The Lack of Shared Understanding of Non-Functional Requirements in Continuous Software Engineering: Accidental or Essential?

Colin Werner, Ze Shi Li, Neil Ernst, Daniela Damian
Department of Computer Science
University of Victoria, Victoria, Canada
{colinwerner, lize, nernst, danielad}@uvic.ca

Abstract—**Building shared understanding of requirements is key to ensuring downstream software activities are efficient and effective. However, in continuous software engineering (CSE) some lack of shared understanding is an expected, and essential, part of a rapid feedback learning cycle. At the same time, there is a key trade-off with avoidable costs, such as rework, that come from accidental gaps in shared understanding. This trade-off is even more challenging for non-functional requirements (NFRs), which have significant implications for product success. Comprehending and managing NFRs is especially difficult in small, agile organizations. How such organizations manage shared understanding of NFRs in CSE is understudied. We conducted a case study of three small organizations scaling up CSE to further understand and identify factors that contribute to lack of shared understanding of NFRs, and its relationship to rework. Our in-depth analysis identified 41 NFR-related software tasks as rework due to a lack of shared understanding of NFRs. Of these 41 tasks 78% were due to avoidable (accidental) lack of shared understanding of NFRs. Using a mixed-methods approach we identify factors that contribute to lack of shared understanding of NFRs, such as the lack of domain knowledge, rapid pace of change, and cross-organizational communication problems. We also identify recommended strategies to mitigate lack of shared understanding through more effective management of requirements knowledge in such organizations. We conclude by discussing the complex relationship between shared understanding of requirements, rework and, CSE.**

Index Terms—**shared understanding, non-functional requirements, continuous software engineering, rework**

I. Introduction

Shared understanding is a critical success factor in achieving high quality software that meets stakeholders' needs [1]. It also fosters a more effective, motivated, and collaborative team, more efficient use of resources, and less conflict [2]. However, empirical research on shared understanding in software engineering projects is scarce [3].

Another critical success factor in software is non-functional requirements (NFR) [**?**], [4], also known as architecturally significant requirements [5]. 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 continuous software engineering (CSE), similar to agile software development, functional requirements (FR) are the primary focus, while NFRs are neglected [6]. CSE, especially in small, agile organizations, focuses on automated, rapid release of working software, and often leads to short-term FR prioritization [7].

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 [8] and accounts for 40-50% of the effort on software projects [9]. Some rework may be due to an avoidable lack of shared understanding, which we define as **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 [10], it captures the unknown unknowns [11] and represents desirable learning and feedback [12].

In this paper we describe empirical evidence from a multi-case, mixed-method study of shared understanding of NFRs in three small, agile organizations employing CSE. To conceptualize lack of shared understanding of NFRs we traced it to rework development tasks. Seeking to shed light on the complex relationship between shared understanding, rework, and CSE, our study examined forty-one NFR-related development tasks identified as rework and was driven by the following research questions:

RQ1: What contributes to lack of shared understanding of NFRs?

RQ2: Which NFRs are most associated with a lack of shared understanding?

RQ3: What amount of a lack of shared understanding of NFRs is **accidental** versus **essential**?

In our close, iterative interactions with our industrial partners, we learned what factors contribute to a lack of shared understanding of NFRs, a large proportion of rework was due to an **accidental** lack of shared understanding, and strategies our collaborators recommended to avoid it. Our work brings two significant contributions:

1) We add to the scarce, yet much needed, empirical evidence on shared understanding of NFRs in software engineering, including contributing factors and practices towards avoiding it.
2) We bring awareness, based on empirical findings, to challenges that CSE brings to shared understanding of NFRs.

Finally, our study provides important implications for both research and practice. For research, there is a need to address new challenges brought forth by CSE in RE by revisiting supporting roles, methods, and tools for the identification, documentation, and communication of requirements. For practitioners, we outline how to document and communicate NFRs to avoid a lack of understanding in CSE.

II. Background and Related Work

A. Shared Understanding

Creating a shared understanding is an important aspect and challenge to consider when attempting to develop high-quality software [13], especially in RE [14]. 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" [3]. Glinz and Fricker [3] 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.

The most important goal of RE is to create a shared understanding between the development team and stakeholders [15]; moreover, the practice of creating and maintaining a shared understanding is not well established [16]. Previous work has shown that a misunderstanding of requirements at the beginning of development resulted in substantial rework [17]. Unfortunately, shared understanding is *often* passive, informal, and unstructured [3]; the result of which negatively affects software quality and necessitates rework [18], which is further compounded with the cross-cutting nature of NFRs. A recent survey [19] stipulates creating a shared understanding of customer expectations is the top challenge in agile adoption, yet understudied [3]. However, 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.

B. Non-Functional Requirements

Despite their importance to success [3], NFRs are ambiguous, not well documented, shared in an informal matter, and not well understood [20], [21]. The wide-ranging and extensive NaPiRE study [22] found that "unclear / unmeasurable" NFRs were one of the top problems respondents had in their small organizations [23]. What some organizations think are NFRs are actually FRs [24]. NFRs are also inherently difficult to verify or validate [25], [26]. NFRs are cross-cutting across multiple disciplines, especially in a short, fast paced software development iterations [27]. Finally, the inability to decompose an NFR into measureable and valuable artifacts that can be completed within a release cadence [27] further increases the difficulty in managing an NFR in CSE. This difficulty impedes the forming of a shared understanding of that NFR and can result in costly rework. A number of empirical studies have been performed on the challenges of NFRs in practice [28]–[30]; however, these studies do not focus on identifying lack of shared understanding of NFRs in CSE.

C. Rework

Various definitions and classifications of rework have been proposed, such as Swanson's three classification types (corrective, adaptive, and perfective) [31] or Basili's classification based on the source of rework [32]. Regardless of the classification and the large costs associated with rework [32], there is consensus that some rework could *not* have been prevented [9] and is acceptable. In fact, a certain amount of rework is desirable [33], as no rework could indicate developers are not performing their jobs with due diligence, popularized in the ideas captured by Ries [34]. Technical debt due to NFRs increases the number of quality issues, which can worsen and compound rework upon rework in later development phases [35]. False implicit shared understanding [3] of NFRs can lead to substantial technical debt and ultimately force an organization to perform major rework [36].

D. Continuous Software Engineering

CSE is a paradigm that emphasizes rapid and automated releases of working software [37]. This suggests that RE, including the treatment of NFRs, are more challenging in CSE, whereby many common best practices may not be followed and requirements may be informally captured "just-in-time" [38].

A number of studies [39]–[44] 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 which Facebook monitors to evaluate the PERFORMANCE of their system [42]. While metrics are a key strategy to operationalize and create a shared understanding of an NFR in CSE, Perflab is self-proclaimed as "not supported" and "not turn-key".

For software SECURITY, Jaatun argues that proper attention to incident management, including involding and educating developers, can help alleviate SECURITY issues in CSE [39]; although this is strictly for SECURITY and is primarily focused on the Building Security In Maturity Model [45].

Other studies [40], [41], [44] 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 significant influence on a system's architecture, including positives, negatives, and trade-offs. While these studies show how to architect a system in CSE, they do not look at how to create a true shared understanding of explicit NFRs in CSE. Furthermore, no empirical evidence exists that measures the effect of shared understanding of NFRs [3], let alone in CSE. Thus, our goal is to fill this gap in research by explicitly focusing on the state of the practice of shared understanding of NFRs in CSE.

III. Research Methodology

We conducted a multi-case study [46] using a mixed-methods approach [47] in collaboration with three independent organizations using qualitative and immersive techniques [48]. To study lack of shared understanding of NFRs in relation to rework, we investigated how the organizations handled NFRs and specifically traced lack of shared understanding on NFRs to rework tasks. In an iterative, collaborative process with our partner organizations, we performed an in-depth analysis of such tasks to identify reasons for lack of shared understanding, including whether it was **accidental** or **essential**, and which practices to employ to avoid lack of shared understanding.

A. Collaborating Organizations

We identified three small, agile organizations using CSE and cloud-based platforms through local contacts and trade shows. Our organizations shared development artifacts, granted access to employees, and participated in focus groups. For this paper our organizations are referred to as Alpha, Beta, and Gamma. Each organization develops in-house software utilizing cloud-based platforms, which contributes to their primary source of revenue. 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 [49]–[51].

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.

B. Preliminary Study

We first sought to understand which NFRs were relevant to our organizations and what processes and tools they employ to handle NFRs. One author spent multiple days over a few months embedded at each organization, learning about their products, processes, and business [48]. Through our immersive visits we spoke with 37 different team members from various departments. Across our organizations, we spoke with 17 developers, 8 development managers, 5 product managers, 4 executives, 2 customer success specialists, 1 quality assurance member, and 1 director of sales. As part of these informal conversations, we discussed shared understanding, NFRs, and rework, including how much each organization estimates they spend on rework, how they manage tacit knowledge, and NFRs that cause rework. Our early immersive meetings and observations at our organizations informed our focused investigation into the lack of shared understanding of NFRs and its relationship to rework.

C. Data Collection

To analyze lack of shared understanding of NFRs as it materialized in rework, we performed an in-depth analysis of development tasks at each organization, where each task represented either a bug, feature, a story, or an epic, depending on the context at each organization. We analyzed and systematically coded these development tasks from task-management tools 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 our organizations over two full months. We indicate below the full time taken for specific analysis activities.

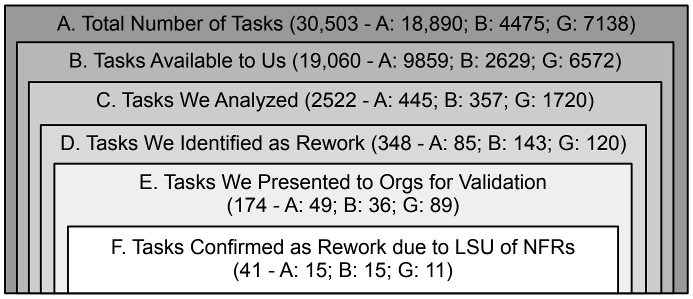
A. Total Number of Tasks (30,503 - A: 18,890; B: 4475; G: 7138)
B. Tasks Available to Us (19,060 - A: 9859; B: 2629; G: 6572)
C. Tasks We Analyzed (2522 - A: 445; B: 357; G: 1720)
D. Tasks We Identified as Rework (348 - A: 85; B: 143; G: 120)
E. Tasks We Presented to Orgs for Validation (174 - A: 49; B: 36; G: 89)
F. Tasks Confirmed as Rework due to LSU of NFRs (41 - A: 15; B: 15; G: 11)

Fig. 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))

1) 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. We exported all available tasks, and associated attributes through a script over one week and spent another week understanding the data. The resulting data contained 30,503 tasks across our organizations (Figure 1 Box A). Due to the large number of tasks and limited time with each organization, we performed task sampling over four weeks to achieve a more practical and manageable number of tasks for our analysis, as summarized in Figure 1.

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

2) 1st Member Checking and Validation of Rework Tasks: We then confirmed if each task actually represented rework and, more importantly, if that rework was due to lack of shared understanding of NFRs. We held a number of meetings at each organization to discuss our selection of 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. We started off by clarifying the definition of NFRs, shared understanding, rework, and the relationship between the

TABLE I
BASE FOCUS GROUP QUESTIONS

Q#	Question
Q1	Do you know when/why/how the rework in this task was done?
Q2	Do you know when/why/how the original work leading up to this task was done?
Q3	Which NFR or NFRs do you associate with the lack of shared understanding of this task?
Q4	Do you know why there was a lack of shared understanding of the NFR for this task?
Q5	What could the organization have done to prevent the lack of shared understanding for this task?
Q6	Do you know why this action was not taken for this task?
Q7	Do you believe this lack of shared understanding was avoidable for this task?
Q8	How complex do you believe this task was? (Small, Medium, or Large)
Q9	How important was this task to the organization? (High, Medium, or Low)
Q10	Do you have anything else to add?

three to ensure consistency. Due to partner time constraints, we discussed only 174 of the 348 tasks *we* identified as rework (Figure 1 Box E). Overall, our organizations confirmed 41 tasks as rework due to lack of shared understanding of NFRs (Figure 1 Box F). These 41 tasks were used as the basis for our in-depth analysis through further focus group sessions at each organization.

3) Focus Group Sessions on Rework Associated with Lack of Shared Understanding of NFRs: We scheduled focus group sessions at each organization to discuss each organization's share of the 41 rework tasks. We examined the factors contributing to lack of shared understanding (RQ1), which NFRs were more prone to lack of shared understanding (RQ2), and what amount of lack of shared understanding could be avoided (**accidental**) versus **essential** in the complex relationship with rework (RQ3). 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. We performed 2 focus group sessions at each organization. The base set of questions for each of the 41 tasks is in Table I. We clarified the definition of NFRs, shared understanding, rework, avoidability, and the relationship between the four in a similar fashion to our 1st member checking phase. For example, we ensured that each participant equally understood avoidable in the sense that the organization could have prevented the lack of shared understanding by taking action prior to the rework task arising. These sessions were, with permission, audio-recorded.

D. Data Analysis

To answer our research questions, our data analysis triangulated contextual data from our 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 lack of shared understanding of NFRs. The key to our analysis of the rich, rework task data was the contextual, organizational knowledge we acquired during the preliminary study at each organization.

To analyze the qualitative data, we transcribed the recordings of the focus-group sessions to obtain 41 transcripts, taking one week. We performed thematic analysis [52] spanning 4 weeks on these transcripts. We used an open coding [53] approach to develop a purely inductive codebook [52], 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 to calculate Cohen's kappa coefficient. We continued until the kappa value stabilized above 0.6 (substantial inter-rater reliability agreement [54]), 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 we used the constant comparison method; codes were added, removed, and merged based on the discussions between the coders. Our final codebook took two weeks to compile and had 48 codes (85% of the codes were used in the first 10 transcripts). The complete codebook is in our replication package.

To answer RQ1 and RQ3, we employed thematic synthesis [52] over a two week period to abstract themes from our derived codes. We constructed themes 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). Similarly, a number of themes emerged related to the **accidental** versus **essential** nature of shared understanding (RQ3), including practices to avoid a lack of shared understanding. We performed a 2nd round of member checking with our organizations to validate these themes. For RQ2, we counted the number of references to particular NFRs for each task.

IV. FINDINGS

RQ1: What Contributes to Lack of Shared Understanding of NFRs? We asked participants if they knew why there was lack of shared understanding of NFRs for each of the 41 tasks (Q4). In our data analysis three themes emerged: fast pace of change, lack of domain knowledge, and inadequate communication. The themes and associated codes are in Table II.

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*

TABLE II
SUMMARY OF FACTORS CONTRIBUTING TO LSU OF AN NFR (RQ1)

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

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 our 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).*

2) 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 [3] mentions domain knowledge as a problem to building shared understanding. Our 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 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 [11], 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 customer's would utilize a particular feature.

We also observed an organization's difficulty in adapting to new horizontal markets with respect to localization techniques that 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.

We also found 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, we observed assessing an NFR 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 [55]. Our 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 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.

3) 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

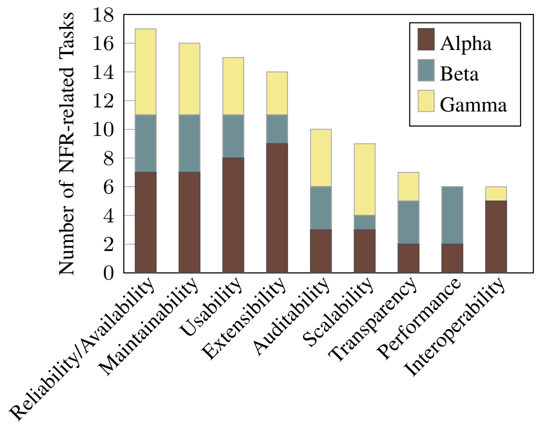

Fig. 2. NFRs with ≥ 6 NFR-related tasks (RQ2)

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 break-downs 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.

RQ2: Which NFRs are Most Associated with Lack of Shared Understanding? We asked each organization what NFRs were associated with each task (Q3). The results of NFRs with at least six or more associated NFR-tasks are in Figure 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.

RQ3: What Amount of a Lack of Shared Understanding is Accidental versus Essential? Recall we asked participants whether the lack of shared understanding was avoidable (Q7). We also asked participants 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 our organizations. The remaining 22% represents a **essential** lack of shared understanding.

Practices to Avoid Lack of Shared Understanding 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 our research team was the first time the organization had realized the true cost of lack of shared understanding.

a) Shared Development Standards (22 of 41 tasks): Our organizations believed 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. Our 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.

b) Adequate Communication and Documentation (19 of 41 tasks): Unsurprisingly, inadequate communication was one of the main contributors to 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. We observed communication issues 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 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 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 understanding what's going on during that time frame"*. Without documentation, tracking decisions and actions may be difficult. Moreover, on-boarding and training is challenging without documentation serving as a guide.

V. DISCUSSION

In this work we aimed to empirically investigate shared understanding in RE. In particular, we studied the lack of shared understanding of NFRs and its effects as traced in the form of rework in software projects. Using mixed-methods, we studied three small, agile organizations practicing CSE and using cloud-based platforms. We found some NFRs (RQ2) (RELIABILITY/AVAILABILITY, MAINTAINABILITY, USABILITY, and EXTENSIBILITY) are more prone to lack of shared understanding in these organizations, and the majority of this lack of shared understanding (78%) was **accidental**, i.e. avoidable, whereas 22% represented **essential**, i.e. not avoidable, lack of shared understanding (RQ3).

From our analysis three factors contributing to the lack of shared understanding (RQ1) emerged: fast pace of change, lack of domain knowledge, and inadequate communication. We start our discussion of these findings in the context of our organizations' operating environment. In particular, these were agile organizations practicing CSE, which we believe may pose additional challenges to the treatment of NFRs. We aim to shed some light on the complex relationship between shared understanding, rework, and CSE.

We also identified two practices (RQ3) (shared development standards and adequate communication and documentation) whereby organizations may benefit in building a shared understanding and awareness of NFRs. However, organizations have a fixed set of resources and thus need to strategically prioritize when to build a shared understanding against developing new features. Our next point in discussion will be on *when* and *how* organizations could employ these strategies to build shared understanding in the context of CSE.

We believe these findings have important research implications for the intersection of shared understanding of NFRs and CSE. In addition, our final discussion point refers to practical implications from our findings, including adaptations we believe may enable a more proactive approach to avoid rework due to an **accidental** lack of shared understanding.

A. Shared Understanding and CSE

At first sight, CSE advocates for ongoing customer feedback and rapid iteration cycles [37] that should result in a heightened understanding of customer requirements. In our deeper empirical investigation, our study reveals the potential for a much more complex relationship between lack of shared understanding (as materialized in rework) and practices of CSE. Our findings 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 [56]. A quick feedback loop is associated with lowering the impact of lack of shared understanding [3]. However, as NFRs are cross cutting and difficult to decompose into sub-components that can be completed within a short iteration [27], 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 our organizations, we hypothesize that CSE hurts the shared understanding of NFRs, much like how agile RE practices increase the risk of overemphasis of FRs [6]. A notable complication is CSE has a broader scope than agile, as not only do you need to slice NFRs [57], but you need to subsequently ship them in rapid, frequent deployments [58].

We found NFRs were 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. 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 our 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 [3].

Another claimed benefit of CSE is the removal of intra-organizational barriers, such as between product managers, DevOps, and developers [37]. 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, we found that 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 intuition 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 a 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.

Our findings 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 our 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 [59], were often neglected or underspecified [60] leading to a lack of shared understanding. We found the primary difference between an agile product owner and our organization's 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 are part of the automated CSE build [3]. However, we found that our organizations did not heavily invest in substantial testing efforts, even for FRs. Furthermore, NFRs are difficult to test [25], [26] and may require specific architectural considerations to be automatically tested [61], therefore NFRs are largely left as a manual task in CSE [62]. 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 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 (front-end) without building the other (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 [63]; these imperfect tests are the first step in acknowledging and bringing awareness to the value of the NFR and start fostering a shared understanding.

Research Implication 1

Our evidence suggests that one possible side-effect of CSE is a decrease in the shared of understanding of NFRs that leads to rework. There is a need for further exploratory studies to confirm this, as well as defining roles, methods, and tools to identify and mitigate the potential lack of shared understanding in CSE. How could revised methodologies leverage a partnership and mutual respect between product managers and developers to ensure adequate dissemination of NFR and domain knowledge in CSE organizations?

B. What Triggered Our Organizations to Build a Shared Understanding?

Our 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 we discussed in the preceding section, our 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, our organizations relied on reactive stimuli. They used different triggers to identify the need for building shared understanding of NFRs. These triggers were regulatory requirements, accumulating technical debt, needs of important customers, and disruption of service.

1) 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 [55]. Explicit regulatory policies, such as PRIVACY, can be extracted, visualized, and re-published to ensure a higher quality of shared understanding between various stakeholders [64].

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 [55]. Employees need to understand the risk of finding and using an external library without first vetting the library for PRIVACY considerations.

2) Accumulating Technical Debt: At our organizations, another trigger was to build a shared understanding of NFRs when the organization had incurred significant technical debt as a result of **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 [65]. Refactoring software has many benefits, in particular for NFRs, such as increased RELIABILITY, MAINTAINABILITY, and REUSABILITY [66], [67]. However, we believe 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 rearchitecture 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, we found that refactoring can potentially *increase* the shared understanding of NFRs.

3) Needs of Important Customer: As our organizations have a broad set of customers, each of which must be satisfied, it is vitally important to identify and prioritize perspective customers [68]; however, often the focus is on FRs. At two of our organizations (Alpha and Gamma) we found that a shared understanding was triggered 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.

4) 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 [69]. Disruption of service usually involves a loss of functionality, and subsequent rework to patch the problem. The silver lining in a disruption is that we found our 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 our study 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.

Research Implication 2

The fast paced CSE environment is one possible explanation why an organization would be more reactive to triggers, such as technical debt, as opposed to proactive in building a shared understanding of NFRs. We believe there is a need for further 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 shared understanding? How can we identify reactive triggers before rework occurs and use them as proactive triggers? Is there a difference in the level of shared understanding between a reactive and proactive trigger?

C. Implications to Practice: How Could Organizations Build a Shared Understanding in CSE?

Upon encountering a trigger (i.e. 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, our organizations are largely reactive to rework due to **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 [18]. Furthermore, the participants in our study suggested two practices that could help them build a shared understanding: shared development standards and documentation and communication. We discuss practical implications as suggested by our empirical findings and which we believe 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 [70], we do not believe the documentation needs 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 [71]. While determining what is considered a sufficient level of documentation might be different at each organization [72], 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 [73].

An organization should first select a single, vitally important NFR, such as PERFORMANCE or RELIABILITY based on their own context. In the most simplistic manner possible, the NFR must be documented to ensure cross-functional visibility. All of our organizations maintain some form of dashboard that helps increase cross-functional visibility, thus we believe these dashboards are able to 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 [74], [75], 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 [76]. Establishing a cognitive shared understanding involves identifying insufficiency in an organization’s understanding and rectifying misunderstandings [76]. 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 [27], [77] is to ensure NFRs can be quantified, for example as response measures or acceptance criteria. As part of their efforts for AUDITABILITY, our collaborating organizations actively add metrics to catch errors and increase insight on their system. We recognize that some NFRs are more difficult to verify than others; although even PRIVACY has been managed through custom tools that automatically tests for software deficiencies [55]. 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 [78], 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, we believe that building awareness and educating the organization regarding the value behind building a shared understanding of NFRs, including priority, is paramount to software success.

VI. Threats to Validity

To ensure validity of our qualitative research, we provide a replication package (https://doi.org/10.5281/zenodo.3671463), including statistics of the analyzed tasks, codebook, and inter-rater agreement sessions. Unfortunately, due to non-disclosure and ethics agreements we are unable to reveal full transcripts.

We discuss threats present in data gathering, data analysis, and the results. When gathering data, we mitigated the effects of respondent bias (the observer effect) through two steps. First, we assured the participants that our presence was not to critique them, but rather to acquire insight about lack of shared understanding of NFRs in their respective organizations. Second, we discussed with two participants simultaneously for each focus group and encouraged the 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) we discussed the definitions prior to our questions. For construct validity, while interviewing participants we defined and used the term avoidable lack of shared understanding; however, in our theoretical conceptualization we use the creative analogy of **accidental** or **essential**. We also used transcription tools 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 our task sampling, as we did not analyze every task. This was due to the limited time we had with each organization and restricted access to the data. However, the organization participants felt our selected tasks were representative of tasks in their context.

With respect to credibility in data analysis, we conducted pair coding until we reached a point where we consistently achieved a moderate to substantial level of inter-rater agreement. Second, we conducted two rounds of member checking with our organizations and participants. The first round of member checking involved the organizations confirming the 41 tasks were rework as a result of lack of shared understanding of an NFR. The second round of member checking verified our findings and themes, including factors contributing to and practices to avoid lack of shared understanding of an NFR. We elicited ordinal feedback (Strongly Disagree-Strongly Agree) on each of our themes and practices. 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 our organizations. Finally, for conclusion validity we recognize that our research implications are phrased as hypotheses, *not* factual conclusions, requiring further research investigations.

VII. Conclusion

Effectively building and managing a shared understanding of requirements is key to successful software projects. NFRs pose additional challenges given their cross-cutting nature that makes them more difficult to handle, particularly in CSE deployments.

Evidence from our study suggests that CSE negatively affects the shared understanding of NFRs, resulting in important implications to research that should reconsider RE in the context of CSE. Our research renews and refocuses our awareness on the complex relationship between rework and shared understanding in CSE.

With respect to practical implications of our study, 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.

Acknowledgments

We thank our three partner organizations and employees for their time and collaboration. We acknowledge Omar Elazhary, Austin Lee, and Derek Lowlind (University of Victoria) for their assistance in our study. Our research was supported by grant NSERC-CRD 535876.

References

[1] 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.

[2] 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.

[3] 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.

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

[5] L. Chen, M. A. Babar, and B. Nuseibeh, "Characterizing architecturally significant requirements," *IEEE software*, vol. 30, no. 2, pp. 38–45, 2012.

[6] 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.

[7] 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, event-place: Gothenburg, Sweden.

[8] 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.

[9] 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.

[10] F. Brooks and H. Kugler, *No silver bullet*. April, 1987.

[11] 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.

[12] E. Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, 1st ed. New York: Currency, Sep. 2011.

[13] 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.

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

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

[23] 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.

[24] J. Eckhardt, A. Vogelsang, and D. M. Fernández, "Are "Non-functional" Requirements really Non-functional? An Investigation of Non-functional Requirements in Practice," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 832–842.

[25] 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," 2003.

[26] 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.

[27] S. Bellomo, N. A. Ernst, R. L. Nord, and I. Ozkaya, "Evolutionary Improvements of Cross-Cutting Concerns: Performance in Practice," *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 545–548, 2014.

[28] 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.

[29] 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.

[30] 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.

[31] 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, event-place: San Francisco, California, USA.

[32] 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.

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

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

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

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

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

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

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

[49] 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.

[50] 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.

[51] 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.

[52] 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.

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

[54] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[55] 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.

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

[57] N. Ernst, S. Bellomo, R. L. Nord, and I. Ozkaya, "Enabling Incremental Iterative Development at Scale: Quality Attribute Refinement and Allocation in Practice," p. 35, 2015.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] A. Caracciolo, M. F. Lungu, and O. Nierstrasz, "How Do Software Architects Specify and Validate Quality Requirements?" in *Software Architecture*, ser. Lecture Notes in Computer Science, P. Avgeriou and U. Zdun, Eds. Springer International Publishing, 2014, pp. 374–389.

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

[64] A. R. d. Silva, J. Caramujo, S. Monfared, P. Calado, and T. Breaux, "Improving the Specification and Analysis of Privacy Policies - The RSLingo4Privacy Approach," Feb. 2020, pp. 336–347.

[65] 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.

[66] 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.

[67] 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.

[68] 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.

[69] Upguard. (2019) The cost of downtime at the world's biggest online retailer. [Online]. Available: https://web.archive.org/web/20200218200417/https://www.upguard.com/blog/the-cost-of-downtime-at-the-worlds-biggest-online-retailer

[70] 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, iSSN: null.

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

[72] 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.

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

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

[75] 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.

[76] J. Buchan, "An Empirical Cognitive Model of the Development of Shared Understanding of Requirements," in *Requirements Engineering*, S. D. Junqueira Barbosa, P. Chen, A. Cuzzocrea, X. Du, J. Filipe, O. Kara, I. Kotenko, K. M. Sivalingam, D. Ślęzak, T. Washio, X. Yang, D. Zowghi, and Z. Jin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, vol. 432, pp. 165–179.

[77] M. Glinz, "A Risk-Based, Value-Oriented Approach to Quality Requirements," *IEEE Software*, vol. 25, no. 2, pp. 34–41, Mar. 2008.

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