Architecture*. Springer, 2014, pp. 374–389.

[11] E. Targett. (2018) Amazon outage: Estimated $99 million lost. Accessed: August 23, 2019. [Online]. Available: https://web.archive.org/web/20190823180248/https://www.cbronline.com/news/amazon-outage-lost-sales

[12] W. Alsaqaf, M. Daneva, and R. Wieringa, "Quality requirements challenges in the context of large-scale distributed agile: An empirical study," *Information and Software Technology*, vol. 110, pp. 39–55, Jun. 2019.

[13] J. Eckhardt, A. Vogelsang, and D. M. Fernández, "Are 'non-functional' requirements really non-functional?: An investigation of non-functional requirements in practice," in *International Conference on Software Engineering*. New York, NY, USA: ACM, 2016, pp. 832–842.

[14] B. Nuseibeh and S. Easterbrook, "Requirements engineering: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*, ser. ICSE '00. New York, NY, USA: ACM, 2000, pp. 35–46.

[15] L. Chen, M. Ali Babar, and B. Nuseibeh, "Characterizing Architecturally Significant Requirements," *IEEE Software*, vol. 30, no. 2, pp. 38–45, Mar. 2013.

[16] S. Bellomo, N. Ernst, R. L. Nord, and I. Ozkaya, "Evolutionary improvements of cross-cutting concerns: Performance in practice," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 545–548.

[17] G. Booch, "Object oriented design with applications," Tech. Rep., 1994.

[18] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, 1999.

[19] E. A. C. Bittner and J. M. Leimeister, "Why shared understanding matters–engineering a collaboration process for shared understanding to improve collaboration effectiveness in heterogeneous teams," in *2013 46th Hawaii International Conference on System Sciences*, Jan. 2013, pp. 106–114, iSSN: 1530-1605.

[20] P. Darch, A. Carusi, and M. Jirotka, "Shared understanding of end-users' requirements in e-Science projects," in *2009 5th IEEE International Conference on E-Science Workshops*, Dec. 2009, pp. 125–128, iSSN: null.

[21] D. I. K. Sjøberg, T. Dybå, B. C. D. Anda, and J. E. Hannay, *Building Theories in Software Engineering*. London: Springer London, 2008, pp. 312–336.

[22] P. Johnson, M. Ekstedt, and I. Jacobson, "Where's the theory for software engineering?" *IEEE Software*, vol. 29, no. 5, pp. 96–96, 2012.

[23] J. Herbsleb, A. Mockus, and J. Roberts, "Collaboration in software engineering projects: A theory of coordination. international conference on information systems," in *International Conference on Information Systems*, 2006.

[24] R. K. Yin, *Case Study Research: Design and Methods*, 3rd ed. Thousand Oaks, Calif: SAGE Publications, Inc, Dec. 2002.

[25] C. B. Seaman, "Qualitative methods in empirical studies of software engineering," *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[26] C. Potts, "Software-engineering research revisited," *IEEE Software*, vol. 10, no. 5, pp. 19–28, Sep. 1993.

[27] D. S. Cruzes and T. Dyba, "Recommended steps for thematic synthesis in software engineering," in *2011 International Symposium on Empirical Software Engineering and Measurement*, Sep. 2011, pp. 275–284, iSSN: 1938-6451.

[28] P. Ralph, "Toward methodological guidelines for process theories and taxonomies in software engineering," *IEEE Transactions on Software Engineering*, vol. 45, no. 7, pp. 712–735, 2019.

[29] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting Empirical Methods for Software Engineering Research," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 285–311.

[30] E. B. Swanson, "The dimensions of maintenance," in *Proceedings of the 2nd International Conference on Software Engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 492–497.

[31] V. Basili, S. Condon, K. El Emam, R. Hendrick, and W. Melo, "Characterizing and modeling the cost of rework in a library of reusable software components," in *Proceedings of the (19th) International Conference on Software Engineering*, May 1997, pp. 282–291, iSSN: 0270-5257.

[32] R. Fairley and M. Willshire, "Iterative rework: the good, the bad, and the ugly," *Computer*, vol. 38, no. 9, pp. 34–41, Sep. 2005.

[33] E. Ries, *The lean startup: how today's entrepreneurs use continuous innovation to create radically successful businesses*, 1st ed. New York: Crown Business, 2011.

[34] A. Martini and J. Bosch, "The danger of architectural technical debt: Contagious debt and vicious circles," in *2015 12th Working IEEE/IFIP Conference on Software Architecture*, May 2015, pp. 1–10, iSSN: null.

[35] W. Behutiye, P. Karhapää, L. López, X. Burgués, S. Martínez-Fernández, A. M. Vollmer, P. Rodríguez, X. Franch, and M. Oivo, "Management of quality requirements in agile and rapid software development: A systematic mapping study," *Information and Software Technology*, p. 106225, Nov. 2019.

[36] A. Hoffmann, E. A. C. Bittner, and J. M. Leimeister, "The emergence of mutual and shared understanding in the system development process," in *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer, 2013, pp. 174–189.

[37] M. Corvera Charaf, C. Rosenkranz, and R. Holten, "The emergence of shared understanding: applying functional pragmatics to study the requirements development process," *Information Systems Journal*, vol. 23, no. 2, pp. 115–135, 2013.

[38] S. A. Fricker, R. Grau, and A. Zwingli, "Requirements engineering: best practice," in *Requirements Engineering for Digital Health*. Springer, 2015, pp. 25–46.

[39] E.-M. Schön, J. Thomaschewski, and M. J. Escalona, "Agile Requirements Engineering: A systematic literature review," *Computer Standards & Interfaces*, vol. 49, pp. 79–91, Jan. 2017.

[40] E. Bjarnason, K. Wnuk, and B. Regnell, "A case study on benefits and side-effects of agile practices in large-scale requirements engineering," in *Proceedings of the 1st Workshop on Agile Requirements Engineering.* ACM, 2011, p. 3.

[41] D. Damian and J. Chisan, "An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 433–453, July 2006.

[42] "World quality report." [Online]. Available: https://www.capgemini.com/ca-en/news/world-quality-report/

[43] J. Aranda, S. Easterbrook, and G. Wilson, "Requirements in the wild: How small companies do it," in *International Conference on Requirements Engineering.* IEEE, 2007, pp. 39–48.

[44] L. Cao and B. Ramesh, "Agile requirements engineering practices: An empirical study," *IEEE Software*, vol. 25, no. 1, pp. 60–67, Jan. 2008.

[45] N. A. Ernst and G. C. Murphy, "Case studies in just-in-time requirements analysis," in *2012 Second IEEE International Workshop on Empirical Requirements Engineering (EmpiRE)*, Sep. 2012, pp. 25–32, iSSN: 2329-6356.

[46] *No Silver Bullet—Essence and Accident in Software Engineering*, 1986.

[47] A. Sutcliffe and P. Sawyer, "Requirements elicitation: Towards the unknown unknowns," in *2013 21st IEEE International Requirements Engineering Conference (RE)*, July 2013, pp. 92–104.

[48] K. Wiegers and J. Beatty, *Software requirements.* Pearson Education, 2013.

[49] B. Ramesh, L. Cao, and R. Baskerville, "Agile requirements engineering practices and challenges: an empirical study," *Information Systems Journal*, vol. 20, no. 5, pp. 449–480, 2010.

[50] C. Gralha, D. Damian, A. I. T. Wasserman, M. Goulão, and J. Araújo, "The evolution of requirements practices in software startups," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 823–833.

[51] L. Chen, M. A. Babar, and B. Nuseibeh, “Characterizing architecturally significant requirements,” *IEEE Software*, vol. 30, no. 2, pp. 38–45, 2012.

[52] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-functional requirements in software engineering.* Springer Science & Business Media, 2012, vol. 5.

[53] D. Ameller, C. Ayala, J. Cabot, and X. Franch, “How do software architects consider non-functional requirements: An exploratory study,” in *2012 20th IEEE International Requirements Engineering Conference (RE)*, Sep. 2012, pp. 41–50, iSSN: 1090-750X.

[54] A. Borg, A. Y. H. Yong, P. Carlshamre, and K. Sandahl, “The bad conscience of requirements engineering: an investigation in real-world treatment of non-functional requirements,” in *Third Conference on Software Engineering Research and Practice in Sweden (SERPS’03), Lund*, 2003.

[55] Y. Jiang and B. Adams, “Co-evolution of infrastructure and source code-an empirical study,” in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 45–55, iSSN: 2160-1860.

[56] R. Berntsson Svensson, T. Gorschek, and B. Regnell, “Quality requirements in practice: An interview study in requirements engineering for embedded systems,” in *Requirements Engineering: Foundation for Software Quality*, ser. Lecture Notes in Computer Science, M. Glinz and P. Heymans, Eds. Berlin, Heidelberg: Springer, 2009, pp. 218–232.

[57] W. Alsaqaf, M. Daneva, and R. Wieringa, “Understanding challenging situations in agile quality requirements engineering and their solution strategies: Insights from a case study,” in *2018 IEEE 26th International Requirements Engineering Conference (RE)*, Aug. 2018, pp. 274–285, iSSN: 1090-705X.

[58] W. Behutiye, P. Karhapää, D. Costal, M. Oivo, and X. Franch, “Non-functional requirements documentation in agile software development: Challenges and solution proposal,” in *Product-Focused Software Process Improvement*, ser. Lecture Notes in Computer Science, M. Felderer, D. Méndez Fernández, B. Turhan, M. Kalinowski, F. Sarro, and D. Winkler, Eds. Cham: Springer International Publishing, 2017, pp. 515–522.

[59] B. Ramesh, L. Cao, and R. Baskerville, "Agile requirements engineering practices and challenges: an empirical study," *Information Systems Journal*, vol. 20, no. 5, pp. 449–480, 2010.

[60] I. Inayat, S. S. Salim, S. Marczak, M. Daneva, and S. Shamshirband, "A systematic literature review on agile requirements engineering practices and challenges," *Computers in Human Behavior*, vol. 51, pp. 915–929, Oct. 2015.

[61] D. M. Fernandez, "Supporting requirements-engineering research that industry needs: The NaPiRE initiative," *IEEE Software*, vol. 35, no. 1, pp. 112–116, Jan. 2018.

[62] S. Wagner, D. M. Fernández, M. Felderer, and M. Kalinowski, "Requirements engineering practice and problems in agile projects: Results from an international survey," in *Iberoamerican Congress of Software Engineering (CibSE)*, 2017.

[63] L. Yu, E. Alégroth, P. Chatzipetrou, and T. Gorschek, "Utilising CI environment for efficient and effective testing of NFRs," *Information and Software Technology*, vol. 117, p. 106199, 2020.

[64] B. Fitzgerald and K.-J. Stol, "Continuous software engineering: A roadmap and agenda," *Journal of Systems and Software*, vol. 123, pp. 176–189, 2017.

[65] M. Fowler. (2006) Continuous integration. Accessed: April 2, 2020. [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[66] M. G. Jaatun, "Software security activities that support incident management in secure DevOps," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ser. ARES 2018. New York, NY, USA: Association for Computing Machinery, 2018.

[67] D. Cukier, "DevOps patterns to scale web applications using cloud services," in *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, ser. SPLASH '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 143–152.

[68] M. Shahin, M. Zahedi, M. A. Babar, and L. Zhu, "An empirical study of architecting for continuous delivery and deployment," *Empirical Software Engineering*, vol. 24, no. 3, pp. 1061–1108, Jun. 2019.

[69] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, “Development and Deployment at Facebook,” *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, Jul. 2013.

[70] X. Li, Y. F. Li, M. Xie, and S. H. Ng, “Reliability analysis and optimal version-updating for open source software,” *Information and Software Technology*, vol. 53, no. 9, pp. 929–936, Sep. 2011.

[71] S. Bellomo, N. Ernst, R. Nord, and R. Kazman, “Toward design decisions to enable deployability: Empirical study of three projects reaching for the continuous delivery holy grail,” in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Jun. 2014, pp. 702–707, iSSN: 2158-3927.

[72] G. McGraw, B. Chess, and S. Migues, “Building security in maturity model,” *Fortify & Cigital*, 2009.

[73] R. Colomo-Palacios, E. Fernandes, P. Soto-Acosta, and X. Larrucea, “A case analysis of enabling continuous software deployment through knowledge management,” *International Journal of Information Management*, vol. 40, pp. 186–189, 2018.

[74] D. Ståhl, K. Hallén, and J. Bosch, “Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the Eiffel framework,” *Empirical Software Engineering*, vol. 22, no. 3, pp. 967–995, Oct. 2016.

[75] K. Petersen and C. Wohlin, “The effect of moving from a plan-driven to an incremental software development approach with agile practices,” *Empirical Software Engineering*, vol. 15, pp. 654–693, 2010.

[76] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö, “The highways and country roads to continuous deployment,” *IEEE Software*, vol. 32, no. 2, pp. 64–72, Mar. 2015.

[77] A. Nilsson, J. Bosch, and C. Berger, “Visualizing testing activities to support continuous integration: A multiple case study,” in *Agile Processes in Software Engineering and Extreme Programming*, ser. Lecture Notes in Business Information Processing, G. Cantone and M. Marchesi, Eds. Cham: Springer International Publishing, 2014, pp. 171–186.

[78] T. Savor, M. Douglas, M. Gentili, L. Williams, K. Beck, and M. Stumm, "Continuous deployment at Facebook and OANDA," in *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*. Austin, Texas: ACM Press, 2016, pp. 21–30.

[79] L. Chen, "Continuous delivery: Overcoming adoption challenges," *Journal of Systems and Software*, vol. 128, pp. 72–86, Jun. 2017.

[80] R. Hoda and J. Noble, "Becoming agile: a grounded theory of agile transitions in practice," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 141–151.

[81] S. Gregor, "The nature of theory in information systems," *MIS Quarterly*, pp. 611–642, 2006.

[82] C. Wohlin, D. Šmite, and N. B. Moe, "A general theory of software engineering: Balancing human, social and organizational capitals," *Journal of Systems and Software*, vol. 109, pp. 229–242, 2015.

[83] R. Wieringa and M. Daneva, "Six strategies for generalizing software engineering theories," *Science of Computer Programming*, vol. 101, pp. 136–152, 2015.

[84] K.-J. Stol and B. Fitzgerald, "Uncovering theories in software engineering," in *2013 2nd SEMAT Workshop on a General Theory of Software Engineering (GTSE)*, 2013, pp. 5–14.

[85] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*. Springer, 2008, pp. 285–311.

[86] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies," *Empirical software engineering*, vol. 10, pp. 311–341, 2005.

[87] J. Saldaña, "Coding and analysis strategies," *The Oxford Handbook of Qualitative Research*, pp. 581–605, 2014.

[88] M. B. Miles and A. M. Huberman, *Qualitative data analysis: An expanded sourcebook*. sage, 1994.

[89] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst, and M.-A. Storey, "Uncovering the benefits and challenges of continuous integration practices," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2570–2583, 2021.

[90] D. Damian and D. Moitra, "Guest editors' introduction: Global software development: How far have we come?" *IEEE Software*, vol. 23, no. 5, pp. 17–19, 2006.

[91] J. D. Herbsleb and A. Mockus, "An empirical study of speed and communication in globally distributed software development," *IEEE Transactions on Software Engineering*, vol. 29, no. 6, pp. 481–494, 2003.

[92] M. Paasivaara, K. Blincoe, C. Lassenius, D. Damian, J. Sheoran, F. Harrison, P. Chhabra, A. Yussuf, and V. Isotalo, "Learning global agile software engineering using same-site and cross-site teams," in *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ser. ICSE '15. IEEE Press, 2015, pp. 285–294.

[93] J. M. Corbin and A. Strauss, "Grounded theory research: Procedures, canons, and evaluative criteria," *Qualitative sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[94] R. E. Boyatzis, *Transforming Qualitative Information: Thematic Analysis and Code Development*. sage, 1998.

[95] M. B. Miles, M. Huberman, and J. Saldana, *Qualitative data analysis: A methods sourcebook*. SAGE Publications, Incorporated, 2013.

[96] C. Marshall and G. B. Rossman, *Designing qualitative research*. Sage publications, 2014.

[97] Y. S. Lincoln and E. G. Guba, "Establishing dependability and confirmability in naturalistic inquiry through an audit." 1982.

[98] E. G. Guba, Y. S. Lincoln *et al.*, "Competing paradigms in qualitative research," *Handbook of qualitative research*, vol. 2, no. 163-194, p. 105, 1994.

[99] V. Bitsch, "Qualitative research: A grounded theory example and evaluation criteria," *Journal of agribusiness*, vol. 23, no. 345-2016-15096, pp. 75–91, 2005.

[100] C. Geertz, *Thick description: Toward an interpretive theory of culture.* Routledge, 2008.

[101] J. R. Landis and G. G. Koch, “The measurement of observer agreement for categorical data,” *Biometrics*, pp. 159–174, 1977.

[102] Z. S. Li, C. Werner, and N. Ernst, “Continuous requirements: An example using GDPR,” in *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW).* IEEE, 2019, pp. 144–149.

[103] A. R. da Silva, J. Caramujo, S. Monfared, P. Calado, and T. D. Breaux, “Improving the specification and analysis of privacy policies - the RSLingo4Privacy approach,” in *ICEIS 2016 - Proceedings of the 18th International Conference on Enterprise Information Systems*, 2016, pp. 336–347.

[104] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, “Measure it? manage it? ignore it? software practitioners and technical debt,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 50–60.

[105] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi, “Refactoring a legacy component for reuse in a software product line: a case study,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 2, pp. 109–132, 2006.

[106] M. Kim, T. Zimmermann, and N. Nagappan, “A field study of refactoring challenges and benefits,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE ’12. New York, NY, USA: Association for Computing Machinery, 2012.

[107] H. Saiedian and R. Dale, “Requirements engineering: making the connection between the software developer and customer,” *Information and Software Technology*, vol. 42, no. 6, pp. 419–428, 2000.

[108] Upguard. (2019) The cost of downtime at the world’s biggest online retailer. Accessed: February 18, 2020. [Online]. Available: https://web.archive.org/web/20200218200417/https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer

[109] A. T. T. Ying and M. P. Robillard, "Selection and presentation practices for code example summarization," in *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 460–471.

[110] M. Anderson, "Performance modelling of reactive web applications using trace data from automated testing," Ph.D. dissertation, University of Victoria, 2019.

[111] M. Anisetti, C. A. Ardagna, E. Damiani, F. Gaudenzi, and G. Jeon, "Cost-effective deployment of certified cloud composite services," *Journal of Parallel and Distributed Computing*, vol. 135, pp. 203–218, 2020.

[112] R. Nouacer, M. Djemal, S. Niar, G. Mouchard, N. Rapin, J.-P. Gallois, P. Fiani, F. Chastrette, A. Lapitre, T. Adriano *et al.*, "Equitas: A tool-chain for functional safety and reliability improvement in automotive systems," *Microprocessors and Microsystems*, vol. 47, pp. 252–261, 2016.

[113] M. Soni, "End to end automation on cloud with build pipeline: the case for DevOps in insurance industry, continuous integration, continuous testing, and continuous delivery," in *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*. IEEE, 2015, pp. 85–89.

[114] M. Cusumano, "Cloud computing and SaaS as new computing platforms," *Communications of the ACM*, vol. 53, no. 4, pp. 27–29, Apr. 2010.

[115] A. Benlian and T. Hess, "Opportunities and risks of software-as-a-service: Findings from a survey of IT executives," *Decision Support Systems*, vol. 52, no. 1, pp. 232–246, Dec. 2011.

[116] S. M. Greenstein, "Lock-in and the costs of switching mainframe computer vendors: What do buyers see?" *Industrial and Corporate Change*, vol. 6, no. 2, pp. 247–273, 03 1997.

[117] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 580–589.

[118] J. Opara-Martins, R. Sahandi, and F. Tian, "Critical review of vendor lock-in and its impact on adoption of cloud computing," in *International Conference on Information Society (i-Society 2014)*, Nov. 2014, pp. 92–97.

[119] ——, “Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective,” *Journal of Cloud Computing*, vol. 5, pp. 1–18, 2016.

[120] W. Maalej, M. Nayebi, T. Johann, and G. Ruhe, “Toward data-driven requirements engineering,” *IEEE Software*, vol. 33, no. 1, pp. 48–54, 2015.

[121] E. C. Groen, N. Seyff, R. Ali, F. Dalpiaz, J. Doerr, E. Guzman, M. Hosseini, J. Marco, M. Oriol, A. Perini *et al.*, “The crowd in requirements engineering: The landscape and challenges,” *IEEE Software*, vol. 34, no. 2, pp. 44–52, 2017.

[122] W. Behutiye, P. Karhapää, D. Costal, M. Oivo, and X. Franch, “Non-functional requirements documentation in agile software development: challenges and solution proposal,” in *International conference on product-focused software process improvement*. Springer, 2017, pp. 515–522.

[123] N. Misaghian and H. Motameni, “An approach for requirements prioritization based on tensor decomposition,” *Requirements Engineering*, vol. 23, no. 2, pp. 169–188, 2018.

[124] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, “Assessing the bus factor of git repositories,” in *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Mar. 2015.

[125] A. H. Van de Ven and M. S. Poole, “Explaining development and change in organizations,” *Academy of Management Review*, vol. 20, no. 3, pp. 510–540, 1995.

[126] C. Werner, Z. S. Li, N. Ernst, and D. Damian, “The lack of shared understanding of non-functional requirements in continuous software engineering: Accidental or essential?” in *2020 IEEE 28th International Requirements Engineering Conference (RE)*, 2020, pp. 90–101.

[127] C. Werner, Z. S. Li, D. Lowlind, O. Elazhary, N. Ernst, and D. Damian, “Continuously managing NFRs: Opportunities and challenges in practice,” *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2629–2642, 2022.

[128] C. M. Werner and D. M. Berry, “An empirical study of the software development process, including its requirements engineering, at very large organization: How

to use data mining in such a study," in *Requirements Engineering for Internet of Things*. Singapore: Springer Singapore, 2018, pp. 15–25.

[129] A. Cavoukian, "Privacy by design," 2009.

[130] D. Williams and D. Williams, "Exploring software project characteristics on scope creep," in *Procedings of Southern Association for Information Systems (SAIS)*, 2009.

[131] "Datadog: Cloud monitoring as a service." [Online]. Available: https://www.datadoghq.com/

[132] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed., ser. SEI Series in Software Engineering. Addison-Wesley Professional, 2012.

[133] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[134] J. Humble and G. Kim, *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution, 2018.

[135] "Continuous Integration." [Online]. Available: https://martinfowler.com/articles/continuousIntegration.html

[136] M. Kersten, *Project to Product: How to Survive and Thrive in the Age of Digital Disruption with the Flow Framework*. IT Revolution Press, 2018.

[137] N. Ernst, S. Bellomo, R. L. Nord, and I. Ozkaya, "Enabling incremental iterative development at scale: Quality attribute refinement and allocation in practice," *Software Engineering Institute, Tech. Rep. CMU/SEI-2015-TR-008*, p. 35, 2015.

[138] L. Compagna, P. El Khoury, A. Krausová, F. Massacci, and N. Zannone, "How to integrate legal requirements into a requirements engineering methodology for the development of security and privacy patterns," *Artificial Intelligence and Law*, vol. 17, no. 1, pp. 1–30, Mar. 2009.

[139] H. Femmer, D. Méndez Fernández, S. Wagner, and S. Eder, "Rapid quality assurance with requirements smells," *Journal of Systems and Software*, vol. 123, pp. 190–213, Jan. 2017.

[140] D. Ameller, C. Ayala, J. Cabot, and X. Franch, "Non-functional requirements in architectural decision making," *IEEE Software*, vol. 30, no. 2, pp. 61–67, 2012.

[141] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal, "Chaos engineering," *IEEE Software*, vol. 33, no. 3, pp. 35–41, 2016.

[142] S. Gupta, L. Ulanova, S. Bhardwaj, P. Dmitriev, P. Raff, and A. Fabijan, "The anatomy of a large-scale experimentation platform," in *International Conference on Software Architecture (ICSA)*, Apr. 2018.

[143] S. Neely and S. Stolt, "Continuous delivery? easy! just change everything (well, maybe it is not that easy)," in *2013 Agile Conference*, Aug. 2013, pp. 121–128.

[144] K.-T. Rehmann, C. Seo, D. Hwang, B. T. Truong, A. Boehm, and D. H. Lee, "Performance monitoring in SAP HANA's continuous integration process," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 4, pp. 43–52, 2016.

[145] M. Rath, J. Rendall, J. L. Guo, J. Cleland-Huang, and P. Mäder, "Traceability in the wild: automatically augmenting incomplete trace links," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 834–845.

[146] J. Schwarz, A. Steffens, and H. Lichter, "Code smells in infrastructure as code," in *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*. IEEE, 2018, pp. 220–228.

[147] A. Rahman, C. Parnin, and L. Williams, "The seven sins: Security smells in infrastructure as code scripts," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 164–175.

[148] K. Saatkamp, U. Breitenbücher, O. Kopp, and F. Leymann, "Topology splitting and matching for multi-cloud deployments." in *CLOSER*, 2017, pp. 247–258.

[149] R. Shu, X. Gu, and W. Enck, "A study of security vulnerabilities on docker hub," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, 2017, pp. 269–280.

[150] Z. A. Mann and A. Metzger, "Optimized cloud deployment of multi-tenant software considering data protection concerns," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 609–618.

[151] R. C. de Boer and H. van Vliet, "Writing and reading software documentation: How the development process may affect understanding," in *2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering*, May 2009, pp. 40–47.

[152] T. Clear, "Documentation and agile methods: striking a balance," *ACM SIGCSE Bulletin*, vol. 35, no. 2, pp. 12–13, 2003.

[153] V. Santos, A. Goldman, and C. R. De Souza, "Fostering effective inter-team knowledge sharing in agile software development," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1006–1051, 2015.

[154] K. C. Desouza and Y. Awazu, "Knowledge management at SMEs: five peculiarities," *Journal of knowledge management*, 2006.

[155] A. J. Ko, R. DeLine, and G. Venolia, "Information needs in collocated software development teams," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 344–353.

[156] J. Buchan, "An empirical cognitive model of the development of shared understanding of requirements," in *Requirements Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 432, pp. 165–179.

[157] M. Glinz, "A risk-based, value-oriented approach to quality requirements," *IEEE Software*, vol. 25, no. 2, pp. 34–41, Mar. 2008.

[158] M. Poppendieck and M. A. Cusumano, "Lean software development: A tutorial," *IEEE Software*, vol. 29, no. 5, pp. 26–32, 2012.

Appendix A

Publications

A.1 Publications from this Dissertation

[1] C. Werner, Z. S. Li, N. Ernst and D. Damian, "The Lack of Shared Understanding of Non-Functional Requirements in Continuous Software Engineering: Accidental or Essential?," in *IEEE 28th International Requirements Engineering Conference (RE)*, Zurich, Switzerland, pp. 90-101, 2020.

[2] C. Werner, Z. S. Li, D. Lowlind, O. Elazhary, N. A. Ernst and D. Damian, "Continuously Managing NFRs: Opportunities and Challenges in Practice," in *IEEE Transactions on Software Engineering*, pp. 2629-2642, no. 7, 2021.

[3] C. Werner, "Towards a Theory of Shared Understanding of Non-Functional Requirements in Continuous Software Engineering," in *Proceedings of the ACM & IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 300-304, 2022.

[4] C. Werner, L. Okpara, K. J. Stol and D. Damian, "A Theory on the Shared Understanding of Non-Functional Requirements in Continuous Software Engineering," *Pending Review in ACM Transactions on Software Engineering and Methodology*, 2023.

A.2 Other Publications

[1] C. Werner and D. Berry, "An Empirical Study of the Software Development Process, Including Its Requirements Engineering, at Very Large Organization:

How to Use Data Mining in Such a Study," in *Requirements Engineering for Internet of Things: 4th Asia-Pacific Symposium, APRES*, Melaka, Malaysia, November 9–10, 2017, Proceedings 4, pp. 15-25, Springer Singapore, 2018.

[2] C. Werner, G. Tapuc, L. Montgomery, D. Sharma, S. Dodos, and D. Damian, "How Angry are Your Customers? Sentiment Analysis of Support Tickets that Escalate," *2018 1st International Workshop on Affective Computing for Requirements Engineering (AffectRE)*, Banff, AB, Canada, pp. 1-8, 2018.

[3] T. Rae, C. Werner, and D. Damian, "Exploring Challenges In Adoption of Continuous Delivery," *CASCON Expo*, Markham, Ontario, Canada, 2018.

[4] C. Werner, Z. S. Li, and D. Damian, "Can a Machine Learn Through Customer Sentiment?: A Cost-Aware Approach to Predict Support Ticket Escalations," *in IEEE Software*, vol. 36, no. 5, pp. 38-45, Sept.-Oct, 2019.

[5] C. Werner, Z. S. Li, and N. Ernst, "What Can the Sentiment of a Software Requirements Specification Document Tell Us?," *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, Jeju, South Korea, pp. 106-107, 2019.

[6] Z. S. Li, C. Werner, and N. Ernst, "Continuous Requirements: An Example Using GDPR," *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, Jeju, South Korea, pp. 144-149, 2019.

[7] T. Rae, C. Werner, and D. Damian, "Design Guidelines to Overcome Continuous Development Adoption Challenges," *CASCON Expo.* Markham, Ontario, Canada, 2019.

[8] O. Elazhary, C. Werner, Z. S. Li, D. Lowlind, N. A. Ernst and M. A. Storey, "Uncovering the Benefits and Challenges of Continuous Integration Practices," *IEEE Transactions on Software Engineering*, pp. 2570-2583, no. 7, 2021.

[9] Z. S. Li, and C. Werner, "Ongoing Challenges and Solutions of Managing Data Privacy for Smart Cities," *In Smart Cities in Asia: Regulations, Problems, and Development*, pp. 23-32. Singapore: Springer Nature Singapore, 2022.

[10] Z. S. Li, C. Werner, N. Ernst, and D. Damian, "Towards Privacy Compliance: A Design Science Study in a Small Organization," *Information and Software Technology*, 146, 2022.

[11] L. Okpara, C. Werner, A. Murray, and D. Damian, "A Case Study of Building Shared Understanding of Non-Functional Requirements in a Remote Software Organization," *2022 IEEE 30th International Requirements Engineering Conference (RE)*, 2022.

[12] L. Okpara, C. Werner, A. Murray, and D. Damian, "The Role of Informal Communication in Building Shared Understanding of Non-functional Requirements in Remote Continuous Software Engineering," *Requirements Engineering*, pp. 1-23, (2023).

Appendix B

RQ1 Artifacts

B.1 RQ1 Focus Group Questions

1. Do you know when/why/how the rework in this task was done?
2. Do you know when/why/how the original work leading up to this task was done?
3. Which NFR or NFRs do you associate with the lack of shared understanding of this task?
4. Do you know why there was a lack of shared understanding of the NFR for this task?
5. What could the organization have done to prevent the lack of shared understanding for this task?
6. Do you know why this action was not taken for this task?
7. Do you believe this lack of shared understanding was avoidable for this task?
8. How complex do you believe this task was? (Small, Medium, or Large)
9. How important was this task to the organization? (High, Medium, or Low)
10. Do you have anything else to add?

B.2 RQ1 Complete Codebook

Table B.1: RQ1 Codebook

Code	Description	Total Tasks
BusinessContext	Talking about information from the business side of the organization	19
DifferentPriorityScale	There is an apparent difference or hesitation with the priority scale	23
Communication	Discussing something to do with communication	15
FeatureParity	Ensuring whatever are building has feature parity with an older version	10
Ambiguous	Usually something to do with an ambiguous item (such as a requirement)	13
MissedFeature	Failing to identify a feature or something else	10
NFRFunctional	Describing an NFR that includes functional elements	3
JustGetItToWork	Just get the MVP to function and get it out the door	24
TradeOff	The concept of doing one thing over another (where the other suffers)	17
EducationalRework	The original work was ok because it was learning what exactly we needed for the rework	15
ExtensibilityNFR	NFR: Extensibility	14
Standardization	Standardizing aspects of development (coding style, refactoring, process)	22

continues on next page

Table B.1 – continued from previous page

Code	Description	Total Tasks
ReworkUponRework	The rework kept manifesting, either repeatably as the same issue or in multiple places	18
MaintainabilityNFR	NFR: Maintainability	16
ConsistencyNFR	NFR: Consistency	4
ReadabilityNFR	NFR: Readability	2
LackOfKnowledge	Lack of knowledge, but necessarily a lack of shared understanding as we don't know	27
ChangeHereBreaksThere	The concept of changing one component without the realization of how vast the after affects are	10
Documentation	Talking about documentation	9
LotsOfInvestigation	There was a lot of investigation to bring the developer up to speed on the issue, regardless of how big the actual fix was	10
DifferentComplexityScale	There is an apparent difference or hesitation with the complexity scale	6
ReliabilityNFR	NFR: Reliability, Resiliency, Stability, Availability	17
DeployabilityNFR	NFR: Deployability	5
LackOfTests	There was a lack of testing	18
ThirdPartyIntegration	One of the sources of LSU for NFRs and rework is due to 3rd party integrations	13

continues on next page

Table B.1 – continued from previous page

Code	Description	Total Tasks
ChangingRequirements	The product and market is constantly changing as time goes on and thus the requirements change	17
LackOfResources	Money, people, time	7
ScalabilityNFR	NFR: Scalability	9
TransparencyNFR	NFR: Transparency	7
Tooling	Discussing how tooling can help resolve rework which might be a lack of shared understanding of an NFR	5
PrivacyNFR	NFR: Privacy	1
ConfigurationManagementNFR	NFR: Configuration Management	5
InformationOverload	There is just too much information at one time for everyone to equally process	4
UsabilityNFR	NFR: Usability	15
InteroperabilityNFR	NFR: Interoperability	6
AccessibilityNFR	NFR: Accessibility	2
TraceabilityNFR	NFR: Traceability	3
AutomatedPipeline	The continuous pipeline, could include integration, building, testing, deployment, etc	3
WhoCanMerge	Talks about who can review and merge code	1
DependencyNFR	NFR: Dependency	1
ReproducibilityNFR	NFR: Reproducibility	2
RevenueNFR	NFR: revenue	7
AuditabilityNFR	NFR: Auditability / Measurability	10

continues on next page

Table B.1 – continued from previous page

Code	Description	Total Tasks
MTTR	Mean time to repair	2
SecurityNFR	NFR: Security	4
PerformanceNFR	NFR: Performance	6
DataIntegrityNFR	NFR: Data Integrity	2
TestabilityNFR	NFR: Testability	1

Appendix C

RQ2 Artifacts

C.1 RQ2 Interview Questions

1. Are you familiar with the term DevOps? Are you familiar with the term continuous (integration, delivery, deployment)?

2. If you are familiar, how do you define these terms?

3. Does your organization practice any continuous practices or DevOps? If so, what are they?

4. Do you define NFRs? If the interviewee answers no, we can first provide an example of an FR to help an interview conceptualize. If the interviewee is still confused, we can provide an example of a quality attribute of a system (i.e. performance requirements) or activities that the organization conducts to ensure a specific quality. Those activities may be using infrastructure as code to improve maintainability.

5. How do you define an NFR?

6. Which NFRs are important to your organization?

7. How do you manage NFRs?

8. How do you document an NFR?

9. In particular, are any NFRs in your source control?

10. How do you test or ensure an NFR is satisfied?

11. How do you trace an NFR through the continuous development?
12. What happens when an NFR fails? Is there a feedback loop from continuous development?
13. Does an NFR require additional resources (additional approval, testing, etc?) when developing it?
14. How do you ensure everyone is aware of a particular NFR?

C.2 RQ2 Complete Codebook

Table C.1: RQ2 Complete Codebook

Code	**Description**	**Count**
AccessibilityNFR	NFR: Accessibility of software for users with disability or special needs	5
AccuracyNFR	NFR: Accuracy of software, data, or process	2
Automation	Discussing the automation of something	93
AvailabilityNFR	NFR: Availability of service or software to users	39
Challenges	A challenge of some sort	71
Codification	The concept of codifying things	92
CollaborativeOverload	Mythical man month too much communication	5
Confidence	When the interviewee talks about their confidence in a process usually the pipeline	21
ConfigurabilityNFR	NFR: Configurability	103
ContinuousPerception	Perception of continuous in terms of explaining the definition of continuous. Could also refer to whether or not knowing the term continuous	137
CostNFR	NFR: Cost of providing implementing, provisioning, or hosting of service or software	7

continues on next page

Table C.1 – continued from previous page

Code	Description	Count
CustomTooling	Bash scripts, extensions of existing tools, or created from scratch	25
DataIntegrityNFR	NFR: Data Integrity	23
DependencyNFR	NFR: Dependency	16
DeployabilityNFR	NFR: Deployability	22
Deployment	After a PR is merged how does it get released	101
Development	The development aspect of coding (usually up to submission of PR)	37
DevOpsPerception	Perception of DevOps in terms of explaining the definition of devops. Could also refer to knowing or not knowing the term devops	133
Documentation	General documentation of something	109
EfficiencyNFR	NFR: Efficiency of the outcome (ie the output) usually measured by how much you use	15
Evolution	How something (ie an NFR) changes or evolves over time.	32
Explicit	Something that is explicit. Written down.	60
ExtensibilityNFR	NFR: Extensibility	6
FeedbackLoop	The concept of receiving feedback from the continuous pipeline (or customers) regarding the software in a systematic manner	46
Implicit	Something that is implicit. Not written down	64
ImplOfNFR	The implementation of an NFR (whether as code or not)	78
InteroperabilityNFR	NFR: Interoperability	5
KnowledgeTransfer	Between people or entities of an organization	78
MaintainabilityNFR	NFR: Maintainability	36
Manual	Doing something manually (ie not automated)	95
Merging	From PR to testing env (staging) until PR is merged.	66
Metrics	Creating or monitoring of a quantitative value	154

continues on next page

Table C.1 – continued from previous page

Code	Description	Count
ModifiabilityNFR	NFR: Modifiability	3
ModularityNFR	NFR: Modularity	11
NFROutsourcing	Relinquishing technical control/responsibility of said NFR to an external entity.	155
NFRPerception	Individual's perception of NFRs	219
Organizational	Something to do with the organization (business, technical, etc)	112
PerformanceNFR	NFR: Performance of the outcome (ie the output) usually measured by time	88
PortabilityNFR	NFR: Portability	14
PrivacyNFR	NFR: Privacy	15
ReadabilityNFR	NFR: Readability	31
RecoverabilityNFR	NFR: Recoverability	10
ReproducibilityNFR	NFR: Reproducibility	52
ReusabilityNFR	NFR: Reusability	28
RevenueNFR	NFR: Revenue	23
Risk	The potential for exploitation of a vulnerability by a threat	10
ScalabilityNFR	NFR: Scalability	56
SecurityNFR	NFR: Security	68
SiloOfRoles	Existence of proverbial wall. Could also refer to devs not understanding ops and vice versa	51
SimplicityNFR	NFR: Simplicity	3
SoftwareArchitecture	Such as GoF design patterns, anti-patterns, etc	52
SomethingWeShouldDo	Acknowledging that interviewee or organization can improve something	44
StabilityReliabilityNFR	NFR: StabilityReliability refers to any mention of stability or reliability of software or service	41
TestabilityNFR	NFR: Testability	47

continues on next page

Table C.1 – continued from previous page

Code	Description	Count
Testing	The testing aspect of development / integration	166
Tooling	Off the Shelf (Kubernetes, Docker, Terraform, etc)	262
TossMoney	General sense to simply toss money at the problem (especially at cloud platforms).	7
TraceabilityNFR	NFR: Traceability	5
TradeOff	Deciding between two important alternatives (i.e. writing unit tests or fixing a bug)	41
TransparencyNFR	NFR: Transparency	26
UsabilityNFR	NFR: Usability	54