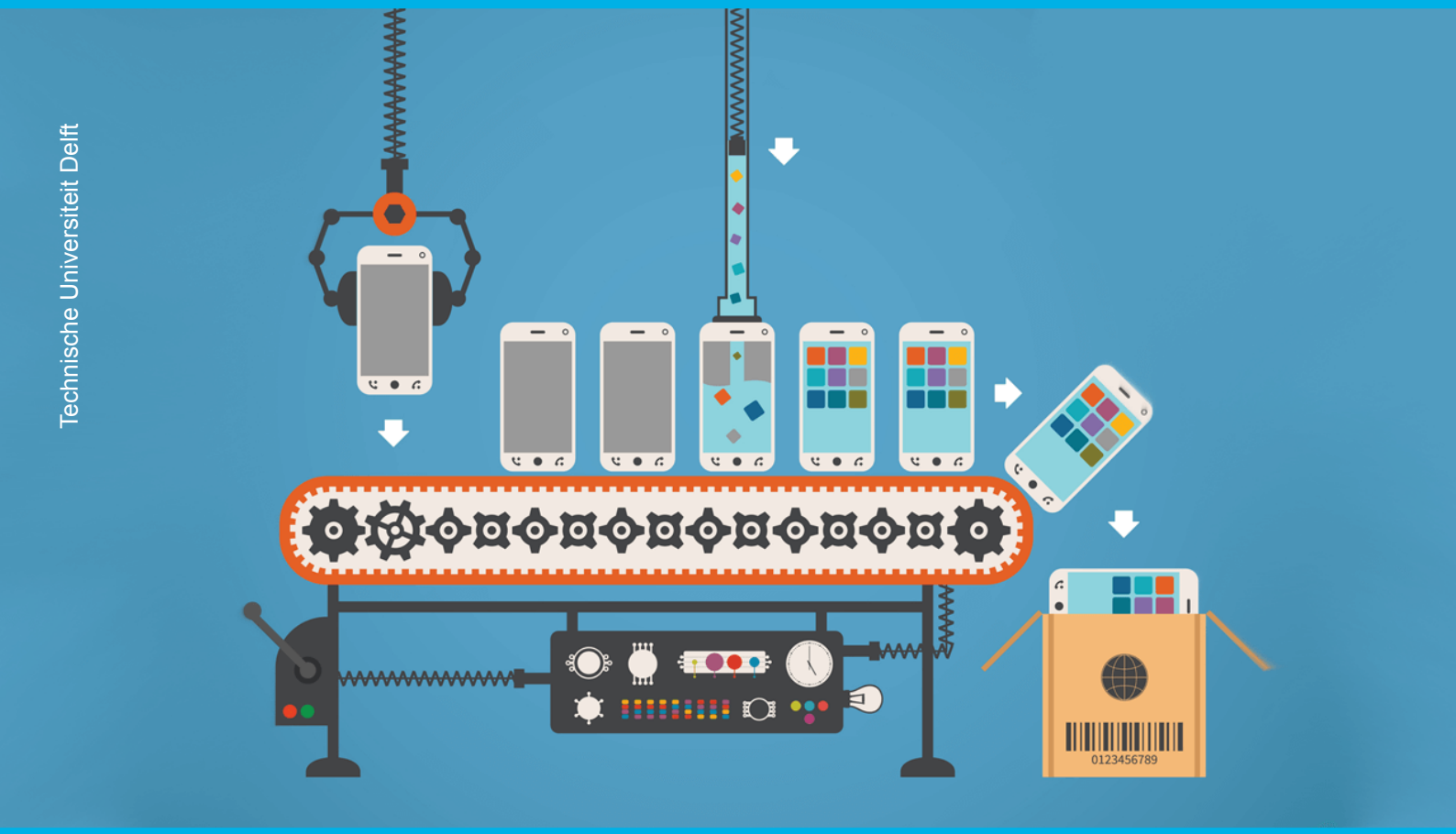
Releasing Fast and Slow

Characterizing Rapid Releases in a Large Software-Driven Organization



Elvan Kula

Technische Universiteit Delft

TU Delft

Releasing Fast and Slow

Characterizing Rapid Releases in a Large Software-Driven Organization

by

Elvan Kula

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Wednesday May 15, 2019 at 16:30 PM.

Student number:	4194217	
Project duration:	February 12, 2018 – September 1, 2018 March 1, 2019 – May 5, 2019	
Thesis committee:	Assistant Prof. Dr. Georgios Gousios,	TU Delft, supervisor
	Full Prof. Dr. Arie van Deursen,	TU Delft
	Assistant Prof. Dr. Asterios Katsifodimos,	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

The appeal of delivering new features faster has led many software projects to change their development processes towards rapid release models. Even though rapid releases are increasingly being adopted in open-source and commercial software, it is not well understood what the effects are of this practice.
This thesis presents an exploratory case study of rapid releases at ING, a large banking company that develops software solutions in-house, to characterize rapid releases. Since 2011, ING has shifted to a rapid release model. This switch has resulted in a mixed environment of 611 teams releasing relatively fast and slow. We followed a mixed-methods approach in which we conducted a survey with 461 participants and corroborated their perceptions with two years of code quality data and one year of release delay data. Our research shows that: rapid releases can be beneficial in terms of code reviewing and user-perceived quality; rapidly released software tends to have a higher code churn, higher test coverage and lower average complexity; rapid releases are perceived to be, and are in fact, more commonly delayed than their non-rapid counterparts; however, rapid releases are correlated with shorter delays (median: 6 days) than non-rapid releases (median: 16 days); challenges in rapid releases are related to managing dependencies and certain code aspects, *e.g.* design debt. Based on our findings we present challenging areas that require further attention, both in practice and in research, in order to move the practice of rapid releases forward.

Preface

I would like to thank my supervisor Georgios Gousios for his encouragement, invaluable advice and the great discussions during this project. Not only did he always make time to answer my questions and review my work, but he also encouraged me to go the extra mile and submit my first research paper to a conference. Special thanks go to Arie van Deursen and Ayushi Rastogi for their valuable input and help with the paper submission. It is wonderful working with you.

I would like to thank Joost Bosman and Hennie Huijgens for giving me the opportunity to write my thesis at ING. I am grateful that I had the chance to analyze rapid releases at such a large scale and to obtain data that is usually sealed within corporate walls. I would like to thank all my amazing colleagues at ING for making my time at the office so much fun. Many thanks go to the participants of this study who provided valuable information in their survey responses.

Lastly, I would like to thank my family and friends for all their unconditional support. I could not have done this without you. I especially want to thank you, Dad, for inspiring me to become a computer scientist. Although you are no longer with us, I felt your presence and guidance with me throughout my thesis. You have influenced me greatly as a person and as a scientist, for which I am ever thankful.

Elvan Kula
May 5, 2019

Contents

1 Introduction

In today's competitive business world, software companies must deliver new features and bug fixes *fast* to gain and sustain the interest of users [3, 79]. The appeal of delivering new features more quickly has led many software projects to change their development processes towards rapid release models [37, 49]. Instead of working for months or years on a major new release, companies reduce their cycle time (*i.e.*, time between two subsequent releases) to a couple of weeks, days or even hours to bring their latest features to customers faster. In this thesis, we use the term *rapid releases* (RRs) to refer to releases that are produced in relatively short release cycles that last a few days or weeks, rather than months or years. For example, modern applications like Google Chrome [1], Spotify [40] and the Facebook app operate with a short release cycle of 2-6 weeks, while web-based software like Netflix and the Facebook website push new updates 2-3 times a day [67]. Next to following rapid release cycles, many modern software projects support different release channels to roll out updates as soon as possible for feedback and testing [2]. The channels range from "stable" or "release", which is the most reliable channel, to the untested and unstable "Canary" (*e.g.*, Google Chrome and Facebook) or "Daily" (*e.g.*, Mozilla Firefox). Users can select their own release channel based on how quickly they want to access new features and how much instability is acceptable for them.

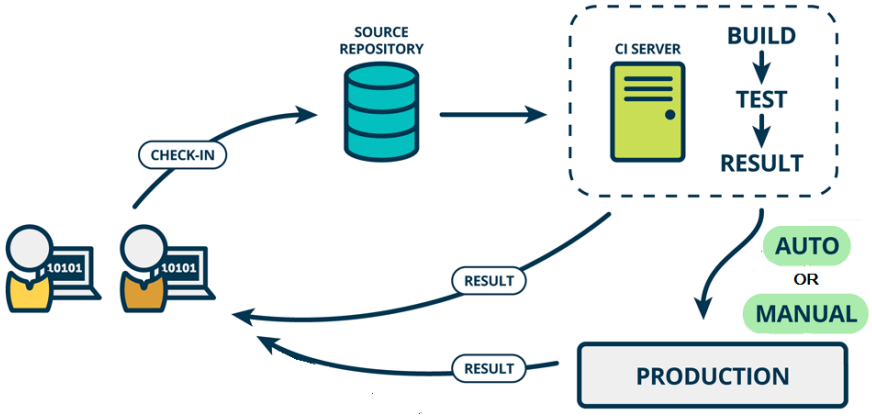


Figure 1.1: A typical software release pipeline [62]

Release engineering is a software engineering discipline concerned with the process of bringing individual code changes of developers reliably to the end user in the form of a high quality software product [33]. A typical release engineering pipeline is shown in Figure 1.1. From

start to finish, a series of steps need to be performed to release software [1]. As soon as a change is complete, developers check-in their code into the source repository to be integrated (merged) by the Continuous Integration (CI) server. The server triggers a new build and tests it, until the new changes land in the master branch. Finally, when the release date is near, a new version of the software (the release) is manually or automatically deployed to a production environment (*i.e.*, copied to a web server, virtual machine or app store) and released to end users.

As businesses have shifted towards rapid release cycles, technology has followed with new ways to bring new content faster to the end user. These efforts focus on practices, such as Continuous Delivery (CD) and DevOps, and the automation of release activities [1]. Automation of release activities has become increasingly important to organizations to achieve a repeatable and faster process of releasing software [1, 33]. Manual processes are simply too error prone and costly, especially with rapid releases. The need for automation of release activities generated a surge of industry-driven interest in the field of release engineering [1].

Most of the recent advances in release engineering, and more specifically, rapid release cycles, have been driven by industry [1]. Although industry has made claims about the benefits of rapid release cycles, these have not been empirically validated [1, 49]. Additionally, organizations indicate that software reliability and newer technologies like mobile apps in rapid releases pose challenges [1]. Even though rapid releases are increasingly being adopted in open-source and commercial software, it is not well understood what the effects are of this practice. It is vital to deepen our understanding of the practices, effectiveness, and challenges surrounding rapid releases in large organizations. Efforts to close this gap would be relevant to practitioners as well as researchers.

1.1. Rapid Releases in Literature

Despite the increased adoption, rapid releases have not been studied until recently in literature. Studies from before 2011 analyzed the benefits and challenges of adopting rapid releases. Begel and Nagappan [5] found that rapid releases are the second most mentioned benefit of agile software development at Microsoft. Rapid releases are claimed to offer a reduced time-to-market and faster user feedback [37]; releases become easier to plan due to their smaller scope [4]; end users benefit because they have faster access to functionality improvements and security updates [49]. These benefits are perceived in both open source and commercial software projects.

Despite these benefits, previous research in the context of OSS projects shows that rapid releases often come at the expense of reduced software reliability [19, 37]. Costa et al. [18, 19] showed that although bugs are fixed faster in rapidly released projects, fixed issues take longer to be integrated in rapid releases than non-rapid releases. Khomh et al. [37, 38] found that proportionally less bugs are fixed in rapid releases and the bugs that are not fixed lead to earlier crashes of the software program. Rapid releases are also perceived to result in accumulated technical debt [82] and increased time pressure [68]. While rapid releases have flourished in industry, empirical validation of their claimed advantages and disadvantages is largely missing [1, 49]. In addition, we found that all studies that have rapid release cycles as main study target are explanatory and most of them are conducted in the context of OSS projects.

1.2. Problem Statement

As rapid releases are increasingly being adopted in open-source and commercial software [37, 49], it is vital to understand their effects on the released software. This thesis attempts to provide a better understanding of rapid releases and their characteristics in industry. Because rapid releases are driven by the idea of reducing release cycle time, timing aspects are intrinsic to rapid release cycles. How often are rapid releases hampered by delays and how long are they delayed, in comparison with longer release cycles? By examining how often

and why rapid releases are delayed, we can deepen our understanding of the effectiveness of rapid releases and better determine in which cases they are appropriate to use. It is also important to explore the quality characteristics of rapid releases. Previous work shows that rapid releases can result in reduced software reliability and that they are perceived to increase technical debt. However, the impact of rapid release cycles on internal code quality has not been investigated. By exploring the quality characteristics of rapid releases, we may better understand the long-term consequences of the "deliver fast to the market" trend on software quality in the short and long run. This can help software organizations understand what compromises may be necessary in terms of quality when switching to shorter release cycles.

The overall goal of this thesis is to explore the timing and quality characteristics of rapid release cycles in an industrial setting. By exploring rapid releases in industry, we can obtain valuable insights in what the urgent problems in the field are, and what data, techniques and tools are needed to address them. It can also lead to a better understanding and generalization of release practices. Such knowledge can provide researchers with promising research directions that can help industry today.

1.3. Main Research Questions

To facilitate the characterization of rapid releases, we address in this work four research questions that focus on timing characteristics and quality characteristics. As rapid releases are driven by the idea of delivering new content faster to the end user, it is important to explore how often they are delivered on time and how this compares to non-rapid releases. Basically, are rapid releases feasible in an industrial setting? This leads to our first research question:

RQ1: ***How often do rapid and non-rapid teams release software on time?***

To understand the cases in which rapid releases are appropriate in organizations, it is essential to examine the factors that cause delay in rapid releases and how these factors compare to delay factors in non-rapid releases. Such insights could help organizations understand when and why rapid releases are feasible. We define our second research question as follows:

RQ2: ***What factors are perceived to cause delay in rapid releases?***

To understand the impact of delays in rapidly released projects, we want to find out how software development teams deal with delays in rapid releases. Can we identify differences in the ways teams manage delays in rapid releases and non-rapid releases? This leads to our third research question:

RQ3: ***How do rapid and non-rapid teams address release delays?***

Finally, we want to get a better understanding of the consequences of shorter release cycles on the quality of the released software. Such insights may inform the design of tools to help developers manage them. We only focus on internal code quality as we do not have access to data on external (user-perceived) quality at ING. We define our last research question as follows:

RQ4: ***How do rapid release cycles affect code quality?***

1.4. Approach

To address our research questions, we performed an exploratory, large-scale case study of rapid releases at ING, a Netherlands-based internationally operating bank that develops software solutions in-house. Although ING has moved to rapid release cycles since 2011, not all teams have been able to make this shift, possibly due to their application's nature or customers' high security requirements. The development context at ING is therefore mixed, consisting of teams with a release cycle duration ranging from 1 week to a few months. In other words, the teams release relatively fast and slow. We identified 433 teams out of 611

software development teams at ING as *rapid teams*, *i.e.* teams that release relatively more often than others (release cycle time ≤ 3 weeks). The remaining 178 teams with a release cycle > 3 weeks are termed as *non-rapid* teams, *i.e.* teams that release less rapidly than others. The large scale and mixed environment of ING allowed us to perform a comparative study of rapid releases and non-rapid releases (NRs) to explore how a shorter release cycle length relates to time and quality aspects of releases.

We followed a *mixed-methods approach* [17], a methodology of research that involves the systematic integration of quantitative and qualitative data. Mixed method studies originated in the social sciences and in the last decade their procedures have been refined to suit a wide variety of research fields, including software engineering [83]. The basic premise of this methodology is that such integration allows for a more complete and holistic use of data. By collecting both qualitative and quantitative data, findings can be validated through a side-by-side comparison. As part of our case study, we conducted a survey with 461 participants at ING and corroborated their perceptions with two years of code quality data and one year of release delay data. In Chapter 4, we explain our case study design and the process of data collection in more detail.

1.5. Main Contributions

To the best of our knowledge, this is the first exploratory study in the field of rapid releases. It is also the first study to analyze rapid releases at a scale of over 600 teams, contrasting them with non-rapid releases. As a first step towards a better understanding of rapid releases in industry, this thesis makes four contributions:

1. **Methodological contribution:** The first contribution is the reusable set-up of our study, as explained in Section 4. We designed an exploratory case study following a mixed-methods approach to address the timing and quality characteristics of rapid releases.
2. **Survey contribution:** We have released the iteratively-tested survey with questions for eliciting developers' perceptions on rapid releases (see Appendix A).
3. **Empirical contribution:** The third contribution is a thorough analysis of the survey responses to our research questions, as well as a statistical analysis of large-scale code quality data and release delay data collected from software development teams at ING.
4. **Conceptual contribution:** The last contribution is a discussion containing a comparison of our findings with previous literature, recommendations for practitioners and the design of release engineering pipelines. We conclude with an overview of data-driven directions for future research.

Among our findings, we identified managing dependencies and certain code aspects, *e.g.* design debt, as key challenges in practicing rapid releases. Regarding code quality, survey responses of developers show their mixed perceptions of the effect of rapid releases. On one hand, rapid releases are perceived to simplify code reviewing and to improve their focus on user-perceived quality. On the other hand, developers reported the risk of making poor implementation choices due to deadline pressure and a short-term focus. Our data analysis supports the views on code quality improvements as indicated in a higher test coverage, a lower average complexity and a lower number of coding issues. Regarding release delays, our data analysis corroborates the belief of developers that rapid releases are more often delayed than non-rapid releases. A prominent factor that is perceived to cause delay is related to dependencies, including infrastructural ones.

1.6. Guidelines for Reading

The remainder of this thesis is structured as follows. In Chapter 2 we discuss background information and related work on rapid releases. In Chapter 3 we describe the development context of ING and how the bank adopts rapid release cycles. In Chapter 4 we explain the process of collecting and processing survey responses, code quality data and release delay data. Chapter 5 presents our results on timing and quality characteristics of rapid releases derived from the data collected at ING. We discuss our main findings and integrate them back into related work in Chapter 6. Here we also consider implications for practice and future design, and threats to the validity of this study. In Chapter 7 we conclude and present directions for future work.

To readers who are interested in learning about the timing characteristics of rapid releases only, we recommend reading Sections 2.6.1, 4.4, 5.4—5.6 and 6.2. To readers who are interested in learning about the quality characteristics of rapid releases only, we recommend reading Sections 2.6.2, 4.3, 5.7 and 6.2.

2

Background and Related Work

Rapid releases are defined as *relatively short* release cycles in the order of a few days or weeks rather than months or years [49]. Literature does not provide a definition for the duration of a rapid release cycle. Considering the commonalities in existing studies, we found that all studies on rapid releases cover release cycles that are shorter than six months. Table 2.1 presents an overview of the cycle time of projects that were identified as being rapid in literature.

Despite the increased adoption, rapid releases have not been studied until recently. Studies from before 2011 focused on the motivations and practices for their adoption. Recent efforts analyze the benefits and challenges of adopting rapid releases. More specifically, they analyze the impact of rapid releases on technical debt, software reliability, testing and quality monitoring. While rapid releases have flourished in industry, empirical validation of their claimed advantages and disadvantages is largely missing in literature [1, 49]. In addition, all studies that have rapid release cycles as main study target are explanatory and largely conducted in the context of OSS projects. A majority of the studies is concerned with the open-source Mozilla Firefox project (see Table 2.1) due to the limited availability of release engineering data.

In the remainder of this chapter, we provide background information on software releases in general and an overview of the existing research on rapid releases.

2.1. Software Release Basics

In software engineering, a release is a version of a software system that is fully functional and has been made available to users for testing or public consumption [73]. A version is

Project	OSS or CSS	Study	Release Cycle Time	Regular Cycles
Mozilla Firefox	OSS	[3, 12, 18, 19, 37, 38] [10, 31, 36, 44, 49, 76]	6 weeks	Yes
Google Chrome	OSS	[3, 38, 64]	6 weeks	Yes
Eclipse	CSS	[18, 19]	6 weeks	Yes
Projects at Ericsson	CSS	[58, 81]	4-6 weeks	No
Linux Kernel	OSS	[53, 55, 64, 80]	2-3 months	No
Open Office	OSS	[53, 55]	3-6 months	No
ArgoUML	OSS	[18, 19]	5-6 months	No

Table 2.1: Overview of release cycle times of projects that are identified as being rapid in literature. The columns present the project name, indicator of open source software (OSS) or closed source software (CSS), study reference(s), release cycle time and indicator of whether the project delivers releases at regular intervals (yes/no).

an instance of a configuration item that differs from other instances of that item. A release generally refers to the final version of software and it may also be referred to as launches or increments. Each phase of software development is characterized by a different type of release, as described below [23]:

1. **Internal and external testing releases -** Prior to making a software release available for public consumption, the software should be tested to ensure that it runs as intended. To that end, most organizations use *alpha* versions, *beta* versions and *release candidates*:
   (a) *Alphas* are used for internal testing, which is done by people other than the developers, but still within the organization or community.
   (b) *Betas* are meant for external testing, which is performed by people external to the organization and community. Beta software is usually feature complete but it may have known limitations or bugs.
   (c) The *release candidate* is a version of the software that is almost ready for final release. In this phase no features are developed or enhanced, only bug fixes are allowed.
2. **External product releases** - When a software release is thoroughly tested, it is made available to customers for public consumption. Most organizations work with three levels of product releases: *major*, *minor* and *patches*. *Major* releases are releases that are produced in official release cycles and they present official versions of a software product. *Minor* releases are off-schedule releases that are produced to add functionality in a backwards-compatible manner or to provide significant enhancements (*e.g.*, performance improvements) [19]. A *patch* release corresponds to a developmental configuration that is intended to be transient. It is produced to fix specific bugs (*e.g.*, security leaks) or to add trivial functionality.

Versions are identified with a unique set of numbers or letters that update sequentially. This naming process is called *semantic versioning* [61]. Although there are no industry-wide rules for these unique identifiers, they are usually composed of the configuration item name plus a version number.

2.2. Release Channels

Many modern software projects support a number of different release channels to slowly roll out updates to users [9]. The channels range from the official, most stable version of the application (referred to as “stable” or “release”) to the completely untested and likely least stable “Canary release”). The pre-release channels are intended for users who want to experience the newest features and improvements. These users give developers time to test their code on upcoming versions before they are deployed to end users. This reduces the risk of issues impacting the entire user base. The most common types of release channels are presented below in decreasing order of stability [2]:

- **Stable (or: Release) channel:** This channel’s releases have been fully tested and are the most reliable. Most users use this channel for production use. Stable releases follow the official release schedule for major version releases.
- **Beta channel:** This channel is a bit ahead of the stable/release channel. As explained in the previous section, Beta releases contain new features that a team has tested that will eventually land in the stable channel. Beta releases might still have some bugs and performance issues. If a user is interested in seeing what’s next with minimal risk, then the Beta channel is the place to be.
- **Dev channel:** This channel’s releases are selected from older canary releases that have been tested for a while. Just like the canary channel, this channel is used to test and show people the newest features as soon as possible. It is still unstable and subject to bugs. Updates are usually released weekly or monthly.

- **Canary (or: Daily) channel:** This channel is cutting edge as it receives daily updates of new features as soon as they are built. It closely follows the development repository. Since features are added even before being tested, this channel is prone to issues and errors.

2.3. Release Management

There are different strategies for deciding when to release a new version of a software system, and different projects may apply different techniques. Release strategies can be classified as *feature-based*, *time-based* or a combination of both [55]. Feature-based releases are issued when a set of predefined goals have been fulfilled, *e.g.*, when a set of features have been implemented or a set of reported bugs have been fixed. Time-based releases are issued according to a predefined schedule (*i.e.*, a roadmap). They are in advance planned to be released on a specific date, and the preceding development effort is aimed at producing a release on the predefined date. An important concept within the time-based release strategy is *regular cycles*, where releases are delivered at regular intervals. Another concept is *continuous flow*, where releases are delivered on a demand basis in series or patterns with the aim of achieving continuity.

Once the release strategy is chosen, it is time to determine the release interval. In literature this is also termed as the cycle time. It is defined as the time between the start of the development of a feature and its actual release (when it goes to production). Michlmayr et al. [55] identified five factors that are perceived to affect the choice of release interval: regularity and predictability; nature of the project and its users; commercial factors; cost and effort; and network effects (the need to synchronize the release schedule with the schedules of vendors and dependent projects).

2.4. Motivations for Adopting Rapid Releases

An early study by Begel and Nagappan [5] found that rapid releases are the second most mentioned benefit of agile software development at Microsoft. Rapid releases were practiced by 2/3 of the respondents as a consequence of Continuous Integration (CI) at the time. Respondents reported that the main motivations towards the adoption of shorter release cycles relate to: *easier planning* due to a smaller scope, *easier monitoring of software quality* and *more rapid feedback* regarding new features and bug fixes. Other studies found similar benefits [31, 32, 45, 49, 50, 55, 58] in open source and commercial software projects.

Other motivations reported by studies relate to *customer needs* [11, 44, 75]. As a result of a shorter time-to-market, projects that adopt rapid releases are able to compete better and respond faster to customers' changing needs. Kong et al. [42] found that rapid releases are perceived to be the best practice for achieving customer satisfaction and being responsive to customer needs. Da Costa et al. [18] showed that end users and developers may become frustrated when a reported issue is not integrated in an upcoming release. For example, in the market of web browsers, previous studies reported that the switch of many browsers to shorter cycles is driven by the need to meet the changing demands of users, in line with the fast evolution of web standards. Khomh et al. [38] revealed that the decision of Mozilla Firefox to move to rapid releases was driven by the competition with Google Chrome, which was already using it. The fast expansion of mobile platforms resulted in an increased competition between applications to provide a greater ease of access [76]. This brought a big incentive to rapid releases to stay competitive. Hemmati et al. [31] showed that Firefox was already losing market share to Google Chrome when it decided to adopt rapid releases.

In the OSS context, Schach et al. [80] and Fitzgerald et al. [53] found that another incentive for implementing rapid releases is to *attract collaborators*. Both studies show that a reduced release cycle time can increase the number of participants where the majority are volunteers motivated by the challenges associated with shorter release activities.

2.5. Practices for Adopting Rapid Releases

The *time-based release strategy* is the main strategy adopted by practitioners that implement rapid releases [54][55]. Michlmayr et al. [54] found that time-based releases are preferred in a rapid development environment as they are perceived to result in a better internal planning. However, the authors point out that a release process needs to be automated and optimized (reproducible) in order to afford rapid release cycles [36]. Therefore, rapid release cycles are often adopted through modern release engineering practices aiming to accelerate delivery and implement improvements continuously, such as *continuous integration*, *continuous delivery* and *continuous deployment* [33].

Continuous Integration. In traditional software development, the process of integrating code happened at the end of the development process, after each person had completed their work. Integration generally took weeks or months. *Continuous integration* (CI) is a practice that puts the integration phase earlier in the development cycle so that code is built, tested and integrated more regularly [33]. It refers to the process of continuously checking new code revisions out on dedicated build machines (usually a CI server), compiling them and running a set of tests to check for regressions [1]. CI makes integration a regular event in the order of minutes. A build that passes the tests after integration is called a *green build*. This indicates that the code changes are successfully integrated together and that the code is working as expected by the tests. However, the integrated code is not yet ready to go to production as it has not been tested in a production-like environment.

Continuous Delivery. Continuous Delivery (CD) goes one step further to automate a software release, which involves deploying the code to an environment that is similar to production (after the CI tests pass). This process of deploying to (and testing on) different environments is called a *deployment pipeline* [33]. Such a deployment pipeline makes sure that the software is always ready to go to production. Often the pipeline consists of a development environment, a test environment and a staging environment in which the code is tested differently [62, 77]. Usually, the code is only advanced to the next environment in the deployment pipeline if it passes the tests of the previous environment [25]. This way developers receive new feedback from the tests in each environment, making it easier to understand where an issue might be. If a build passes the tests on all environments, the code is likely to work on the production environment. Finally, it is up to the team to manually deploy a code change to the final production environment.

Continuous Deployment. Continuous deployment is a process that automatically deploys every code change to the final production environment [77]. With Continuous Deployment, companies can push multiple releases into production every day. However, a change might not yet be released as it can be hidden using feature toggles [43]. This practice requires the ability to roll out changes to a subset of servers or customers and to roll back changes quickly in case of a failure.

2.6. Impact of Rapid Releases

Comparing studies on the impact of rapid releases, we distinguished two main directions that previous work has focused on: the impact of switching to rapid releases on (1) *time aspects* and (2) *quality aspects* of the development process. This section presents an overview of existing research on the impact of rapid releases based on our distinction between time- and quality-related aspects.

2.6.1. Time Aspects of Rapid Releases

Recent research efforts have examined the impact of rapid release cycles on the integration delay of fixed issues, factors impacting release cycle time and time pressure experienced by developers.

Integration delay of fixed issues. Although rapid release cycles are claimed to deliver fixed issues to users more quickly, the work of Costa et al. [19] shows that rapidly releasing projects introduce delay in the integration of fixed issues. They studied the integration of addressed issues from the rapidly releasing Firefox project and the non-rapidly releasing ArgoUML and Eclipse projects. It is shown that the time interval between releases in rapid and non-rapid strategies is roughly the same when considering both major and minor releases, with medians of 40 and 42 days, respectively. The authors found that 98% of fixed issues in rapid Firefox releases had their integration delayed by at least one release. In comparison, 34% to 60% of fixed issues in non-rapid releases were delayed by at least one release. To better understand the impact that rapid release cycles have on the integration of fixed issues, the authors performed a comparative study of 72,114 issue reports from the Firefox project before and after its shift to a rapid release cycle. They found that issues are fixed faster in rapid releases but, surprisingly, rapid releases take a median of 54% (57 days) longer than non-rapid releases to deliver fixed issues. The authors note that this may be due to the fact that non-rapid releases prioritize the integration of backlog issues, while rapid releases prioritize issues that were addressed during the current cycle [18].

Factors impacting release cycle time. Kerzazi and Khomh [36] analyzed over 14 months of release data from 246 rapid Firefox releases to determine the factors that affect cycle time. Their analysis shows that testing, both manual and automated, is the most time consuming activity in the release process. As functional dependencies are often poorly defined, the release team of Firefox usually runs all regression test cases for every change in the source code. If the team would have a better insight into the functional dependencies, then they would be able to run just a subset of the test cases and significantly reduce the testing time. The authors found that code merging and packaging activities account for only 6% and 2% of the cycle time, respectively. Furthermore, the authors note that a lack of socio-technical congruence among teams can delay releases. Socio-technical congruence refers to the alignment between the technical dimension of work and the social relationship between team members. It is especially important in the context of parallel development as that of Firefox.

Increased time pressure. The strict release dates in rapid releases are claimed to increase the time pressure under which developers work. Time pressure has been linked to burnout in software development and increased technical debt [74]. Rubin and Rinard [68] conducted a study at 22 high-tech companies and found that most developers work under significant pressure to deliver new functionality quickly. A majority of the study participants observed that their fast-paced development environment prioritizes development speed over software quality. At the same time, participants believed that this pressure often comes at the expense of reduced quality and accumulated technical debt. Furthermore, Mäntylä et al. [49] looked into the impact of rapid releases on time pressure in software testing. They found that testing in rapid releases becomes more deadline-oriented and is therefore performed under greater time pressure than in non-rapid releases. Related to this work, Claes et al. [10] investigated the effect of Firefox's move to rapid releases on the work pattern of developers. Surprisingly, they found that switching to rapid releases has reduced weekend work and working during the night.

2.6.2. Quality Aspects of Rapid Releases

Recent research efforts have examined the impact of a reduced release cycle length on technical debt, software reliability, testing and quality monitoring.

Technical debt. Tufano et al. [82] found that pressure to deliver features for an approaching release date is one of the main causes for code smell introduction. The authors studied the change history of 200 open source projects belonging to the ecosystems of Android, Apache and Eclipse. They discovered that smells are generally introduced the last month before issuing a major release. The findings of Tufano et al. suggest that the frequent release dates in a rapid development environment may result in an increase in code smells. Tufano et al.

also found that there is a peek in the development activity (workload) during the last week before a release date. This suggests that most code changes are made at the last moment, which is associated with a higher risk of poor code quality.

Codabux and Williams [14] performed a case study of technical debt within a large software company with 250 developers. The authors found that developers observe an increase in technical debt in their projects stemming from the speed and lack of discipline in their rapid development environment. Developers reported to cut corners in rapid releases due to a lack of time for activities such as refactoring. Moreover, Torkar et al. [81] performed a case study of rapid projects at Ericsson and the open source Linux kernel (rapid), FreeBSD (non-rapid) and JBoss (non-rapid) projects. The developers perceived that the strict deadlines in rapid projects increase technical debt. To control for technical debt, the Linux and FreeBSD projects do not set strict deadlines for tasks although they have a regular release schedule. Moreover, they allocate time for fixing debt outside of their release schedule through a separate backlog for refactoring tasks and a pre-commit review phase.

Software reliability & testing. In the OSS context, rapid releases are found to have a significant impact on software reliability as well as the testing process. An early study by Baysal et al. [3] compared the release and bug strategies of the (at that time) non-rapidly released Mozilla Firefox and rapidly released Google Chrome based on browser usage data from web logs. They found that bugs were fixed 16 days faster in versions of non-rapid Firefox releases than in rapid Chrome releases, but this difference was not statistically significant. Later, Khomh et al. [37, 38] studied the impact of Firefox's transition to a rapid release model on software reliability. In contrast with Baysal et al.'s findings, they found that bugs were fixed significantly faster in rapid versions, however, there was no significant difference in the number of post-release bugs. They also found that less bugs were fixed in rapid releases, proportionally, and the bugs that were not fixed led to crashes earlier during software execution.

Mäntylä et al. [49] performed a case study on Firefox and analyzed the results of six years of test execution runs, including 5 major non-rapid and 9 major rapid releases. They showed that in rapid releases testing has a narrower scope that enables a deeper investigation of features and regressions with the highest risk. This was also found by other studies [45, 58]. Moreover, Mäntylä et al. showed that rapid releases lead to a more continuous testing process with proportionally smaller spikes before the main release. However, the limited time frame of rapid releases decreases test suite diversity and make testing more deadline oriented. Also, Firefox's transition to a rapid release model reduced the community participation in manual testing and made it harder to develop a larger testing community. To keep up with rapid releases, they had to increase the number of specialized testing resources through contractors.

Petersen et al. [58] reported improvements of test coverage at unit-test level. Porter et al. [60] found that testers of rapid releases lack time to test all possible configurations and environments of the software product. Earlier research [58] discovered that test cycles in rapid releases are often too short to perform time-intensive tests, such as performance testing.

Quality monitoring. In the context of OSS, multiple studies [31, 45, 49, 55] have shown that rapid releases ease the monitoring of quality and increase the incentive of developers to deliver quality software. Mäntylä et al. [49] reported that shorter release cycles driven by testing allows for greater and quicker feedback in each release, thereby improving the focus of developers and testers on quality. Hemmati et al. [31] highlighted the ease of monitoring of progress and quality, as tests become more focused and run more frequently. In addition, Michlmayr et al. [53] note that the quicker feedback in rapid releases provides more information on what parts of the software need more attention quality wise.

3

Context of ING

This case study is performed at ING, a large multinational financial organization with about 54,000 employees (of which 15,000 are software engineers) and over 37 million customers in more than 40 countries [34]. Currently, ING has 611 globally distributed software development teams that work on more than 750 internal and external applications. Around 350 software development teams are located at ING Netherlands (ING NL), the global headquarters of the bank. The remaining teams are spread over 14 countries, including the United Kingdom, Germany, Romania, Poland and Spain. In this chapter, we present the development context of ING and discuss how the bank has switched to rapid release cycles.

3.1. ING's Switch to Rapid Release Cycles

In 2011, ING decided to shorten their development cycles when they planned to introduce *Mijn ING*, a personalized mobile application for online banking. Before 2011, teams worked with release cycles of two to three months between major version releases. However, ING wanted to cut the cycles between major version releases down to three weeks or less to stay ahead of the competition. Since then the bank has moved to a rapid development model, in which all teams work with a time-based release strategy where releases are planned for a specific date. In general, the teams deliver releases at regular week intervals, however, the release cycle length differs across teams and occasionally within a team. The latter can appear in case of a release delay. Section 5.1 presents an overview of the teams' release frequencies at ING.

To be more responsive to changing customer demands and to make shorter release cycles practical, the bank has gone through an agile transformation based on three methodologies: the *Spotify Squad Framework*, *DevOps* and *Continuous Delivery*. In the following sections, we will discuss the implementation of these methods in more detail.

3.2. Spotify Squad Framework

Since 2015 ING has reinvented their organization from the ground up, moving from a traditional organizational model featuring functional departments to a completely agile model. They implemented Spotify's '*Squads, Tribes, Chapters*' model to organize ING staff in 611 squads and 14 tribes [41]. This approach has improved ING's time-to-market and increased employee engagement. Figure 3.1 shows an overview of the Spotify model.

At ING, *Squads* are small multidisciplinary, self-organized teams that consist of 5 to 9 people and operate with a high degree of autonomy. They have end-to-end responsibility for a specific client-related mission. Each squad has autonomy to decide what to build and how to build it [25]. However, their work needs to stay aligned with their mission. A *product owner*

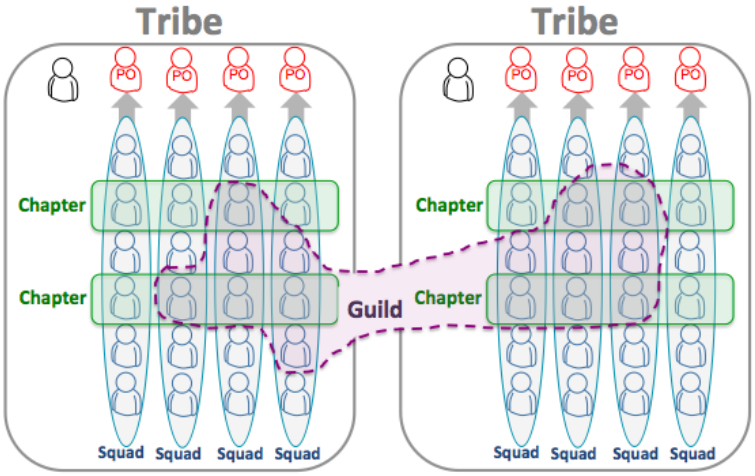


Figure 3.1: Spotify Squad Framework [25]

is responsible for coordinating the activities of squads to keep them aligned. The product owner is seen as a squad member and does not necessarily have a leading role.

Squads that have interconnected objectives are grouped together into *Tribes*. Every tribe has a tribe lead that is responsible for coordinating the activities and priorities with other tribes. Each tribe also has an Agile coach to support squads by identifying and resolving impediments.

At the horizontal level of the functional organization, squad members with the same competences (*e.g.*, web development or quality assurance) are formed into cross-squad *Chapters*. Each chapter is part of a tribe and is supported by a chapter lead that supports them in specific challenges. Similarly, there are *Guilds* that span across different tribes, including people who have a common interest. Guilds often consist of all chapters working in a certain area. The purpose of a chapter and a guild are the same: ensuring transparency by sharing knowledge and keeping the teams aligned [25]. For example, there is a tester from squad A who is struggling with a problem that a tester from Squad B has already solved. If they both are in the same chapter, they can share their problem and help each other out. Otherwise, there would be a guild of testers from all testing-related chapters where they can share their knowledge on the problem.

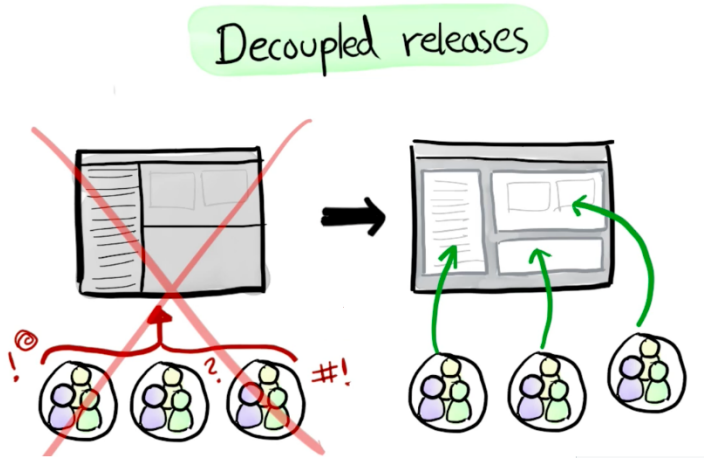


Figure 3.2: Decoupled Releases in Spotify's *Squads, Tribes, Chapters* Model [25]

How does the Spotify model enable faster software delivery? The model aims at building faster decision-making systems and reducing synchronization bottlenecks with multidisciplinary, self-organized teams [25]. This lets teams work faster in a more collaborative manner and simplifies the release process, enabling more frequent releases. In addition, squads use a decoupled architecture where different components/layers of a system do not depend tightly on each other but rather interact with each other using well-defined interfaces. This enables decoupled releases in which each squad is able to release their own code directly into the part of the system they have end-to-end responsibility for.

3.3. DevOps/ BizDevOps

At the beginning of 2011, ING introduced DevOps teams to get the developers and operators to collaborate in a more streamlined manner. DevOps is defined as "a set of practices that are trying to bridge the developer-operations gap at the core of things and at the same time covers all aspects which help in speedy, optimized and high-quality software deliver" [56]. DevOps teams take charge of the application over its whole lifetime (*i.e.*, during development and operations). This means that developers and operators act as part of a common product or service team who are all responsible for the development, support and maintenance of the product [22]. Figure 3.3 presents the old way of working in a traditional IT environment and the new way doing DevOps. The multifunctional, autonomous teams in DevOps allow for a faster and more reliable delivery of software with less conflicts of competence within software development processes.

However, due to the increasing business responsibility of IT, ING is currently undergoing a transition from DevOps towards BizDevOps. BizDevOps is a new concept were Business, Development and Operations work together in software development and operations [29]. This creates a consistent responsibility from business over development to operations, which enables teams to develop software more quickly, be more responsive to customers' needs and ultimately maximize revenue.

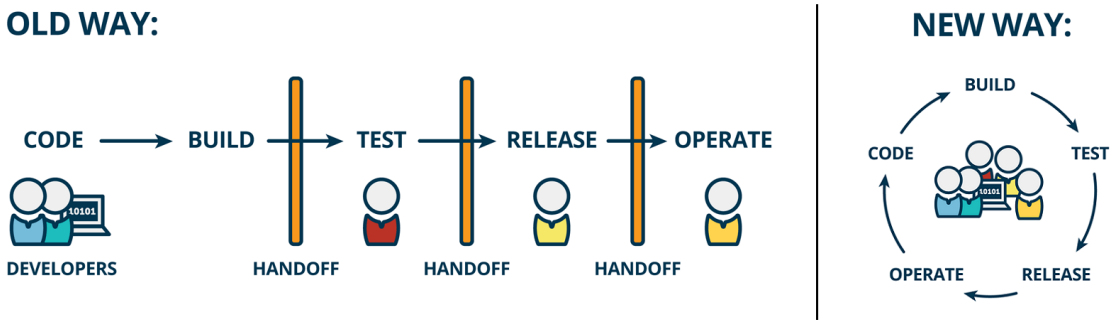


Figure 3.3: Distinction between traditional IT Operations (old way) and DevOps (new way) [62]

3.4. Continuous Delivery Pipeline

In 2015 ING introduced the *Continuous Delivery as a Service* (CDaaS) project to implement a fully automated release engineering pipeline for its software engineering activities. ING put a continuous delivery pipeline in place to enforce an agile development process and to reduce the testing and deployment effort of the DevOps teams. Because these activities used to be primarily manual work, the CDaaS pipeline makes it easier for teams to build software rapidly. The mindset behind CDaaS is to go to production as fast as possible, while maintaining or improving quality, so teams get fast feedback and know they are on the right track.

Fig 3.4 depicts the pipeline. This pipeline facilitates all 611 teams. The teams push more than 2500 deployments to production each month on over 750 different applications. The pipeline is based on a model described by Humble and Farley [33]. Within CDaaS, ING created two versions of the pipeline for their main technology platforms, Windows and Linux. The pipeline contains several specialized tools for every step in the software engineering cycle, such as ServiceNow (backlog management), Gitlab (code), Jenkins (build), SonarQube (code inspection), Fortify (security), Artifactory (artifact repository), and Nolio (deploy).

Every time a developer pushes a commit, this is detected by *Jenkins*, the CI server, which is responsible for monitoring the quality of the source code and for building the solution. Builds at ING are found to be broken for various reasons, such as compiling errors, failing test cases and software quality problems detected by Automated Static Analysis Tools (ASATs). At ING, the ASATs of choice are *SonarQube* and *Fortify*.[1,2] When a build succeeds, the artifacts are stored in the repository using *Artifactory*. At this stage, the application is ready to be deployed on different environments, such as TST (testing), ACC (acceptance) and PRD (production). The pipeline has an automated acceptance mechanism that checks whether a release meets ING's acceptance criteria, before deploying it on the ACC or PRD environment.

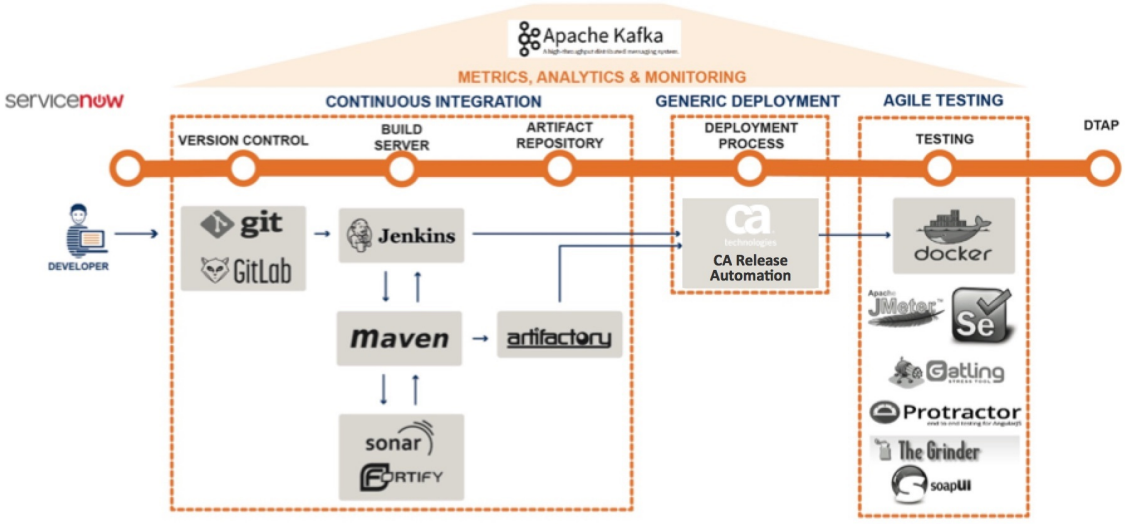


Figure 3.4: Continuous delivery pipeline at ING

[1] https://www.sonarqube.org/
[2] https://www.microfocus.com/en-us/products/static-code-analysis-sast/overview

4

Research Method

The overall goal of our study is *to explore the timing and quality characteristics of rapid release cycles in an industrial setting*. To that end, we conducted an exploratory case study at ING following a mixed-methods approach. We conducted a survey that was answered by 461 participants and corroborated their perceptions with code quality data and release delay data from squads at ING. In the following sections, we explain the process of designing and processing the survey, along with how the quantitative data was collected and analyzed.

4.1. Study Design

We conducted an *exploratory* case study [70] of rapid releases at ING. Following the guidelines of Runeson & Host [70], a case study can be defined by its case, units of analysis and research perspective (positivist, interpretive or critical). Our study can be described as a *single case design* as it involves a single organization, ING. Secondly, our study considers two units of analysis: the group of rapid squads and the group of non-rapid squads. This makes it an *embedded* case study (a study is embedded when multiple units of analysis are studied within a case). Regarding the research perspective, our case study combines characteristics from interpretive and positivist type case studies. Klein and Myers [39] define an interpretive case study to attempt to understand phenomena through the participants' interpretation of their context. Our study attempts to understand how rapid releases work in a large software-driven organization through participants' interpretation of the development context at ING. A positivist case study is defined to search evidence for formal propositions, measure variables, test hypotheses and draw inferences from a sample to a stated population. Although our study does not attempt to formulate formal propositions, we do attempt to draw inferences from a large sample of participants from ING that practice rapid releases.

Data triangulation. To get a better understanding of rapid releases, we addressed the research questions applying data triangulation [71]. Data triangulation is the use of more than one data source or collecting the same data at different occasions. In our case study, we combined three types of data sources to present detailed insights: (1) qualitative survey data, (2) quantitative data on release delays and (3) quantitative data on code quality. This is also known as a *mixed-methods* approach [17, 66] (*i.e.*, the integration of qualitative and quantitative data collection and analysis methods within a study). Such integration allows for a more complete and harmonious utilization of data than if you would collect quantitative or qualitative data only. In our case, it enabled us to capture multiple perspectives with added details on the phenomenon of rapid release. Triangulation also facilitates validation of data through cross verification. In our case, we compared data collected through qualitative methods with quantitative data to test the consistency of findings obtained though developer perceptions.

Since we wanted to learn from a large number of software engineers and diverse projects, we collected qualitative data using an online survey in two phases [26]. In the first phase, we ran a pilot study with two rapid squads and two non-rapid squads at ING. This allowed us to refine the survey questions and to identify emerging themes we could explore further (*i.e.*, we added a question about additional tools used for assessing code quality, and we provided definitions for different types of debt). In the second phase, we sent the final survey to all squads at ING NL. In addition, we analyzed quantitative data stored in ServiceNow[1] and SonarQube to examine the timing and quality characteristics of rapid releases, respectively. We compared the perceptions of developers with release delay data and code quality data for rapid and non-rapid squads. An overview of our study set-up is shown in Figure 4.1. For RQ2, "What factors are perceived to cause delay in rapid releases?", we only analyzed survey data because quantitative (proxy) data on release delay factors is not being collected by ING. The same holds for RQ3, "How do rapid and non-rapid teams address release delays?". There is no quantitative data being collected on the methods used by teams to address release delays.

In general, the quantitative data on release delays and code quality was aggregated at release level, while developers were asked to reflect on team performance in the survey. To be consistent in aggregation, we used the same rapid/non-rapid classification threshold of 3 weeks for both teams and releases. The threshold and aggregation levels are further explained in Section 5.1.

4.2. Developers' Perceptions Collection

We sent the survey to members of both rapid squads and non-rapid squads. To ensure that we collected data from a large number of diverse projects, we selected the members of all squads at ING NL as candidate participants. In total, we contacted 1803 participants belonging to more than 350 squads, each working on their own application that is internal or external to ING. The participants have been contacted through projects' mailing lists.

Survey design. The survey was organized into five sections for research related questions, plus a section aimed at gathering demographic information of the respondents (*i.e.*, role within the squad, total years of work experience and total years in ING). Table 4.1 provides an overview of the questions we asked participants. Appendix A presents a more extensive list of the survey questions and possible answer choices. The five sections were composed of open-ended questions mixed with multiple choice or Likert scale questions. We provided respondents with a mandatory multiple choice question about their squad's release frequency to determine if they are part of a rapid or non-rapid squad.

To address RQ1, "How often do rapid and non-rapid teams release software on time?", we asked respondents to fill in a compulsory multiple choice question on how often their squad releases on time. For RQ3, "How do rapid and non-rapid teams address release delays?", we included a mandatory open-ended question asking respondents what their squads do when a release is delayed. For RQ2, "What factors are perceived to cause delay in rapid releases?", we provided respondents with an open-ended question to gather unbounded and detailed responses on delay factors. For RQ4, "How do rapid release cycles affect code quality?", we provided respondents with a set of two compulsory open-ended questions, asking respondents how they perceive the impact of rapid release cycles on their project's internal code quality, and whether they think that rapid release cycles result in accumulated technical debt. We also added a set of four optional 4-level Likert scale questions (each addressing the impact of rapid releases on testing debt, design debt, documentation debt and coding debt). The types of debt were defined in the survey as presented in Table 4.1 (Q15 - Q18). To further explore the opinions of respondents, we included an optional 'other' response for all questions that had predefined answers.

[1] https://www.servicenow.com/

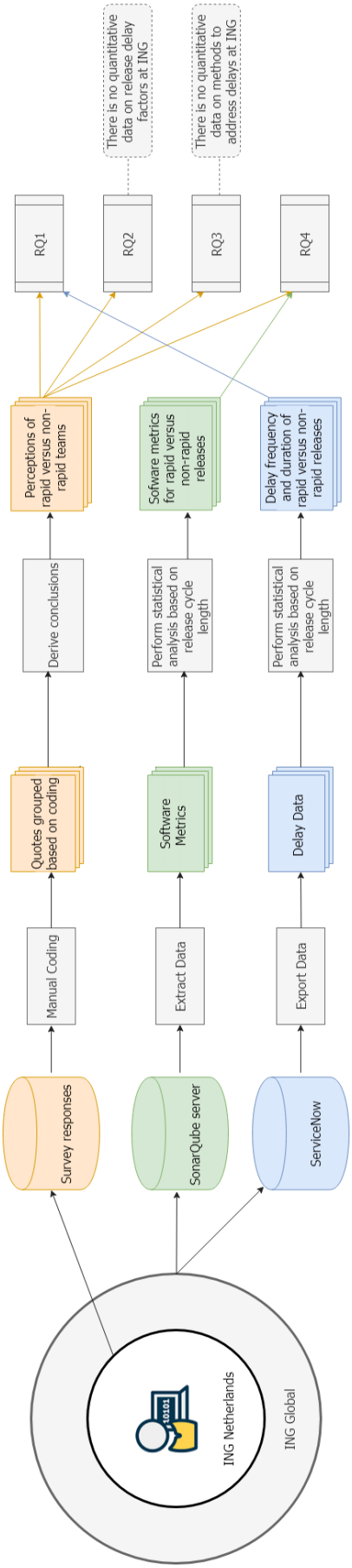


Figure 4.1: Overview of our mixed-methods approach to study the timing and quality characteristics of rapid releases at ING.

Section	#	Question	Type (O/M/L)	Optional/ Mandatory	# Resp. (Rapid)	# Resp. (Non-Rapid)
Demographics	1	What is your squad's name?	O	Optional	221	111
	2	How many years of work experience do you have in the software industry?	M	Mandatory	296	165
	3	Which of the following best describes your role at ING?	M	Mandatory	296	165
	4	How long have you been working at ING?	M	Mandatory	296	165
Timing	5	My squad releases every ... weeks/ months	M	Mandatory	296	165
	6	How often are your squad's releases on track?	M	Mandatory	296	165
Delays	7	In your experience, what is the most common reason for release delays?	O	Mandatory	296	165
	8	What do you, as a squad, do when a release is delayed?	O	Mandatory	296	165
Quality monitoring	9	Apart from SonarQube and Fortify, do you use any other static code analysis tools to assess your code quality?	M	Optional	245	129
	10	(Follow-up on Q9) What other tools do you use?	O	Optional	68	26
	11	Does your squad actively monitor the technical debt in your releases?	M	Mandatory	296	165
Code quality	12	In your experience, how do rapid release cycles affect your project's source code (technical) quality?	O	Mandatory	296	165
	13	"Rapid release cycles result in accumulated technical debt." Do you agree or not? Please explain your answer.	O	Mandatory	296	165
Technical debt	14	"*Rapid release cycles result in accumulated design debt.*" Design debt is caused by poor design and architecture, e.g. a non-flexible architecture that makes it hard to customize and maintain the system.	L	Optional	161	62
	15	"*Rapid release cycles result in accumulated documentation debt.*" Documentation debt is caused by a lack of documentation for code logic or by poor documentation quality.	L	Optional	160	62
	16	"*Rapid release cycles result in accumulated testing debt.*" Testing debt is caused by a lack of testing or by poor testing quality.	L	Optional	157	61
	17	"*Rapid release cycles result in accumulated coding debt.*" Coding debt, or programming debt, is caused by low code quality.	L	Optional	155	61
Related work	18	"*I often experience time pressure to deliver fast to the market.*"	L	Optional	170	93
	19	"*I feel that in my environment development speed is prioritized over software quality.*"	L	Optional	169	93
	20	"*Rapid release cycles prioritize the integration of issues addressed during the current cycle.*"	L	Optional	163	90
	21	"*In my experience, bugs are fixed faster with rapid release cycles.*"	L	Optional	168	93
	22	"*Rapid releases allow for less time for refactoring activities.*"	L	Optional	296	91
	23	"*Rapid release cycles result in more focused testing and a less diverse test suite.*"	L	Optional	168	90

Table 4.1: Survey questions (O/M/L stands for open-ended, multiple choice or Likert scale question).

To assess the perceived validity of related work in the development environment of ING, we provided respondents with a set of six optional 4-level Likert scale questions (each addressing a finding from previous research as discussed in Section 2).

In addition, we provided respondents with a few optional questions on the way they perform quality monitoring. To check if the quality assessment of non-rapid squads and rapid squads is comparable, we provided respondents with two multiple-choice questions asking whether they actively monitor technical debt and which tools they use besides SonarQube and Fortify.

Survey operation. The survey was uploaded onto *Collector*, a survey management platform internal to ING NL. The candidate participants were invited using an invitation mail featuring the purpose of the survey and how its results can enable us to gain new knowledge of rapid releases. For the pilot run, we randomly selected two rapid squads and two non-rapid squads. We e-mailed the 24 employees in the four squads and received seven responses (29% response rate). For the final survey, we e-mailed 1803 squad members and obtained 461 responses (26% response rate). This is above average for online surveys in software engineering (*i.e.*, the typical response rate is usually within the 14—20% range [63]). 296 of our respondents are part of a rapid squad (64%) and the remaining 165 respondents work in non-rapid squads (36%). Table 4.1 presents an overview of the number of responses we received for each survey question (the optional questions were completed by less than 461 respondents). Respondents had a total of three weeks to participate in the survey. We sent two reminders to those who did not participate yet at the beginning of the second and third week. The survey ran from June 19 to July 10, 2018.

Survey analysis. For the multiple choice and Likert scale questions we used R,[2] a language and environment for statistical computing, to display statistical information and create plots. We filtered the responses based on release frequency (rapid or not-rapid) or based on answers to selected questions.

For the open-ended questions, manual analysis was needed. There were four questions in the survey that were completely open and ten questions that were closed but contained an open-ended 'other' or 'Please explain your answer' response. We used *manual coding* [16] to summarize the results of the open-ended questions. Figure 4.2 presents an overview of the coding process. Manual coding is an inductive method where the main themes of each question are identified as they emerge when processing the answers. Then a code is assigned to each identified category and each response is labeled with at least one code.

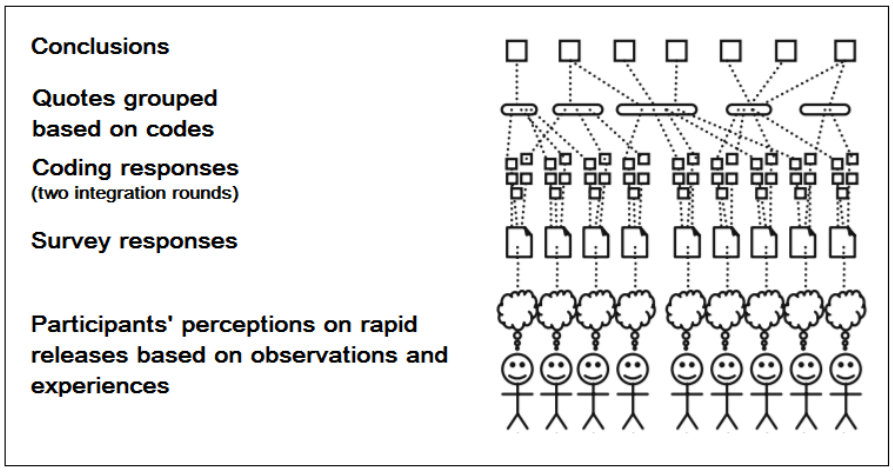


Figure 4.2: Overview of the coding process

We coded by statement and codes continued to emerge till the end of the process. When needed, the coding was conducted on different levels of detail, where broad categories were

[2]https://www.r-project.org/

specialized and detailed ones were generalized. In cases where answers described more than one item, we assigned up to three codes to prevent information loss. We applied manual coding in two integration rounds:

- **Integration round 1:** in the first round, the author and thesis supervisor used an online spreadsheet to code a 10% sample (40 mutually exclusive responses) each. They assigned at least one and up to three codes to each response. Next, they met in person to integrate the obtained codes, meaning that the codes were combined by merging similar ones, and generalizing or specializing the codes if needed. When new codes emerged, they were integrated in the set of codes. The author then applied the integrated codes to 90% of the answers and the supervisor did this for the remaining 10% of the responses.
- **Integration round 2:** in the second round, the author and thesis supervisor had another integration meeting which resulted into the final set of codes. The final set contained 18% more codes than the set resulting from the first integration round. The author then applied the final set of codes to all responses.

4.3. Software Metrics Collection

To analyze the quality of software written by non-rapid squads in comparison with rapid squads, we extracted the SonarQube measurements of releases that were shipped by 190 teams that actively use SonarQube as part of the CDaaS pipeline. Although all 611 teams at ING have access to SonarQube, not all of them run the tool each time they ship a new release. In such a case that a team does not use SonarQube on a regular basis, their measurements do not provide an accurate representation of their actual release cycle time, and, therefore, they are not suitable for our quantitative analysis. We analyzed releases shipped in the period from July 01, 2016 to July 01, 2018. In total, we studied the major releases of 3048 software projects. 67% of these releases were developed following a rapid release cycle (≤ 3 weeks) with a median value of 2 weeks between the major releases. The remaining 33% of the releases were developed following a non-rapid release cycle (> 3 weeks) with a median value of 6 weeks between the major releases.

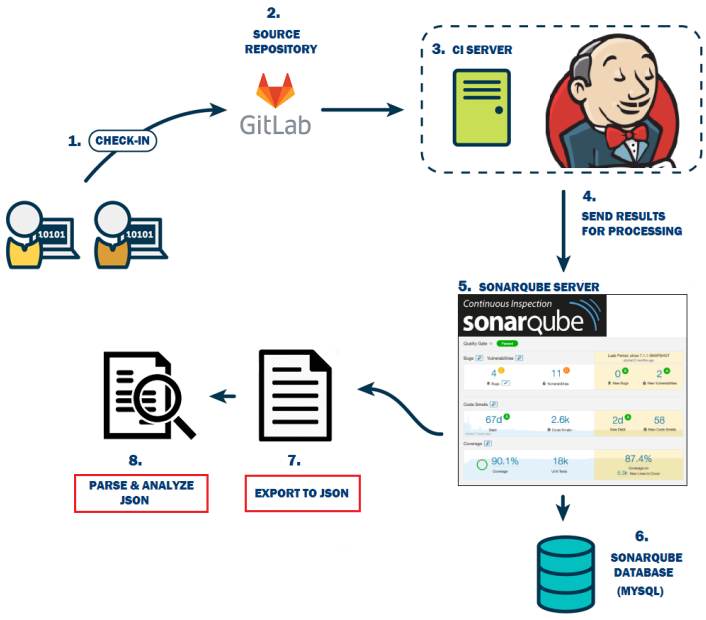


Figure 4.3: An overview of how SonarQube is integrated into the development process at ING. Steps 7 and 8 are our own contributions. As part of step 7, we wrote a Python script to export the code quality results for our analysis.

SonarQube data export. When developers at ING push their code into Gitlab, *Jenkins*, the CI server, triggers a build and executes SonarQube for analysis. The output results are then sent to the SonarQube server for processing. After processing, the analysis results are stored in the SonarQube MySQL database and displayed in the SonarQube UI.

To extract the squads' historical data of past analyses, we wrote a script in Python that uses SonarQube's Web Service API (api/metrics/*) to fetch the metrics for all projects into JSON files. Figure 4.3 shows how SonarQube integrates into the development process and how our script retrieves the analysis reports.

Software metrics analysis. First, we checked the releases in SonarQube, and extracted the start dates of their development phase and their release dates. Then, we classified the releases as non-rapid or rapid based on the time interval between their release date and start date of the development phase (using 3 weeks as threshold). We did not consider the time period between the release dates of two consecutive releases since the development of a release can start before the release of the prior one. Although SonarQube offers a wide range of metrics, we solely considered the subset of metrics that are analyzed by all teams at ING to be consistent. For each release, we extracted the metrics that teams at ING measure to assess their coding performance and coding activity. Out of these metrics, *code churn* is used to assess the level of coding activity within a codebase, and the remaining metrics are seen as indicators for code quality:

1. *Coding Standard Violations:* the number of times the source code violates a coding rule. A large number of open issues can indicate low-quality code and coding debt in the system [46]. As part of this class of metrics, we looked more specifically into the *Cyclomatic Complexity* [51] of all files contained in a release.
2. *Branch Coverage:* the average coverage by tests of branches in all files contained in a release. A low branch coverage can be an indicator of testing debt in the system [24].
3. *Comment Density:* the percentage of comment lines in the source code. A low comment density can be representative of documentation debt in the system [24].
4. *Code Churn:* the number of changed lines of code between two consecutive releases. Since rapid releases deliver code in smaller batches, it can be expected that they correspond with a lower absolute code churn. However, it is not clear how rapid releases influence the normalized code churn.

Since code churn is calculated over the time between releases and this differs among squads, we normalized code churn by dividing it by the time interval between the release date and start date of the development phase. The code complexity and lines of code were used to examine if differences observed in the software quality are potentially caused by changes in the source code's size or complexity. As SonarQube does not account for differences in project size, we normalized the metrics by dividing them by *Source Lines of Code* (SLOC): the total number of lines of source code contained in a release.

Interpretation	Cohen's d
Small	0.148
Medium	0.330
Large	0.474

Table 4.2: Interpretation of Cliff's delta [15]

Finally, we performed a statistical comparison of the metrics between the group of rapid releases and the group of non-rapid releases. We used the statistical analysis tool R to perform all calculations. The *Shapiro-Wilk test* [72] shows that the SonarQube data is not normally distributed. Therefore, to compare between rapid and non-rapid releases, we used the non-parametric *Mann-Whitney U test* [48] and to study effect sizes we used *Cliff's delta* [13], which

varies between -1 and 1. Cliff's delta is based on the same non-parametric principles as the Mann-Whitney U test. Table 4.2 explains the interpretation of Cliff's delta.

4.4. Release Delay Data Collection

To compare the occurrence and duration of delays in rapid and non-rapid releases, we analyzed log data from *ServiceNow*, a backlog management tool used by a majority of the squads at ING. We received access to the log data of 102 squads for releases shipped between October 01, 2017 and October 01, 2018. The log data was exported as a comma-separated value text file through an export option in the tool. First, we checked the releases of each squad in the system, and extracted their start dates, planned release dates and actual release dates. The releases were classified as either rapid or non-rapid based on the time interval between their planned release date and that of the previous release (using 3 weeks as threshold). We acknowledge that the development of a release might start before the planned release date of the previous release but this interval is a good indicator of whether a release is *planned* to be rapid or non-rapid. For each release, we extracted the duration of the delay as the difference in days between the planned release date and actual release date. If a release was pushed to production before or on the planned release date, it was considered to be on time (zero delay).

Finally, we aggregated the delay measurements at release level to perform a statistical comparison of delays between rapid and non-rapid releases. To assess the statistical significance of differences in the percentage of timely releases and delay duration of rapid and non-rapid teams, we use the *Mann-Whitney U test* (Wilcoxon rank sum test) [48]. This test is suitable for comparing metrics on an ordinal scale. To study effect sizes we use *Cliff's delta* [13].

5

Results

This chapter presents our results on timing and quality characteristics of rapid releases derived from survey responses, release delay data and code quality data. First, we present the distribution of release frequencies at ING and explain how we classified teams as being rapid or non-rapid. Then we give an overview of the demographics of survey participants and discuss the perceived validity of related work in the development environment of ING. Next, we present our results that address the research questions as stated in Section 1.3. Example quotes from the survey are marked with [rX], where X refers to the corresponding respondent's identification number. The codes resulting from our manual coding process are underlined.

5.1. Rapid versus Non-Rapid Classification

The development context at ING is mixed, consisting of squads that release at different speeds. Figure 5.1 presents an overview of the teams' release frequencies derived from a custom 'cycle time' API at ING. We determined the average release cycle time of teams for releases shipped in the period from June 01, 2017 to June 01, 2018. The release cycle time is calculated as the time interval between the release date and start date of the development phase (date of first commit). The distribution shown is not fixed as teams at ING intend to keep reducing their release cycle times in the future.

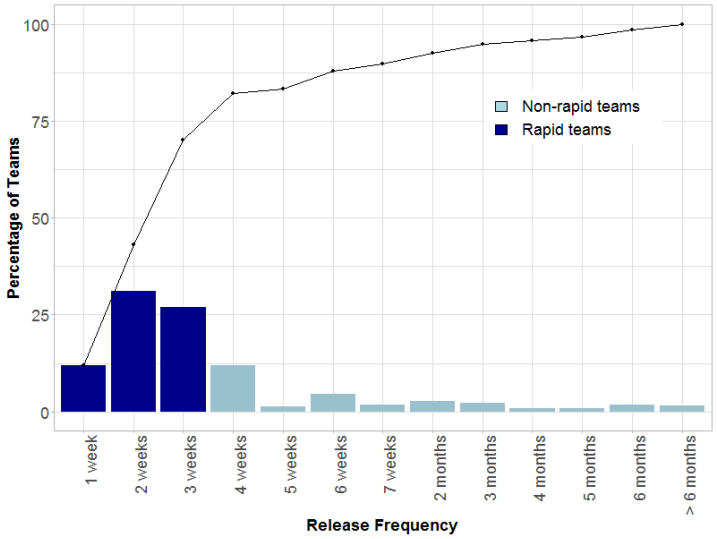


Figure 5.1: Distribution of release frequencies at ING: the black line represents the cumulative percentage of the squads.

For this study we divide the teams at ING in a rapid group and non-rapid group based on *how fast* they release relative to each other. This distinction allows for a statistical comparison between the two groups to explore if a shorter release cycle length influences time and quality aspects of releases. We acknowledge that, within a group, there might be differences among teams with different release cycle lengths. However, we consider this to be beyond the scope of this study.

As all squads at ING are expected to follow the rapid development model, there is no specific culture of rapid releases versus non-rapid releases that enabled us to make a differentiation between the two (*e.g.*, letting squads self-identify). We decided to use the median release cycle time as a point of reference since the distribution shown in Figure 5.1 is not normally distributed and contains outliers. In this case, it is convenient to use the median as it is less affected by outliers than alternatives, such as the mean or mode. We found that the median of the distribution of release cycle times at ING is 3 weeks.

Using the median as a point of reference, we classified teams as either *rapid* (release duration of ≤ 3 weeks) or *non-rapid* (release duration > 3 weeks). This way we identified 433 rapid teams (71%) and 178 non-rapid teams (29%) at ING. The rapid and non-rapid teams are completely disjoint as they follow a regular (fixed) release time interval and do not work with parallel release cycles of different lengths (see Section 3.1). In the same manner, we classified releases as either *rapid* (time interval between release date and start of development phase ≤ 3 weeks) or *non-rapid* (time interval between release date and start of development phase > 3 weeks). In general, rapid teams push rapid releases. However, if a delay causes a cycle length to exceed 3 weeks, a rapid team can push a non-rapid release.

What is the influence of team characteristics? We compared the characteristics of rapid and non-rapid teams to check if we are able to make a fair comparison between both groups. As reported in Section 3.2, teams selected for analysis are similar in size. The 95% confidence interval for team size ranges between 5 to 9 members for both rapid and non-rapid teams. In the next section, we will show that teams are similar in the number of survey respondents (both rapid and non-rapid: 95% CI of 1 to 2 respondents per squad) and distribution of experience in software development (both rapid and non-rapid: 95% CI of 10 to 20 years). Teams are also similar in project size (rapid: 95% CI of 63600 to 98900 SLOC, non-rapid: 95% CI of 55600 to 78600 SLOC) and project domain as depicted in Figure 5.2.

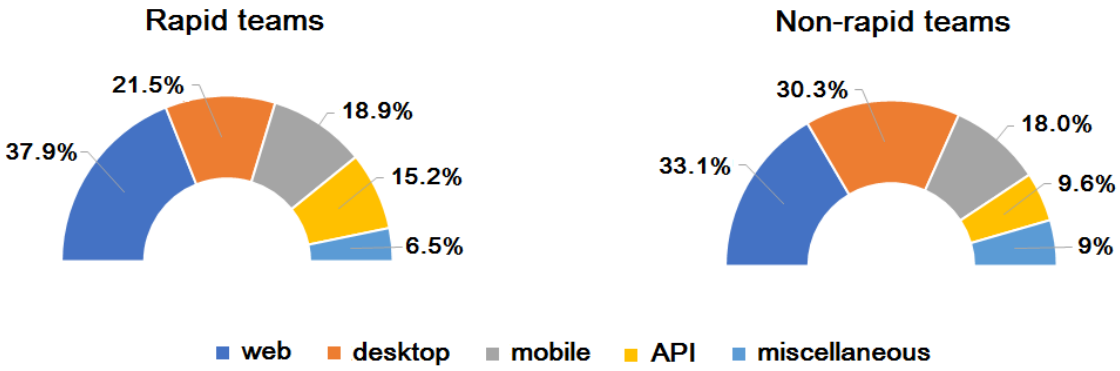


Figure 5.2: Distribution of project domains for rapid and non-rapid squads. Miscellaneous includes among others infra deliveries and reporting/ analytics scripts.

5.2. Characteristics of Survey Participants

Several demographical attributes were collected about the participants, namely their primary role at ING, experience within the software industry and experience within the company. These questions were intended to measure the representativeness of the sample. We also

asked participants to report their squad's name and release frequency. This allowed us to link survey responses with SonarQube data and to make a distinction between responses coming from participants that are part of rapid or non-rapid squads. In addition, we asked participants about the tools they use to analyze source code.

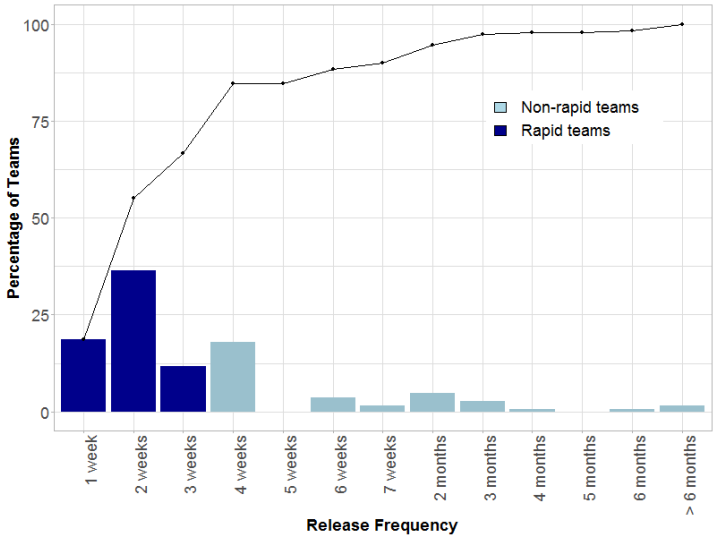


Figure 5.3: Distribution of release frequencies at ING derived from survey responses. The black line represents the cumulative percentage of the squads.

Release frequencies. As reported before, we obtained a number of 461 responses. An overview of the teams' release frequencies derived from survey responses is presented in Figure 5.3. 296 respondents reported to be part of a rapid squad (64%) and the remaining 165 respondents indicated to work in non-rapid squads (36%). The survey participants fairly represent the actual distribution of rapid versus non-rapid squads (rapid: 71%, non-rapid: 29%) at ING.

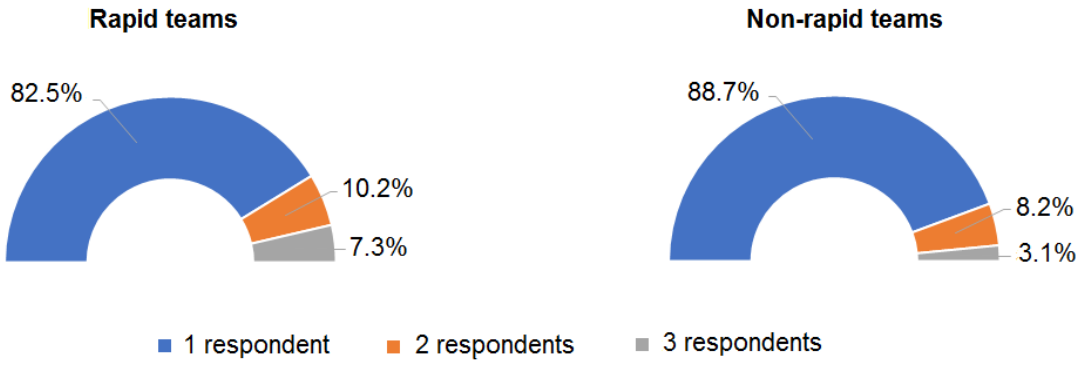


Figure 5.4: Distribution of the number of respondents per squad for rapid and non-rapid squads

Number of respondents. A distribution of the number of respondents per squad for rapid and non-rapid squads is shown in Figure 5.4. A large majority of both rapid (83%) and non-rapid (89%) squads are represented by 1 respondent. Notably, there is no squad with more than 3 respondents. This means that, most likely, no particular squad or particular type of squad was over-represented in the survey sample. To give participants the option to stay anonymous, we made it optional to enter a squad name in the survey. 75 participants (25.3%) from rapid squads and 54 participants (32.7%) from non-rapid squads decided to stay anonymous. Consequently, the actual distribution of the number of respondents per squad

could slightly deviate from the distribution we derived from survey responses. By matching squad names from the survey responses with account names in SonarQube, we confirmed that the respondents represent at least 177 rapid squads and 97 non-rapid squads. This corresponds with 41% and 54% of the total number of rapid and non-rapid squads at ING, respectively.

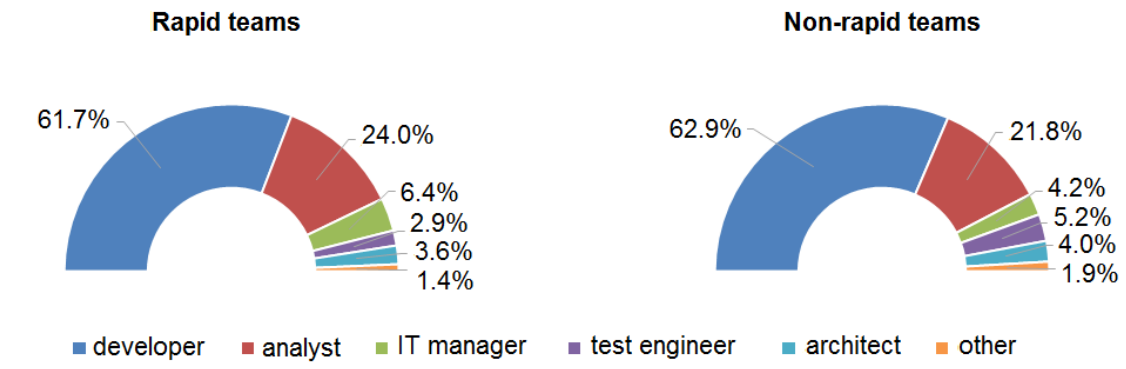


Figure 5.5: The primary roles of survey participants at ING

Team roles. An overview of the participants' distribution of roles in rapid and non-rapid squads is shown in Figure 5.5. In both groups, a majority of the participants (rapid: 68.2%, non-rapid: 72.1%) identified their primary role at ING as a software engineer (*i.e.*, developer, test engineer or architect). Other noteworthy roles are data analyst (rapid: 24%, non-rapid: 21.8%) and IT manager (rapid: 6.4%, non-rapid: 4.2%).

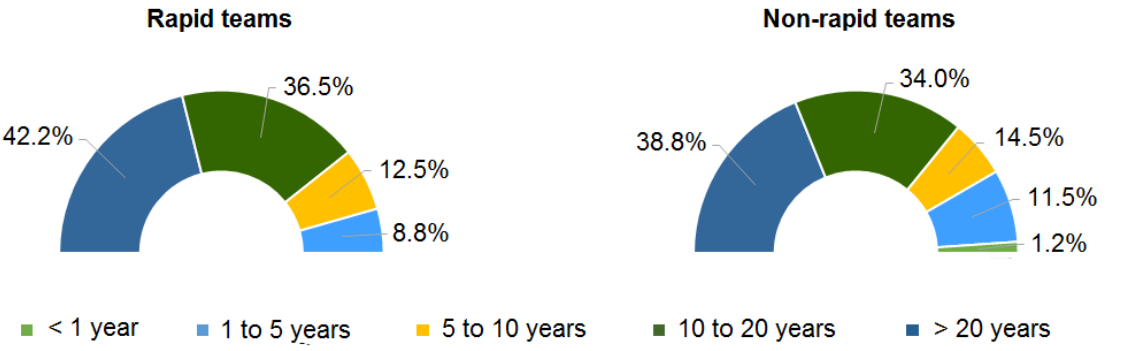


Figure 5.6: The experience of survey participants in the software industry

Experience in the software industry. A distribution of the experience of participants in the software industry and at ING is shown for both rapid and non-rapid squads in Figures 5.6 and 5.7, respectively. A large majority of both groups (rapid: 78.7%, non-rapid: 72.8%) reported to have more than ten years of experience in software development. Also, a majority of both groups (rapid: 61.9%, non-rapid: 54.6%) reported to have more than five years of experience at ING.

Usage of static code analysis tools. To check if the quality assessment of rapid and non-rapid squads is comparable, we asked participants that self-identified as a software engineer whether they monitor technical debt in their releases. A large majority of engineers from rapid squads (83.2%) and non-rapid squads (79.2%) reported to monitor the debt in their project. In addition, we asked participants if they use any static analysis tools next to ING's standard choice of tools (SonarQube and Fortify) to asses their code quality.

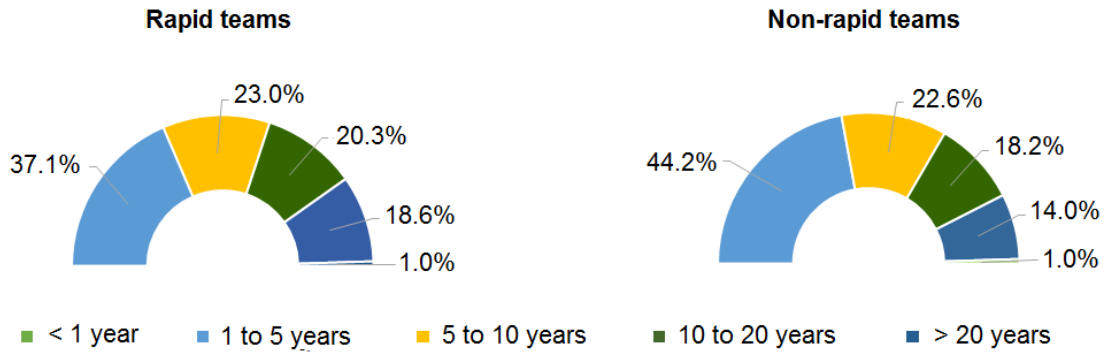


Figure 5.7: The experience of survey participants at ING

In both groups, a similar fraction of participants indicated to use additional tools, namely 28% from rapid squads and 20% from non-rapid squads. The tools *Coverity*, *Checkstyle*, *OWASP*, *FindBugs* and *PMD* were most mentioned in both groups.

5.3. Perceived Validity of Related Work at ING

We wanted to analyze the participants' perceptions on the validity of previous research findings in the development environment of ING. Their perceptions may serve for clarification in the comparison of our results with that of related work (see Section 6.2). To assess the perceived validity of related work in ING, we provided survey participants with a set of six 4-level Likert scale questions. The Likert scale consisted of statements that correspond to previous research findings related to various aspects of a rapid development environment, including time pressure, integration of bug fixes, testing and refactoring activities. All of these findings are discussed in Chapter 2 and selected based on their relevance to our research questions.

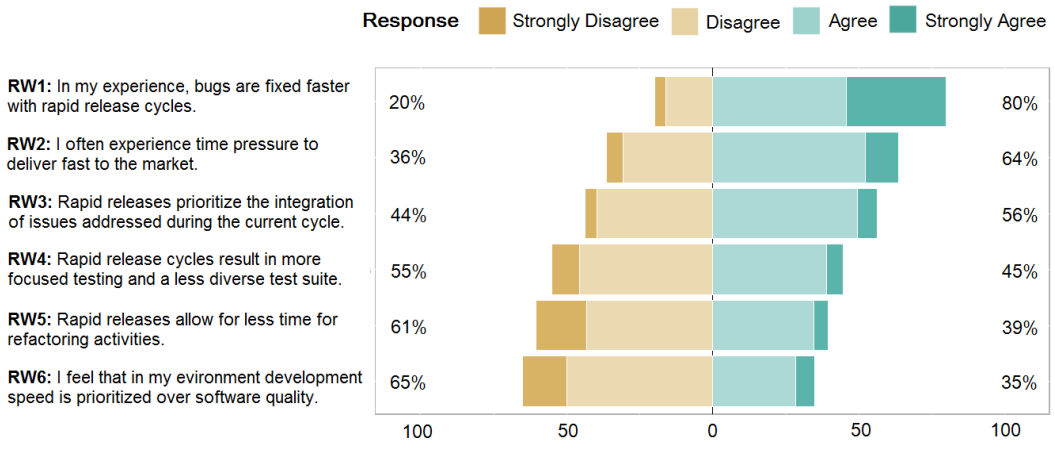


Figure 5.8: Validation of findings from previous work in the development environment of ING

An overview of the responses to the Likert scale questions in descending order of agreement is shown in Figure 5.8. A summary of the distribution and variation (quartiles) of the Likert data is presented in Table 5.1. The outcome to the Likert scale questions can be interpreted as follows: a higher level of agreement indicates a higher perceived validity of the corresponding finding in the context of ING.

Statement	SD	D	A	SA	25th	Median	75th
RW1	3.8%	16.2%	45.7%	34.3%	D	A	SA
RW2	5.6%	30.8%	52.3%	11.2%	D	A	A
RW3	38.8%	39.8%	49.5%	6.8%	D	A	A
RW4	9.5%	45.7%	39.0%	5.7%	D	D	A
RW5	17.3%	43.3%	34.6%	4.8%	D	D	A
RW6	15.1%	50.0%	28.3%	6.6%	D	D	A

Table 5.1: Level of agreement with statements RW1 - RW6 from related work. Columns show SD = Strongly Disagree, D = Disagree, N = Neutral, A = Agree and SA = Strongly Agree. Last three columns present the first (25%), second (median, 50%) and third (75%) quartile values.

Figure 5.8 shows that for three out of six findings (RW1, RW2 and RW3), a majority agrees with the finding. For the other three findings, a majority disagrees with the finding. Using the Mann-Whitney U test, we found that there are no significant differences between the responses from participants of rapid and non-rapid teams.

RW1: The first statement, "*In my experience, bugs are fixed faster with rapid release cycles*," refers to the finding that bugs are fixed significantly faster in rapid releases. This finding was reported by multiple studies, namely [19] (see Section 2.6.1) and [37, 38] (see Section 2.6.2). A majority (80%) of our respondents agree with this statement in the context of ING. In their open-ended responses, participants explained that the smaller increments in rapid releases make it easier to find where a bug is introduced and to fix it. Many respondents pointed out that this allows them to address and deliver bug fixes more quickly to the end user. For example, respondent [r44] explained: "[In rapid releases] *all low-hanging-fruit is picked up more quickly. Not only bugs, but also small improvements.*" Many respondents indicated that they are usually able to deliver bugfixes in the upcoming release. As a consequence of the high delivery speed, a small fraction of teams reported to use hot fixes less often in rapid releases (*i.e.*, an unscheduled release that fixes a specific bug or issue).

RW2: Regarding time pressure, the Likert scale included the statement "*I often experience time pressure to deliver fast to market*," which originates from the studies [49, 68] as discussed in Section 2.6.1. A majority (64%) of our respondents reported to agree with this statement. In their open-ended responses, participants explained that they experience pressure to deliver software on time, while managing stakeholder expectations. For example, respondent [r16] explained: "*The backlash for not delivering on time is a lot bigger than the praise for delivering on time.*" A small fraction of respondents from non-rapid teams indicated that the time pressure builds up in long release cycles and that they perceive it to be the worst during the final phase before issuing a major release.

RW3: Another finding related to bug fixes, "*Rapid releases prioritize the integration of issues addressed during the current cycle*," originates from [18] (see Section 2.6.1). A small majority (56%) of our respondents reported to agree with this statement. They revealed that issues in the current release cycle receive priority in case of moderate or high customer impact. A majority of the respondents that disagree with the statement explained that the scope of a release is already determined before the start of a sprint and that it remains unchanged during the sprint. The respondents' mixed opinions about this statement suggest that squads within ING apply different techniques for prioritizing issues.

RW4: Regarding testing, we provided respondents with the statement "*Rapid releases result in more focused testing and a less diverse suite*," with reference to the work of [49] (see Section 2.6.2). A small majority (55%) of our respondents reported to disagree with the statement. However, we noticed that the formulation of the statement led to confusion among respondents, which might have affected the results for this Likert scale question. Some participants misinterpreted the "less diverse suite" part of the statement as a reference to a test suite of reduced size instead of reduced effectiveness. A majority of the participants that dis-

agree with the statement stressed the importance of regression testing, chain testing and/or performance testing, which they continue performing in rapid releases. A majority of the respondents that agree with the statement explained that (due to the shorter timeframe of rapid releases) they limit their testing effort to new and updated functionality. Many participants of this group mentioned to cut the overall time spent on regression testing, which corresponds with our results on testing debt in Section 5.7.1. The respondents' mixed opinions about this statement indicates that squads within ING apply different principles and techniques for prioritizing tests.

RW5: Concerning refactoring activities, the Likert scale included the statement "*Rapid releases allow for less time for refactoring activities*," which originates from the work of [14, 81] (see Section 2.6.2). A majority (61%) of our respondents reported to disagree with the statement. Many of them explain that they are allowed to spend a certain fraction of time per sprint on refactoring, which is fixed (thus not dependent on release cycle length). Others report to work with clean-up cycles (*i.e.*, release cycles dedicated to refactoring debt and discussing design issues) which are planned separately from the regular releases. Most of the participants that reported to agree with the statement explained that rapid releases allow for a more continuous refactoring process, which keeps debt from piling up. For example, respondent [r193] expressed that: "*Rapid releases can force you to break up refactoring into chunks that can be released separately. That's a good thing actually. But most importantly, the contents of a release could very well be only technical, without direct visibility to end users.*"

RW6: Another finding related to time pressure states, "*I feel that in my environment development speed is prioritized over software quality*," which refers to the work of [68] in 22 high-tech companies (see Section 2.6.1). Interestingly, this is the statement that received the least amount of recognition by our respondents. Many respondents reported that their squads (together with the product owner) make a trade-off between delivery speed and their commitment to quality. For example, respondent [r165] reported: "*There is often pressure from stakeholders to deliver new features fast. But most teams will successfully balance that against commitment to quality.*" In addition, respondent [r97] explained: "*As a bank we have high quality standards to meet. Our code quality checks make sure there are no shortcuts.*"

5.4. RQ1: Frequency and Duration of Delays in Rapid Releases

For this research question we looked into perceived delay data from survey responses and delay data from ServiceNow. The results are summarized in Figure 5.9 and Table 5.2. Figure 5.9 shows a percentage distribution based on the percentage of times rapid and non-rapid teams perceive to and actually release software on time. The left side of the figure corresponds to the survey responses, while the right side visualizes the actual release delay data. Both the survey responses and delay data are aggregated at the team level in this figure. Table 5.2 presents a statistical analysis of the delay frequency and duration in the release delay data, aggregated at the release level.

5.4.1. Developers' Perceptions

For the survey question, "How often are your squad's releases on track?", 85% of teams had 1 respondent, 9% had 2 respondents (both responses accepted) and the remaining teams had 3 respondents (responses aggregated using majority vote). Using the Mann-Whitney U test, we found that the differences observed in the perceived percentage of timely releases for rapid and non-rapid teams are statistically significant at a confidence level of 95% (p-value = 0.003). We measured an effect size (Cliff's delta) of 0.925, which corresponds to a large effect.

The left side of Figure 5.9 shows that a majority of respondents from both non-rapid (60%) and rapid (76%) teams believe that they release software on time less than half of the time. The figure also shows that rapid teams believe to be more delayed than their non-rapid counterparts. On one extreme, 8% of rapid teams perceive to be on time 75% to 100% of the time.

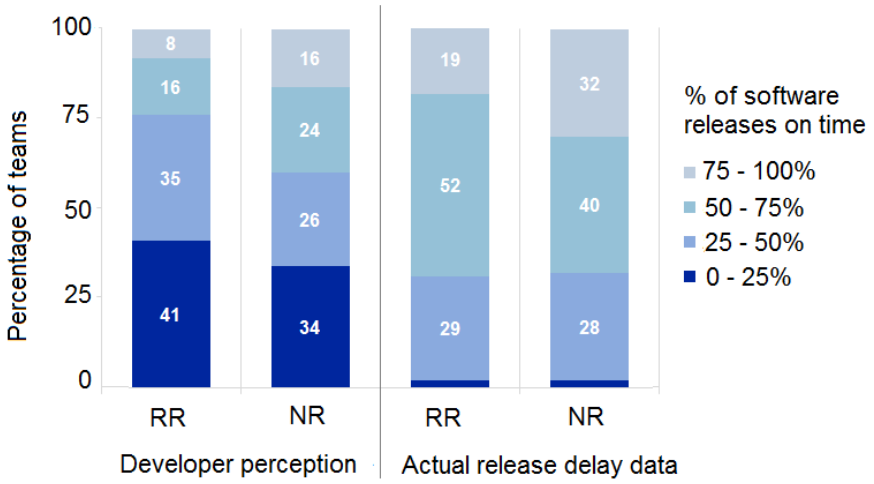


Figure 5.9: Percentage distribution of rapid and non-rapid teams based on the percentage of times they release software on time

The percentage doubles to 16% for non-rapid teams. On the other extreme, 41% of rapid teams perceive to be on time 0% to 25% of the time. The percentage drops to 34% for non-rapid teams.

Further analysis of the survey responses revealed that a majority of the rapid teams that perceive to be on track 75% to 100% of the time develop web applications (54%) and desktop applications (18%). The rapid teams that perceive to be less than 25% of the time on track develop mobile applications (68%) and APIs (19%).

5.4.2. Release Delay Measurements

According to the Mann-Whitney U test, the differences observed in the actual percentage of timely releases for rapid and non-rapid teams are statistically significant at a confidence level of 95% (p-value < 0.001). We measured an effect size (Cliff's delta) of 0.833, which corresponds to a large effect. We present our observations in a comparison of the release delay data and survey data below:

Observation I: A majority of both rapid and non-rapid teams release software more often on time than they perceive. The right side of Figure 5.9 shows that in general both rapid and non-rapid teams release software more often on time than our respondents believe. While the majority of respondents from both rapid and non-rapid teams perceive to release software on time less than 50% of the time, the data shows that a majority of teams (both rapid and non-rapid) deliver software on time more than half of the time. 41% of the rapid respondents and 34% of the non-rapid respondents believe to release software on time 0% to 25% of the time, while we could not find any team that is so often delayed in the actual release delay data. For the 50% to 75% and 75% to 100% categories we observe a similar trend.

Observation II: Rapid teams perceive to be, and in fact are, more commonly delayed than their non-rapid counterparts. The data corroborates the perception of respondents that rapid teams are more delayed than their non-rapid counterparts. One extreme is that 19% of rapid teams are on time 75% to 100% of the time, while the percentage increases to 32% for non-rapid teams. The percentage of teams that release software on time 25% to 50% of the time is similar for rapid (29%) and non-rapid (28%) teams.

Figure 5.10 shows the distribution of the percentage of timely software releases, grouped by rapid versus non-rapid releases. A majority of both rapid and non-rapid releases are delivered more than 60% of the time, however, rapid releases are mainly associated with a

lower percentage of software releases on time. Table 5.2 shows that the median percentage of timely software releases is 68.5% for non-rapid releases while it is 63.0% for rapid releases. According to the Mann-Whitney U test, this difference is statistically significant at a confidence level of 95% with a small effect size (Cliff's delta) of 0.257.

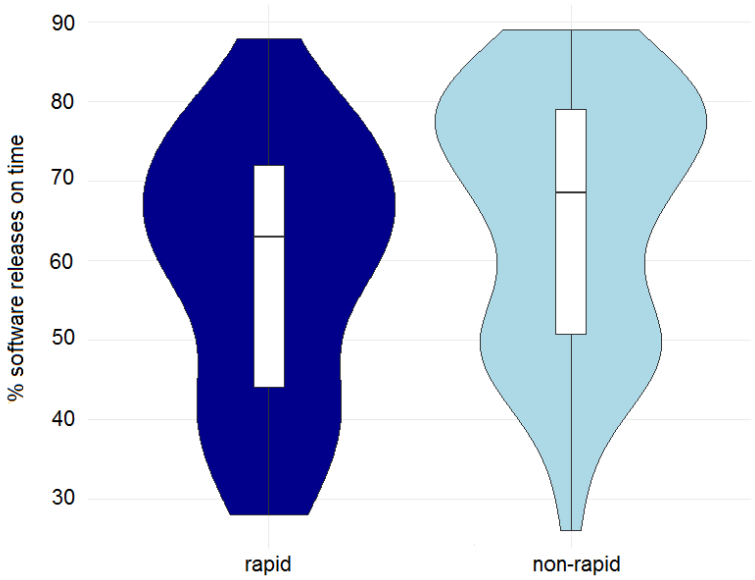


Figure 5.10: Distributions of the percentage of timely software releases grouped by rapid and non-rapid releases

Observation III: Rapid releases are correlated with shorter delays (median: 6 days) than non-rapid releases (median: 15 days). Although rapid releases are more often delayed than their non-rapid counterparts, further analysis of the delays shows that on average rapid releases are hindered by shorter delays (95% confidence interval: 5 to 9 days) than non-rapid releases (95% confidence interval: 18 to 31 days). We observe that delays in rapid releases take a median time of 6 days, while taking 15 days (median) in non-rapid releases. Figure 5.11 shows the distribution of delay duration, grouped by rapid releases versus non-rapid releases. In comparison with non-rapid releases, we observe that rapid releases have shorter delays than non-rapid releases. According to the Mann-Whitney U test, this difference is statistically significant at a confidence level of 95% with a large effect size of 0.695. Furthermore, in comparison with non-rapid releases, rapid releases exhibit more outliers above the upper quartile. This indicates that rapid releases are less predictable than non-rapid releases when it comes to the duration of delays that take longer than average.

Observation IV: The rapid teams that are perceived to be, and in fact are, least often on track develop mobile applications and APIs. Further analysis of the data showed a similar trend as observed for mobile applications and APIs in the survey responses. A majority of the rapid teams that are less than 50% of the time on track develop mobile applications (47%, average delay duration: 7 days) and APIs (12%, average delay duration: 5 days). We did not find any significant difference in project domains for the rapid teams that are on track more than 75% of the time.

Metric	Mean		Median		*P*-value	95% CI		Effect Size
	RR	NR	RR	NR		RR	NR	(Cliff's delta)
% software releases on time*	59.14	65.80	63.0	68.5	0.0394	[53.2, 64.8]	[61.4, 68.6]	-0.257 (small)
delay duration*	7	25	6	15	< 0.001	[5, 9]	[18, 31]	-0.695 (large)

* indicates statistical significance (p-value < 0.05)

Table 5.2: Effects of rapid releases on the percentage of timely software releases and delay duration (days). *P*-values are based on the Mann-Whitney U Test (Wilcoxon Rank Sum test).

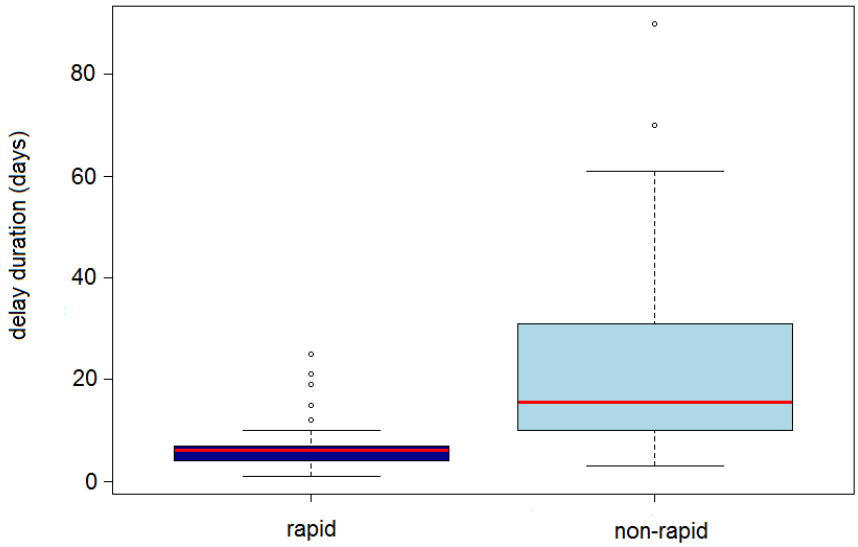


Figure 5.11: Distributions of the duration of delays in releases grouped by rapid and non-rapid releases

RQ1: How often do rapid and non-rapid teams release software on time?
Rapid teams perceive to be, and in reality are, more commonly delayed than their non-rapid counterparts. The majority of rapid teams that are least often on track develop mobile applications and APIs. Rapid releases are delivered 63% (median) of the time on time, while non-rapid releases are delivered 69% of the time on time. However, delays in rapid releases take a median time of 6 days, while taking 15 days in non-rapid releases. Considering both delay frequency and duration, rapid releases seem generally more effective with a delay of 2.22 days per release compared to 4.65 days per non-rapid release.

5.5. RQ2: Perceived Delay Factors in Rapid Releases

For this research question, we only analyzed survey responses, because quantitative data on release delay factors is not being collected by ING. Survey respondents mentioned several factors that they think introduce delays in releases. Table 5.3 shows the coding scheme that resulted from our coding process. This scheme contains an overview of all delay factors derived from survey responses, their description and frequency in survey responses. A list of the delay factors arranged in decreasing order of their occurrence in responses of participants from rapid teams is shown in Figure 5.12.

Observation I: Dependencies (especially in infrastructure) and testing are the top mentioned delay factors in rapid releases. Figure 5.12 shows that dependencies, infrastructure (which can be seen as a specific type of dependency) and testing (in general) are the most prominent factors that are perceived to cause delay in rapid teams.

Further analysis of the most prominent factor perceived to delay rapid and non-rapid teams (dependency) explained the sources of dependency in the organization. Developers, in their open-ended responses, attributed two types of dependencies to cause delay in their releases. At a technical level, developers have to deal with cross-project dependencies. Teams at ING work with project-specific repositories and share codebases across teams within one application. This often results in dependencies across teams and sometimes across products. At a workflow level, developers mention to be hindered by task dependencies. Inconsistent

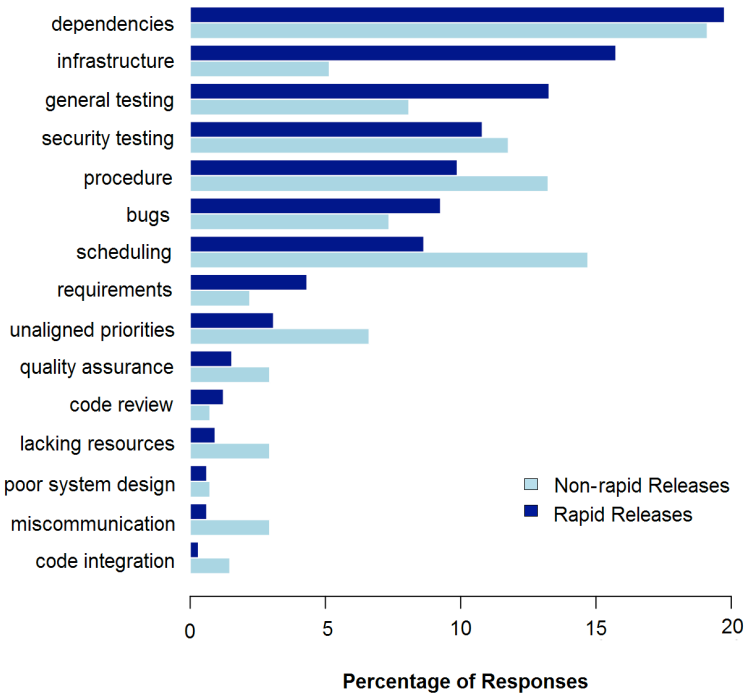


Figure 5.12: Factors perceived to cause delays in rapid and non-rapid teams

schedules, unaligned priorities and unavailable back-end services of other teams are perceived to cause delays in dependent teams. A small fraction of our respondents also report to be hampered by project delays and/or failure of third party software providers. Many developers seem to struggle with estimating the impact of both types of dependencies in the release planning.

Rapid teams also report delays related to infrastructure and testing (in general). These factors do not feature in the top mentioned factors influencing non-rapid teams. Regarding infrastructure, respondents mention that issues in infrastructure are related to the failure of tools responsible for automation (such as Jenkins and Nolio) and sluggishness in the pipeline caused by network or proxy issues. Respondent [r168] states that "*Without the autonomy and tools to fix itself, we have to report these issues to the teams of CDaas and wait for them to be solved*". A small fraction of the respondents reported to be hindered by late infrastructure deliveries, such as (Cloud) servers and firewalls. Regarding testing, developers mentioned that the unavailability and instability of the test environment induces delay in releasing software. Respondent [r11] states that "*In that case we want to be sure it was the environment and not the code we wish to release. Postponing is then a viable option*".

Other factors which were listed in at least 10% of responses are security testing, following mandatory procedures (such as for change management) prior to every release, fixing bugs, and scheduling the release, including planning effort and resources.

Observation II: Rapid teams and non-rapid teams experience similar issues, however, issues related to infrastructure and testing are mentioned more often by rapid teams. Similar to rapid teams, non-rapid teams report to be largely influenced by dependencies. The other factors which were considered important by at least 10% of both rapid teams and non-rapid teams are security testing, procedure and scheduling.

A factor which is perceived to prominently affect rapid and non-rapid teams is security testing. Any software release at ING needs to pass the required security penetration test and secure code review, which are centrally performed by the *CIO Security* department at ING. Respondents report that they often have to delay releases because of "*delayed penetration tests*" [r66], "*unavailability of security teams*" [r133] and "*acting upon their findings*" [r86].

Delay Factor	Description	Frequency	
		RR	NR
Dependencies	Technical dependencies and task dependencies due to cross-product collaboration	19.6%	19.4%
Infrastructure	(1) Sluggishness and failing tools in the build pipeline, (2) Delayed infrastructure deliveries	15.5%	5.8%
General testing	(1) Unavailable or unstable test environment, (2) Insufficient testing, (3) Unexpected test output	13.2%	7.9%
Security testing	(1) Delayed penetration tests, (2) Acting upon findings of the CIO Security department	10.8%	11.5%
Procedure	Following mandatory change procedures (OCD, DCAB) to get approval for release	9.8%	13.3%
Bugs	Unexpected (live) incidents and regressions that need to be fixed prior to releasing	9.1%	7.3%
Scheduling	Effort overruns due to inaccurate software effort estimation	8.8%	15.2%
Requirements	Unclear, incomplete and/ or changing software requirements	4.3%	2.4%
Unaligned priorities	Changing and/ or unaligned priorities between teams	3.0%	6.8%
Quality assurance	Long quality assurance process	1.4%	3.0%
Code review	Peer code reviewing takes an unexpected long time	1.4%	0.6%
Lacking resources	Missing required knowledge and time pressure	1.0%	3.0%
Poor system design	Inflexibility and issues in system design that make it harder for changes to be incorporated	0.7%	0.6%
Miscommunication	Lack of communication and false assumptions about the schedule of other teams	0.7%	3.0%
Code integration	Integration issues in merge requests with other teams	0.7%	1.2%

Table 5.3: Coding scheme resulting from our analysis of the open-ended survey responses for RQ2 “What factors are perceived to cause delay in rapid releases?”. Columns present the identified delay factors, their description and frequency in survey responses (percentage of rapid and non-rapid respondents).

When we discussed these problems with the security department, they explained that they usually have to delay security tests because of missing release notes or an unstable acceptance environment. As the squads are responsible for the delivery of such artifacts, we expect that rapid squads (that frequently take part in the approval process) are generally better prepared for security tests. This might be an explanation for the fact that security testing was more often mentioned as a delay factor by non-rapid teams than rapid teams.

Another factor that is perceived to delay rapid and non-rapid releases is following mandatory procedure for change management and quality assurance prior to every release. Many developers report that these procedures can be lengthy but necessary to get approval for a release. The factor scheduling includes planning effort and resources. A recurring issue in the survey responses is the unplanned work related to risk items. The results show that rapid teams mentioned scheduling as a delay factor less often than non-rapid teams. This may be caused by the fact that the smaller scope of rapid releases is found to simplify release planning.

The factors infrastructure and testing (in general) were significantly more often mentioned by participants from rapid teams than participants from non-rapid teams. Although more analysis is needed, we suspect that this difference is caused by the fact that rapid teams make more often use of the CDaaS pipeline and test environment for frequent releases. Therefore,

they have a higher chance of being delayed by infrastructural issues.

Observation III: Rapid teams that work on mobile apps and APIs perceive to be delayed by dependencies and testing. Further analysis of the survey responses showed that the rapidly released mobile applications and APIs that are least often on time (found in RQ1) are hindered by dependencies and testing. Many mobile app developers reported to experience delay due to dependencies on a variety of mobile technologies and limited testing support for mobile-specific test scenarios. API developers report to be delayed by dependencies in back-end services and expensive integration testing.

> ***RQ2: What factors are perceived to cause delay in rapid releases?***
> *Dependencies, especially in infrastructure, and testing are the top mentioned delay factors in rapid releases. Rapid teams and non-rapid teams perceive to experience similar issues (dependencies, security testing, following procedure and scheduling), however, issues related to infrastructure and testing are mentioned more often by rapid teams.*

5.6. RQ3: Addressing Release Delays

For this research question, we only analyzed survey responses, because quantitative data on how squads address release delays is not being collected by ING. Survey respondents mentioned *how* their teams address delays and *what* actions they undertake in their open-ended responses. Tables 5.4 and 5.5 present the approaches and actions that resulted from our coding process, respectively. The tables show an overview of the codes that we identified in the survey responses, their description and combined frequency in survey responses. An alluvial diagram of the identified approaches and actions is presented in Figure 5.13. From left to right, we show how the two main approaches are correlated with concrete actions undertaken by rapid and non-rapid teams. The width of the links visualizes the number of survey responses sharing the same categories (*i.e.*, codes).

Observation I: Teams report to address delays through *rescheduling* and *releasing as soon as possible*. Rapid teams report both approaches equally often, while a majority of non-rapid teams report to reschedule. We identified two main approaches that both rapid and non-rapid teams undertake in case of a release delay. Both rapid and non-rapid teams report to address delays through rescheduling, *i.e.* the action of postponing a release to a new date, and re-planning or re-prioritizing the scope of the delivery. Both groups also report to have the option to release as soon as possible, *i.e.* in the time span of a few days. Participants from rapid teams mentioned both approaches equally often, while a majority (76%) of the participants from non-rapid teams report to reschedule. This suggests that rapid teams are more flexible regarding release delays as they more often have the choice to manage a delay within a few days. Respondents explain that their teams decide to release as soon as possible when a release contains a critical bug fix or security update that needs to be shipped as quickly as possible. In such a case teams often use *release toggles* [27] or *partial releases*: "*We cherry-pick the critical features onto the current production branch into an intermediate release*" [r164]. Another respondent explains that toggles are also used to temporarily remove defects from the software: "*Using release toggles we can disable a new feature containing a bug, so that the release can still be pushed to production.*" [r60]

Observation II: Teams combine *rescheduling* most often with long-term oriented actions, such as discussing root causes and solving dependencies, while *releasing as soon as possible* is mainly associated with the short-term oriented action of solving issues. Participants from rapid squads and non-rapid squads report to undertake similar actions to address the factors underlying a delay. Further analysis of the survey responses showed that teams combine rescheduling most often with the actions of discussing the root cause of a delay, solving dependencies and (less often) solving issues. In their open-ended responses, participants explain that the longer time span of rescheduling allows teams to

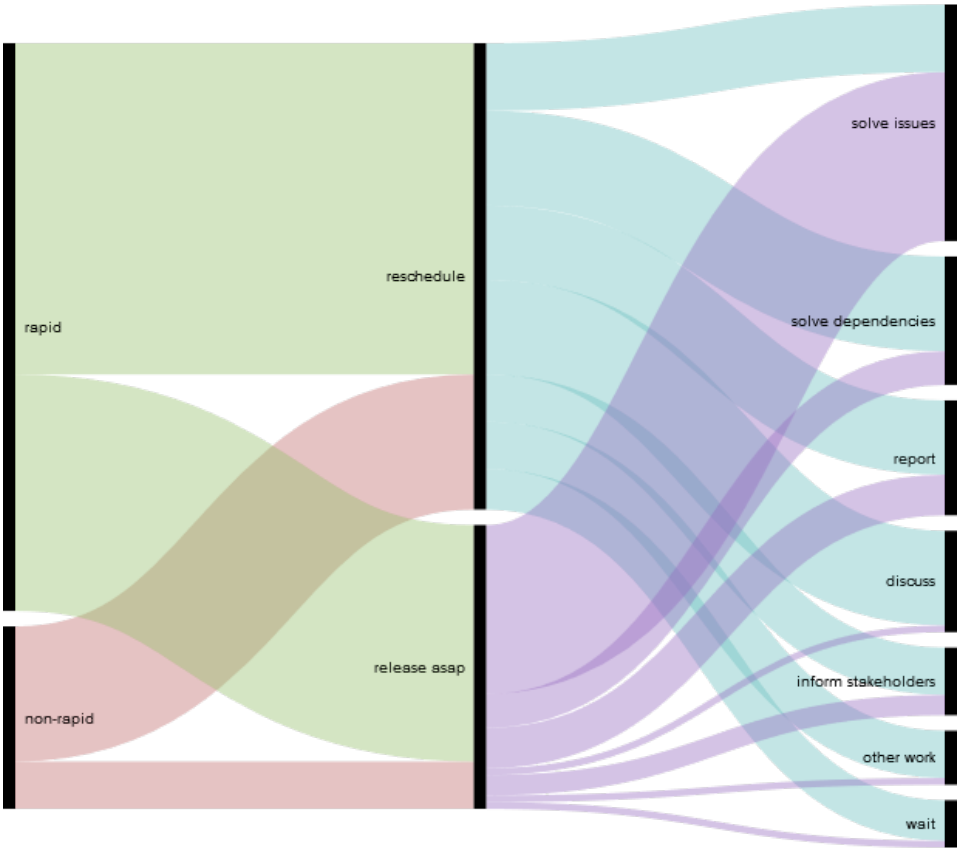


Figure 5.13: Overview of how rapid and non-rapid teams address release delays

review the situation and to reflect on lessons learned. Some respondents mentioned that their teams use the outcome of the discussion to create a special user story targeted at fixing the problem (*e.g.*, "*We discuss the problem within the team, create a story to tackle the root-cause of the problem and try to prioritize the fix (if we can fix it).*" [r2]). Compared to non-rapid teams, rapid teams more often report to tackle dependencies in case they reschedule. Other actions that are reported together with rescheduling by at least 10% of the respondents are communication-related: informing stakeholders and reporting (*i.e.*, to management and dependent teams).

The approach to release as soon as possible is most often reported to be combined with problem solving activities, termed as solving issues here. Another activity which is listed in at least 15% of the responses together with releasing as soon as possible is reporting. The action of solving dependencies is more often reported together with the long-term approach rescheduling. This suggests that dependencies are a type of issue which require more time to be solved than other problems.

RQ3: How do rapid and non-rapid teams address release delays?
Teams report to address delays through rescheduling and releasing as soon as possible. Rapid teams report both approaches equally often (53.3% release asap, 46.7% rescheduling), while a majority (76.4%) of non-rapid teams report to reschedule. Teams combine rescheduling most often with long-term oriented actions, such as discussing root causes and solving dependencies, while releasing as soon as possible is mainly done through the short-term oriented action of solving issues.

Approach	Description	Frequency	
		RR	NR
Release asap	Attempt to release in the time span of a few days	53.3%	23.6%
Reschedule	Postpone a release to a new date, and re-plan or re-prioritize the scope of the delivery	46.7%	76.4%

Table 5.4: The main two approaches used to address delays in releases

Action	Description	Frequency		Frequency (RA)		Frequency (RS)	
		RR	NR	RR	NR	RR	NR
Solve issues	Remove roadblocks and work harder to solve the issues that cause delay	29.4%	36.9%	21.3%	30.1%	8.1%	6.8%
Solve dependencies	Collaborate with the responsible squad(s) or third party(ies) to solve dependencies	16.6%	13.6%	6.2%	5.4%	10.4%	8.2%
Report	Raise an impediment and escalate to management and dependent squads	16.2%	10.7%	10.0%	7.1%	6.2%	3.6%
Discuss	Discuss root causes, refine the problem and reflect on lessons learned	14.2%	19.0%	3.4%	3.2%	10.8%	15.8%
Inform stakeholders	Inform project stakeholders that are affected by the delay	8.8%	9.7%	5.3%	1.8%	3.5%	7.9%
Other work	Do other work (e.g., administration, start on next release) while being delayed	7.4%	6.7%	2.1%	1.9%	5.3%	4.8%
Wait	In case of external issues, wait until they are fixed by another squad or third party	7.4%	3.4%	1.1%	1.2%	6.3%	2.2%

Table 5.5: Coding scheme resulting from our analysis of the open-ended survey responses for RQ3 “How do rapid and non-rapid teams address release delays?”. Columns show the identified actions, their description and frequency in survey responses (percentage of rapid and non-rapid respondents). The last two columns present the combined frequency of an action together with the approach of releasing as soon as possible (RA) and the approach of rescheduling (RS) for rapid and non-rapid respondents.

5.7. RQ4: Impact of Rapid Releases on Code Quality

For this research question, we only considered the 202 survey responses from developers in rapid teams. We removed 165 non-rapid respondents next to 94 rapid respondents who did not identify as a developer at ING.

5.7.1. Developers’ Perceptions

Developers had mixed opinions on how rapid releases affect code quality. A distribution of the effect of rapid releases (improve, degrade, no effect) on different factors related to code as perceived by developers is shown in Figure 5.14. It shows responses suggesting improvements in quality in green, degradation in quality in red and no effect in grey.

Observation I: Developers perceive the smaller changes and rapid feedback in rapid releases to improve code quality. A majority of developers perceive that the small changes in rapid releases make the code easier to review, positively impacting the refactoring effort (*e.g.* “*It gets easier to review the code and address technical debt*” [r16]). Developers also report

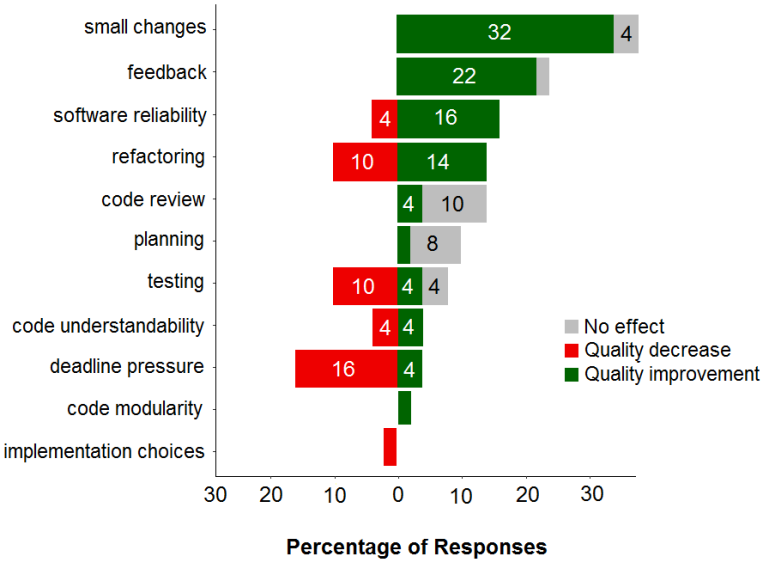


Figure 5.14: Developer perception of the impact of rapid releases on code quality aspects

that the small deliverables simplify the process of integrating and merging code changes, and they lower the impact of errors in development. A few developers mention that rapid releases motivate them to write modular and understandable code.

A large number of developers mention the benefits of rapid feedback in rapid releases. Feedback from issue trackers and the end user allows teams to continuously refactor and improve their code quality based on unforeseen errors and incidents in production. Rapid user feedback is perceived to lead to a greater focus of developers on customer value and software reliability (*e.g.* "*[Rapid releases] give more insight in bugs and issues after releasing. [They] enable us to respond more quickly to user requirements*" [r232], "*We can better monitor the feedback of the customers which increased [with rapid releases].*" [r130]). This enables teams to deliver customer value at a faster and more steady pace (*e.g.* "*[With rapid releases] we can provide more value more often to end users.*" [r65], "*Features are delivered at a more steady pace*" [r16]).

Observation II: Developers perceive the deadline pressure in rapid releases to reduce code quality. The short-term focus in rapid releases may result in design debt in the long-run. Many developers report to experience an increased deadline pressure in rapid releases, which can negatively affect the code quality. Developers explain to feel more pressure in shorter releases as these are often viewed as "*a push for more features*" [r143]. They believe that this leads to a lack of focus on quality and an increase in workarounds (*e.g.*, "*Readiness of a feature becomes more important than consistent code.*" [r26]). A few developers report to make poor implementation choices under pressure (*e.g., "In the hurry of a short release it is easy to make mistakes and less optimal choices"* [r320]).

Technical debt. We checked whether the respondents monitor technical debt in their releases through a multiple choice question in the survey. 168 out of 202 developers of rapid teams reported to monitor the debt in their project. For our analysis, we only considered responses from these developers and we focused on four common types of debt as identified in the work of Li et al. [46]: *coding debt*, *design debt*, *testing debt* and *documentation debt*. An overview of the responses to the Likert scale questions is shown in Figure 5.15. A summary of the distribution and variation (quartiles) of the Likert data is presented in Table 5.6. According to a majority of the developers, rapid releases do not result in accumulated debt of any type.

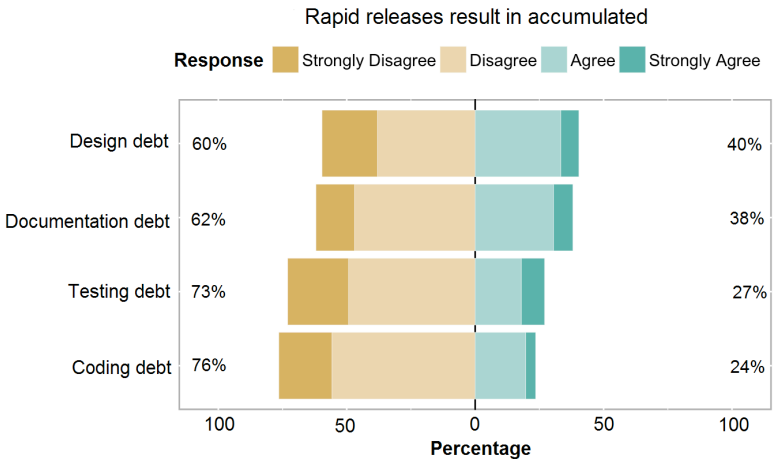


Figure 5.15: Developer perception of the impact of rapid release cycles on different types of technical debt

Type of debt	SD	D	A	SA	25th	Median	75th
Design debt	21.8%	38.2%	30.9%	9.1%	D	D	A
Documentation debt	20.2%	45.9%	25.7%	8.3%	D	D	A
Testing debt	28.7%	43.5%	16.7%	11.1%	SD	D	A
Coding debt	26.4%	46.2%	21.7%	5.7%	SD	D	A

Table 5.6: Level of agreement with the statement that rapid release cycles result in accumulated debt of a certain type. Columns show SD = Strongly Disagree, D = Disagree, N = Neutral, A = Agree and SA = Strongly Agree. Last three columns present the first (25%), second (median, 50%) and third (75%) quartile values.

Since the developers' explanations on coding debt were similar to the factors mentioned in Figure 5.14, we will now focus on other types of debt:

Design debt. Many developers report that the short-term focus of rapid releases makes it easier to lose sight of the big picture, possibly resulting in design debt in the long run. Especially in case of cross-product collaboration, rapid releases do not leave enough time to discuss design issues (*e.g.*, "*We are nine teams working together on the same application. Due to time constraints design is often not discussed between the teams.*" [r147])

Testing debt. A majority of developers mention rapid releases to have both positive and negative effects on their testing effort. Rapid releases are perceived to result in a more continuous testing process since teams update their test suite in every sprint. However, due to their shorter time span, developers report to focus their testing effort in rapid releases on new features and high-risk features. This focus is found to "*allow more complete testing of new features*" [r184] and to "*make it easier to determine what needs to be tested*" [r186]. Developers mention to spend less time on creating regression tests in rapid releases.

Documentation debt. A majority of the developers do not perceive a decrease in the amount of documentation in rapid releases. However, developers report that the short-term focus in rapid releases reduces the quality of documentation. When there is pressure to quickly deliver new functionality, documentation quality receives the least priority (*e.g.*, "*The need for high quality documentation is low in the short-term*" [r155], '*Documentation is the first which will be dropped in case of time pressure*" [r84]). Developers mention to cut corners by using self-documenting code, and by skipping high-level documentation regarding the global functionality and integration of software components.

5.7.2. Software Quality Measurements

To gain a more detailed insight into the code quality of the teams, we performed a comparative analysis of SonarQube measurements for rapid releases and non-rapid releases. We also compared the SonarQube results to the developers' perceptions by linking the data at a team level. To account for differences in project size, we normalized all metrics by SLOC. We do not quantify technical debt as, to the best of our best knowledge, there are no widely accepted metrics to approximate technical debt.

Statistical test. We performed a statistical test to assess whether the differences between the evaluated metrics for rapid releases and non-rapid releases are statistically significant. More broadly, such a test is used to interpret whether there are differences in the distributions of two independent groups for a dependent variable. In our case, the independent groups correspond with the group of rapid releases and the group of non-rapid releases. The dependent variable corresponds with the SonarQube metric that is being evaluated. We test the following null hypothesis H_0 and alternative hypothesis H_A for each metric i for major releases:

H_0^i**:** the distribution of values for metric i are equal for rapid releases and non-rapid releases.

H_A^i**:** the distribution of values for metric i are not equal for rapid releases and non-rapid releases.

Normality test. First, we applied a preliminary normality test, the *Shapiro-Wilk test* [72], to check whether the data is normally distributed. The output of this test indicates whether a parametric or non-parametric test is suitable for our dataset. The null-hypothesis of the Shapiro-Wilk test is that the population is normally distributed. Thus, if the p-value is less than the standard conventional significance level of 0.05, then the null hypothesis is rejected and there is evidence that the data is not normally distributed. We applied the Shapiro-Wilk test to each metric (test) separately and found that the SonarQube data is not normally distributed.

Mann-Whitney U test. We used the *Mann-Whitney U test* [48] to test our null hypothesis H_0^i for all metrics i. The Mann-Whitney U test is a non-parametric statistical test used for assessing the significance of differences between the distributions of two independent groups. Non-parametric statistical methods make no assumptions about the distributions of the assessed variables, which is necessary since the values of our metrics are not normally distributed.

Bonferroni correction To adjust for multiple comparisons we apply the Bonferroni correction [7]. This correction is used when several dependent or independent statistical tests are being performed at the same time. In our case, we are assessing statistical significance for five metrics (thus five tests) simultaneously. In order to prevent false positives, the Bonferroni correction sets the confidence level for the entire set of n comparisons equal to α and the confidence level for each comparison equal to $\frac{\alpha}{n}$. To account for the number of comparisons

Metric	Mean		Median		*P*-value	95% CI		Effect Size
	RR	NR	RR	NR		RR	NR	(Cliff's delta)
Coding Violations Density*	0.03	0.05	0.02	0.03	0.00234	[0.02, 0.04]	[0.03, 0.06]	-0.595 (large)
Cyclomatic Complexity*	0.14	0.17	0.15	0.16	0.00418	[0.13, 0.16]	[0.14, 0.20]	-0.336 (med)
Branch Coverage*	57.51	43.02	68.40	49.15	0.00016	[48.90, 76.70]	[27.40, 59.00]	0.286 (small)
Comment Density	10.13	11.07	7.20	8.30	0.04533	[4.79, 12.50]	[5.11, 15.00]	-0.109 (negl)
Code Churn*	0.05	0.03	0.03	0.02	0.00782	[0.03, 0.07]	[0.02, 0.05]	0.263 (small)
SLOC	67066	83983	7447	9286	0.09045	[63600, 98900]	[55600, 78600]	-0.181 (negl)

* indicates statistical significance (*p*-value < 0.01, Bonferroni-corrected)

Table 5.7: Effects of rapid releases on software metrics. SLOC was used as a normalization factor. *P*-values are based on the Mann-Whitney U Test (Wilcoxon Rank Sum test). Effect sizes are indicated to be large, medium, small or negligible.

being performed, we divided the conventional significance level of 0.05 by 5 (the total number of metrics), giving a corrected significance level of 0.01. This means that the p-value of the Mann-Whitney U test for a specific metric needs to be < 0.01 to reject the null hypothesis.

Effect size. To assess how large the effect of the release cycle length on the metrics are, we measure Cliff's delta [13] as effect size because it is non-parametric and based on the same principles as the Mann-Whitney U test. Cliff's delta or d measures how often the values in one distribution are larger than the values in a second distribution. The measure d is given by:

$$d = \frac{\sum_{i,j}[x_i > x_j] - [x_i - x_j]}{mn}$$

where the two distributions are of size n and m with items x_i and x_j, respectively, and $[(\ldots)]$ is the Iverson bracket, which is 1 when the contents are true and 0 when false. Norman Cliff suggested that d values of $0.148, 0.330$ and 0.474 correspond to small, medium and large effects.

Observation III: The SonarQube data supports quality improvement views: software built with rapid releases is correlated with a lower cyclomatic complexity, a lower density of coding issues and a higher test coverage. The results of our analysis are summarized in Table 5.7 and Figure 5.16. The figure shows the violin plots of the assessed metrics for rapid releases and non-rapid releases. A violin plot is similar to a box plot, however it also depicts the probability density of the data at different values.

We present our results of the code quality data analysis below:

- ***The cyclomatic complexity is significantly lower in software built with rapid releases.*** Figure 5.16a shows that rapid releases are associated with a lower cyclomatic complexity than non-rapid releases. The Mann-Whitney U test confirms this observation (p-value = $0.00418 \leq 0.01$), therefore we can reject the null hypothesis. The effect size yields a d value of -0.336, which should be interpreted as a negative, medium effect size according to Norman Cliff's rule of thumb. This indicates that a rapid release cycle is moderately correlated with a lower cyclomatic complexity.

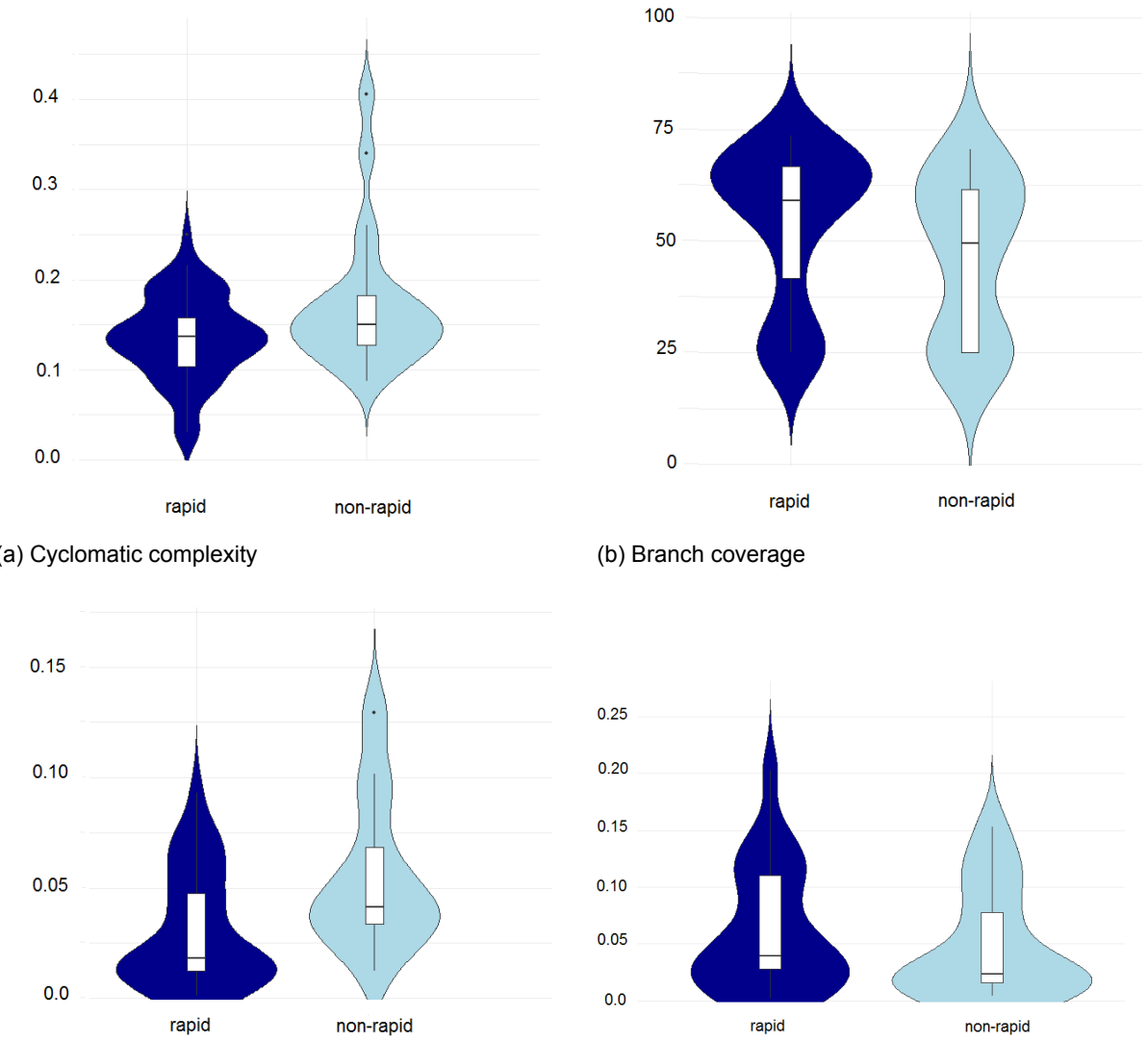
  This result corresponds with the perceptions of developers on coding debt. Developers do not perceive an increase in coding debt, and they report to find it easier to review and refactor code in rapid releases. This makes it likely for them to write less complex code and fix issues more quickly.

  Figure 5.16a shows a greater variance in cyclomatic complexity for non-rapid releases.

- ***The density of coding standards violations is significantly lower in software built with rapid releases.*** Figure 5.16c shows that the rapid releases are mainly associated with a lower coding standards violations density than non-rapid releases. The Mann-Whitney U test confirms this observation (p-value = $0.00234 \leq 0.01$), therefore we can reject the null hypothesis. The effect size yields a d value of -0.595, which should be interpreted as a negative, large effect size according to Norman Cliff's rule of thumb. This indicates that a rapid release cycle is strongly correlated with a lower coding standard violations density.

  This result corresponds with the perceptions of developers on coding debt. Developers do not perceive an increase in coding debt, and they report to find it easier to review and refactor code in rapid releases. This makes it likely for them to fix issues more quickly.

  Figure 5.16c shows a greater variance in coding standards violations density for non-rapid releases.



(a) Cyclomatic complexity

(b) Branch coverage

(c) Coding violations density

(d) Code churn

Figure 5.16: Violin plots of the measured cyclomatic complexity, coding violations density, branch coverage and code churn for rapid and non-rapid releases.

- ***The branch coverage is significantly higher in software built with rapid releases.*** Figure 5.16b shows that rapid releases are mainly associated with a higher branch coverage than non-rapid releases. Both boxes contain two peaks (one around 25% branch coverage and the other close to 60% branch coverage), however, the violin plot for rapid releases shows a higher frequency of higher branch coverage values. The Mann-Whitney U test confirms this observation (p-value = 0.00016 ≤ 0.01), therefore we can reject the null hypothesis. The effect size yields a d value of 0.286, which should be interpreted as a positive, small effect size according to Norman Cliff's rule of thumb. This indicates that a rapid release cycle is slightly correlated with a higher branch coverage.

  This result corresponds with the perceptions of developers on the testing effort. Developers report the test process in rapid releases to become more continuous and to allow for more complete testing of new features. As a consequence, rapid releases are likely to exhibit a higher test coverage.

- ***The code churn is significantly higher for rapid releases.*** Figure 5.16d shows that rapid releases are mainly associated with a higher code churn than non-rapid releases. The Mann-Whitney U test confirms this observation (p-value = 0.00782 ≤ 0.01), therefore we can reject the null hypothesis. The effect size yields a d value of 0.263, which should be interpreted as a positive, small effect size according to Norman Cliff's rule of thumb. This signifies that a rapid release cycle is slightly correlated with a higher code churn.

  This result indicates that there is a higher coding activity in rapid teams than in non-rapid teams. As developers did not mention code churn in their responses, we cannot compare this result with their perceptions. It is a possibility that developers are not aware of a higher coding activity in rapid releases.

- ***There is no significant difference between the comment density of rapid releases and non-rapid releases.*** The Mann-Whitney U test yielded a p-value of 0.04533, and, therefore, we cannot reject the null hypothesis. It produced a Cliff's delta value of -0.109, which is a negative but negligible effect size. This indicates that rapid releases show a small decrease in comment density, however, the difference with non-rapid releases is trivial.

  While not statistically significant, a lack of difference is consistent with the perceptions of developers on documentation debt. Developers perceive that rapid releases have an impact on the quality of documentation, but not on the amount (density) of documentation.

RQ4: How do rapid release cycles affect code quality?
Developers have mixed perceptions on the effect of rapid releases on code quality. On one hand, they perceive the smaller changes and rapid feedback in rapid releases to simplify code reviewing and to strengthen the developers' focus on user-perceived quality. On the other hand, they perceive rapid releases to negatively impact implementation choices and possibly design due to deadline pressure and a short-term focus. The code quality data supports the views on code quality improvements as indicated in a higher test coverage, a lower average complexity and a lower density of coding standards violations.

6

Discussion

In this chapter we discuss our main findings and compare them with related work. We also consider implications for practice and design, and finally, we discuss threats to the validity of this study.

6.1. Main Findings

We first revisit our main findings from Chapter 5 and discuss related directions for future research.

Delay factors in rapid releases. We found several factors that might delay rapid releases. Our results show that testing and dependencies are the top mentioned delay factors in rapid teams. In addition, rapid releases in API and mobile app development are more often delayed than other project domains. These findings suggest that project type, or perhaps certain inherent characteristics in different project types, are a delay factor in rapid releases. A study of project properties that delay rapid releases could help the field determine when rapid releases are appropriate to use. Afterwards, future research could examine which project types and organizations do not fit well with the rapid release methodology. Initial work in this direction has been carried out by Kerzazi & Khomh [36] and Bellomo et al. [6].

Total release delay and customer value delivery. Our results show that rapid releases are more commonly delayed than non-rapid releases. However, it is not clear what this means for the project as a whole, given that non-rapid releases are correlated with a longer delay duration. How do rapid releases impact the total delay of a project? This also raises the question of how rapid releases impact the overall perspective of features developed over time. Do rapid releases (despite those delays) help companies to deliver more customer value in a timely manner? Our respondents report that rapid releases enable them to deliver customer value at a faster and more steady pace, which suggests that over time rapid teams could deliver more customer value than non-rapid teams. Future research should compare the total release delay and customer value delivered in rapidly and non-rapidly releasing projects. This could give us a better insight into the effectiveness of rapid releases in terms of timing and customer experience. Another interesting opportunity is to explore the wish list of rapid teams. Which tools, techniques or methods would reduce the total release delay?

The balance game: security versus rapid delivery. Many of our respondents perceive security testing to induce delay in rapid teams. This suggests that organizations make a *trade-off* between rapid delivery and security. A financial organization like ING, with security critical systems, may choose to release less frequently to increase time available for security testing. As one of the respondents puts it "*It is a balance game between agility and security. Within ING the scale balances heavily in favor of security, thereby effectively killing off agility.*" [r21] Further analysis is needed to explore the tension between rapid deployment of new

software features and the need for sufficient security testing. To what extent does security testing affect the cycle time of software releases? In this context, Clark et al. [12] studied security vulnerabilities in Firefox and showed that rapid releases do not result in higher vulnerability rates. Malicious users require time to exploit vulnerabilities in newly released software. Further research is needed in this direction to clear up the interaction between both factors. Such insights could help organizations to find the right balance between the speed of software delivery and maintaining secure software.

Nevertheless, the time span of a release cycle limits the amount of security testing that can be performed. Therefore, further research should also focus on agile security to work towards the automation of security testing, and the design of security measures that are able to adapt to the changes in a rapid development environment. New methods for rapid security verification and vulnerability identification could help organizations to keep pace with rapid release cycles.

Dependency management. Our results show that the timing of both rapid releases and non-rapid releases is perceived to be influenced by dependencies in the ecosystem of the organization. Our respondents report to experience difficulties in assessing the impact of dependencies. There is a need for more insight into the characteristics and evolution of these dependencies. How can we quantify their combined effect on the overall ecosystem? This problem calls for further research into streamlining dependency management. Decan et al. [21][20] studied how dependency networks tend to grow over time, both in size and package updates, and to what extent ecosystems suffer from issues related to package dependency updates. Hedjedrup et al. [30] proposed the construction of a fine-grained dependency network extended with call graph information. This would enable developers to perform change impact analysis at the ecosystem level and on a version basis.

Code quality. We found that rapid releases can be beneficial in terms of code reviewing and user-perceived quality, even in large organizations working with hundreds of software development teams. Our quantitative data analysis shows that that software built with rapid releases tends to have a higher branch coverage, lower average complexity and higher code churn. Previous research [58] reported improvements of test coverage at unit-test level but did not look into other code quality metrics. It is an interesting opportunity for future work to analyze how rapid releases impact code quality metrics in other types of organizations and projects.

Challenges related to code aspects concern design debt and the risk of poor implementation choices due to deadline pressure and the short-term focus of rapid releases. This is in line with previous work [68, 82] that showed that pressure to deliver features for an approaching release date can introduce code smells. A study of factors that cause deadline pressure in rapid teams would be beneficial. It may be that projects that are under-resourced or under more time pressure are more likely to adopt rapid releases, instead of rapid releases leading to more time pressure. The next step would be to identify practices and methods that reduce the negative impact of the short-term focus and pressure in rapid releases.

6.2. Comparison with Related Work

In this section, we integrate our results back into literature and provide a comparison. We follow the set-up of Chapter 2.

Motivations for Adopting Rapid Releases

Earlier studies on rapid releases focused on the motivations behind the adoption of rapid releases. Begel and Nagappan [5] found that the main motivations for practicing rapid releases relate to easier planning, more rapid feedback and a greater focus on software quality.

→ *Our findings*: Our study found similar benefits (see Section 5.7.1). Developers at ING reported that the smaller changes and rapid feedback in rapid releases improve their focus

on user-perceived quality. The greater focus on the end user is perceived to enable teams to deliver customer value at a faster and more steady pace. Furthermore, we found that the small increments in rapid releases are perceived to make it easier to review, integrate and merge code changes.

Time Aspects of Rapid Releases

Integration delay of fixed issues. Costa et al. [19] found that rapid release cycles cause delays in the integration and delivery of issues. Issues are fixed faster in rapid releases but, surprisingly, rapid releases take a median of 54% longer than non-rapid releases to deliver fixed issues. The authors explain that this may be caused by the fact that non-rapid releases prioritize the integration of backlog issues, while rapid releases prioritize issues that were addressed during the current cycle [18].

→ *Our findings*: In line with the work of Costa et al., we found that rapid releases are more commonly delayed than non-rapid releases. Our study complements prior work by exploring how often rapid teams release software on time (see Section 5.4) and what the perceived causes of delay are (see Section 5.5). We extended our analysis by performing a statistical comparison of differences in delay duration and project domain between delays of rapid releases and non-rapid releases.

Factors impacting lead time. Kerzazi and Khomh [36] studied the factors impacting the release cycle time of software releases and found that testing is the most time consuming activity.

→ *Our findings*: In line with the work of Kerzazi and Khomh, we found that testing is (amongst dependencies and infrastructure) one of the top mentioned delay factors in rapid releases (see Section 5.5).

Increased time pressure. The strict release dates in rapid releases are claimed to increase the time pressure under which developers work. Rubin and Rinard [68] conducted a study at 22 high-tech companies and found that most developers experience significant pressure to deliver new functionality quickly.

→ *Our findings*: Our study corroborates the finding that developers experience increased deadline pressure in rapid releases (see Section 5.3). Developers at ING perceive time pressure to be a factor that decreases code quality in rapid releases.

Quality Aspects of Rapid Releases

Technical debt. Tufano et al. [82] found that deadline pressure for an approaching release date is one of the main causes for code smell introduction in the last month before issuing a major release. Industrial case studies of Codabux and Williams [14], and Torkar et al. [81], showed that a rapid development speed is perceived to increase technical debt.

→ *Our findings*: Our study complements prior work by analyzing the impact of rapid releases on certain code quality metrics and different types of debt (see Section 5.7). In line with aforementioned studies, we found that the deadline pressure in rapid releases is perceived to result in a lack of focus and an increase in workarounds. We found that design debt is a bigger concern in rapid releases than other types of debt. However, in contrast with the work of Codabux and Williams [14], developers at ING do not perceive rapid releases to result in accumulated debt of any type. Although this finding requires further analysis, it may be caused by the fact that a majority of developers at ING do not agree with the finding of [14, 81], "*Rapid releases allow for less time for refactoring activities*" (see Section 5.3 - RW5), and the finding of [68], "*My rapid environment prioritizes development speed over software quality*" (see 5.3 - RW6).

Software reliability & testing. In the OSS context, rapid releases are found to have a significant impact on the reliability of software as well as the testing process. Khomh et al. [37, 38] found that less bugs are fixed in rapid releases, proportionally, and the bugs that were not fixed led to crashes earlier during software execution. Mäntylä et al. [49] showed that in rapid releases testing has a narrower scope that enables a deeper investigation of features and regressions with the highest risk. Moreover, rapid releases lead to a more continuous testing process with proportionally smaller spikes before the main release. Multiple studies [45, 60] found that testers of rapid releases lack time to perform time-intensive tests, such as performance testing.

→ *Our findings*: In line with the work of Mäntylä et al. [49], we found that rapid releases are perceived to result in more continuous and focused testing (see Section 5.7). Although a small majority of our respondents reported to disagree with the finding of Mäntylä et al. (see Section 5.3 - RW4), we believe that this might have been caused by the confusion about the corresponding Likert scale question. Our respondents reported to limit their testing effort in rapid releases to new and high-risk features, and to spend less time on creating regression tests. Our study complements aforementioned studies by comparing the branch coverage of rapid releases to that of non-rapid releases.

Quality monitoring. Multiple studies [31, 45, 49, 55] have shown that rapid releases ease the monitoring of quality and motivate developers to deliver quality software.

→ *Our findings*: In line with aforementioned studies, our respondents perceive rapid releases to result in a greater focus on customer value and software reliability (see Section 5.7). The rapid feedback from issue trackers and end users make it easier for teams to monitor and improve their code quality. A small fraction of our respondents mentioned that rapid releases motivate them to write modular and understandable code.

What is the overall contribution of this thesis in comparison with related work? *Previous studies that focus on rapid releases as main study target are explanatory and largely conducted in the context of OSS projects. In this thesis, we present new knowledge by performing an exploratory case study of rapid releases in a large software-driven organization. We are the first to study rapid releases at a scale of over 600 teams, contrasting them with non-rapid releases. This thesis is also the very first study to look into the impact of rapid releases on release delays, delay factors and internal code quality.*

6.3. Implications

In the following, we discuss implications for the design of release engineering pipelines and recommendations for practitioners.

6.3.1. Design Implications

Our results show that rapid teams are commonly hindered by infrastructural (environmental) issues and cross-project dependencies. Such issues call for the ability to query the current and historical state of environments to track down dependencies. To that end, release engineering pipelines should be designed with the overriding goals of *reproducibility* and *traceability*:

- **Reproducibility:** As discussed in Section 2.5, reproducibility refers to the automation of steps in the release engineering pipeline. Automation allows for a consistent, repeatable and faster process of software releases [36].
- **Traceability:** Traceability in software delivery refers to the ability to interrelate a software engineering artifact to any other, to maintain associations between artifacts over time, so-called *trace links*, and to use the resulting network of trace links to answer

questions such as which releases contain a certain code change [47]. Traceability can serve multiple purposes, such as error recovery, assessment of the impact of dependencies and improvement of development efforts. The challenge of achieving traceability is related to the problem of generating accurate and consistent trace links, which may be further complicated by factors such as increased project size and tool diversity. Previous research indicates that organizations struggle to achieve sufficient traceability in software delivery due to lacking support from methods and tools [8, 47]. Rapid releases emphasize the need for tools that quickly determine the versions of every dependency used to create a given environment and to compare with previous versions. Such tools would help engineers to resolve conflicting dependency versions more quickly, thereby enabling more frequent releases. This problem calls for further research into technical infrastructure and the design of traceability tooling that is fully integrated with the software delivery pipeline [65]. Previous work in this direction has been carried out by Stahl et al. [78]. They designed a framework that automatically generates trace links for artifacts in the continuous delivery pipeline.

6.3.2. Recommendations for Practitioners

Here we present four areas that call for further attention from organizations that work with rapid releases.

Managing code quality. In our study we observed that a minority of rapid teams claim not to experience the negative consequences of rapid releases on code quality. When we compared their practices to that of other rapid teams, we noticed that the self-reported 'good' teams are doing regular code reviews and dedicate (at least) 25% of the time per sprint on refactoring. Teams that experience the downsides of rapid releases mention to spend less than 15% of the time on refactoring or to use 'clean-up' cycles (*i.e.*, cycles dedicated to refactoring). Although further analyses are required, we recommend organizations to integrate regular (peer) code reviews in their teams' workflows and to apply continuous refactoring for at least 15% of the time per sprint. Our respondents report that these practices are a way to block low quality work and motivate developers to write better code.

Developers' misconceptions about delays. Although the perceptions of developers at ING are generally in line with the quantitative data, our results show differences in perception on the percentage of timely releases (RQ1, "How often do rapid and non-rapid teams release software on time?"). This indicates that there is a misunderstanding among developers about the timing of their releases. ING should make developers aware of this misunderstanding and make data related to the delivery of projects more insightful to teams. This would allow teams to gain performance insights that resonate with their objectives of planning. We recommend organizations that are similar to ING to investigate the perceptions of their developers on the impediments to faster software delivery (such as delays and reduced code quality). In case a perceived impediment is not supported by quantitative data, organizations should make teams aware of this misunderstanding.

Release planning. Regarding delays, our respondents express the need for more insight on improving software effort estimation and streamlining dependencies. Although software effort estimation is well studied in research (even in rapid releases: [59, 69]), issues relating to effort estimation continue to exist in industry. This calls for a better promotion of research efforts on release planning and predictable software delivery. Organizations should invest more in workshops and training courses on release planning for their engineers. We also recommend organizations to apply recent approaches, such as automated testing, *Infrastructure as Code* and dependency management to the problem of delays in rapid releases.

Releasing fast and responsibly. Developers report to feel more deadline pressure in rapid releases, which can negatively affect their implementation choices and result in an increase in workarounds. This is also reported by previous work [49, 68]. Organizations should not view rapid releases as a push for features. A sole focus on functionality will harm their code quality and potentially slow down releases in the long run. We believe that it is less

effective to motivate organizations to slow down and produce better code quality than helping developers to release fast while breaking less. Future work should identify practices that enable organizations to release fast while maintaining a high level of code quality. Such efforts should attempt to enhance the ways that software development teams communicate, coordinate and assess coding performance.

6.4. Threats to Validity

While performing our study we observed several threats to the validity. In this section we discuss three different types of threats to the validity and how to mitigate them.

Internal validity. Threats to internal validity concern factors that influence the outcome of our study but are not part of the independent variable. In our study, the threats to internal validity concern factors, besides the release cycle length, that might have influenced our results. One factor that could affect the qualitative analysis (*e.g.*, for manual coding) is the bias induced by the involvement of the authors with the studied organization. The author interned at ING at the time of this study. To counter the biases which might have been introduced by the author, the thesis supervisor helped in designing survey questions. The observations and interpretation of the findings were cross-validated by three other expert colleagues in qualitative research. Another known risk of the coding process is the loss of accuracy of the original response due to an increased level of categorization. To mitigate this risk, we allowed multiple codes to be assigned to the same answer.

In our survey design, we phrased and ordered the questions in a sequential order of activities to avoid leading questions and question-order effects [52] (*e.g.*, one question could provide context for the next one and lead respondents to a specific answer). To mitigate this bias, we did not randomize the questions, but we ordered them in a sequential order of activities to help the respondent to concentrate on the context of the questions asked. Social desirability bias [28] (*i.e.*, the tendency of respondents to answer questions in a way that is viewed favorably by others) may have influenced the responses. To mitigate this risk, we made the survey anonymous and let the participants know that the responses would only be evaluated statistically.

Although we compared the characteristics of rapid and non-rapid teams (see Section 5.2), we cannot account for the effect of potential third variable explanations that might have affected our results for rapid versus non-rapid teams. Examples of such factors are project difficulty and scheduling optimism which might differ across teams. It is also a possibility that rapid teams at ING work on software components that are more easy to release rapidly. Participant observations suggest that the customer's security requirements might play a role, further analysis is required. Furthermore, we found that the fraction of non-rapid teams that is working in the mobile domain is 11.4% higher than rapid teams. Related work [35, 57] shows that mobile apps generally have a longer release cycle as they are developed and tested across different platforms, and need to be released through app stores. The differences in mobile domain distribution among teams at ING might have led to too optimistic results for rapid teams. Especially the timing-related results for RQ1, "How often do rapid and non-rapid teams release software on time?", and RQ2, "What factors are perceived to cause delay in rapid releases?", might have been affected by this confounding factor. While a difference in project domain distribution may influence, we argue that a difference of 11.4% is not large enough to significantly influence our final results.

Construct validity. The construct validity of the survey was assessed through a formal pilot run and consultation with expert colleagues in qualitative research. Because participants might mostly remember the last or most remarkable situations that occurred, we speculate that our data, therefore, presents a limited view of the actual effects of rapid releases. For example, the SonarQube data shows a positive correlation between rapid releases and higher code churn values, however, code churn was not mentioned by the survey respondents. Although we enriched the survey data with quantitative data from SonarQube and ServiceNow,

it is likely that our analysis does not represent or include all possible effects of rapid release cycles in industry.

An additional factor that could influence the survey's construct validity is the (un)familiarity of participants with the technical terms used in the questions (*e.g.*, technical debt). To mitigate this risk, (1) we provided formal definitions for difficult terms, (2) we only considered responses from developers for the code quality-related questions, and (3) we made the questions on technical debt optional in case a respondent did not understand the concept of technical debt. However, we acknowledge that there is a possibility that developers who were unfamiliar with certain terms still answered the questions.

The quantitative analysis could be affected by our choice of metrics. We analyzed the metrics that teams at ING measure to assess code quality, however, metrics that are not being monitored might also be affected by rapid releases. For future work, it would be interesting to explore the impact of rapid releases on other measures of software quality. Regarding the release delay data, a risk is that the delays we derived from ServiceNow data do not resemble the actual delays. There is a possibility that teams do not always close releases in ServiceNow on time. Future work should look into data collected from versioning control tools to retrieve more accurate release dates and release delays. We did not have access to this kind of data at ING.

External validity. The threats to external validity concern the generalization of results. As our study only considers one organization, it is important to consider the limitations to the generalizability of our work. We control for variations in our study using a large number of participants, code quality data from 3048 projects spanning a time period of two years and release delay data from 102 teams spanning a time period of a year. We did not impose any restrictions on the sample projects, such as programming language or use of technologies. Although the results are obtained from a large, global organization, we cannot generalize our conclusions to other organizations. Replication of this work in other organizations is required to reach more general conclusions. We believe that further in-depth explorations (*e.g.*, interviews) and multiple case studies are required before establishing a general theory of rapid releases.

We conjecture that our findings are applicable to software-driven organizations that are similar to ING in terms of scale and security level. Our findings indicate a trade-off between rapid delivery and security testing. In a financial organization like ING there is no tolerance for failure in some of their business-critical systems. This may have influenced our results, making our findings only applicable to organizations with similar business- or safety-critical systems. We cannot account for the impact of the large scale of ING on our results. Further research is required to explore how the scale of an organization and project relates to the findings. Replication of this study in organizations of different scale, type and security level is therefore required.

7

Conclusion and Future Work

The appeal of delivering new features faster has led many software projects to change their development processes towards rapid release models. As rapid releases are increasingly being adopted in open-source and commercial software, it is vital to deepen our understanding of the practices, effectiveness, and challenges surrounding rapid releases in large organizations. To that end, we conducted an exploratory case study, addressing timing and quality characteristics of rapid releases at ING. We followed a mixed-methods approach in which we integrated qualitative data from a survey that was answered by 461 participants, and quantitative data on release delays and code quality.

In this chapter, we conclude this thesis by revisiting the research questions we posed in Chapter 1 and by discussing promising directions for future work.

7.1. Conclusion

In this thesis, we have answered four main research questions through the combined analysis of survey data, code quality data and release delay data:

RQ1: How often do rapid and non-rapid teams release software on time?
Rapid teams perceive to be, and are in fact, more commonly delayed than their non-rapid counterparts. The majority of rapid teams that are least often on track (less than 50% of the time) develop mobile applications and APIs. This suggests that project type, or perhaps certain inherent characteristics in different project types, are a delay factor in rapid releases. An analysis of the actual release delay data shows that rapid releases are delivered 63% (median) of the time on time, while non-rapid releases are delivered 69% of the time on time. However, rapid releases are correlated with shorter delays than non-rapid releases. Delays in rapid releases take a median time of 6 days, while taking 15 days in non-rapid releases. Taking both the delay frequency and delay duration into account, rapid releases seem generally more effective with a delay of 2.22 days per release compared to 4.65 days per non-rapid release.

RQ2: What factors are perceived to cause delay in rapid releases?
Dependencies, especially in infrastructure, and testing are the top mentioned delay factors in rapid releases. Dependencies are the most prominent factor perceived to delay rapid and non-rapid teams. Participants report to be hampered by two types of dependencies due to cross-product collaboration. At a technical level, teams have to deal with cross-project dependencies across teams and sometimes across products. At a workflow level, teams are hindered by task dependencies (inconsistent schedules, unaligned priorities and unavailable back-end services). The issues related to infrastructure and testing do not feature in the top mentioned factors of non-rapid teams. Regarding infrastructure, participants mention to be

delayed by sluggishness and the failure of tools in the delivery pipeline. Regarding testing, teams are mainly hindered by the unavailability and instability of the test environment.

RQ3: How do rapid and non-rapid teams address release delays?
Teams report to address delays through *rescheduling* (postponing a release to a new date, and re-planning or re-prioritizing the scope of the delivery) and *releasing as soon as possible* (in the time span of a few days). The latter approach is taken in case of a critical bug fix or security update that needs to be shipped as quickly as possible. Rapid teams report both approaches equally often (53.3% release asap, 46.7% rescheduling), while a majority (76.4%) of non-rapid teams report to reschedule. This suggests that rapid teams are more flexible regarding release delays. Both rapid and non-rapid teams combine *rescheduling* most often with long-term oriented actions, such as discussing root causes and solving dependencies, while *releasing as soon as possible* is mainly done through the short-term oriented action of solving issues.

RQ4: How do rapid release cycles affect code quality?
Developers have mixed perceptions on the impact of rapid releases on code quality. On one hand, they perceive the smaller changes and rapid feedback in rapid releases to simplify code reviewing and to strengthen the developers' focus on user-perceived quality. The greater focus on user-perceived quality is perceived to enable teams to deliver customer value at a faster and more steady pace. On the other hand, they perceive rapid releases to negatively impact implementation choices and possibly design due to deadline pressure and a short-term focus. Developers report that these issues may result in design debt in the long run. Furthermore, developers perceive rapid releases to result in more continuous and focused testing. They report to focus their testing effort on new, high-risk features, which is perceived to come at the expense of regression testing. Regarding documentation, rapid releases are not perceived to affect the documentation density in software. However, developers report to cut corners with self-documenting code, and by skipping high-level descriptions about the global functionality and integration of software components.

The code quality data supports the views on code quality improvements as indicated by a higher test coverage, a lower average complexity and a lower density of coding standards violations. Rapid releases are also correlated with a higher code churn which indicates a higher coding activity in rapid teams.

7.2. Future Work

Based on our findings we identified several future research directions calling for further attention. In Section 6.1, we discussed promising future directions for research in the context of our main findings. Here we present more concrete research directions related to the applicability of rapid releases, the role of confounding factors, the opportunities for rapid feedback, and management of delays and code quality. Progress in these areas is crucial to better utilize the benefits of rapid releases in large, industrial, software-driven organizations.

Fine-grained analysis of release cycle length. We acknowledge that, within a group, there might be differences among teams with different release cycle lengths. However, we consider this to be beyond the scope of this study. An interesting opportunity for future work is to group releases into more fine-grained groupings of weekly release intervals. This would allow for a deeper analysis of the impact of a reduced release cycle length. It would be interesting to explore whether our findings still hold for these fine-grained groupings. Another direction for future work is to explore possible confounding factors. Although we explored the role of several factors (demographics and development practices) that are likely to affect release cycle time, further analysis is required to explore possible confounding factors. Participant observations suggest that customer's high security requirements might play a role. Future work could examine these factors and eliminate them through statistical controls (*e.g.*, through a form of multiple regression).

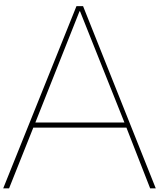
Feedback-driven development. Our results show that the rapid feedback in rapid releases is perceived to improve the focus of developers on the quality of software. Feedback obtained from end users, code reviews and static analysis can be used to guide teams to focus on the most valuable features, and to enable automated techniques to support various development tasks, including log monitoring, and various forms of testing. Such techniques can be used to further reduce the release cycle length. An exploration of these opportunities would help organizations to get the best out of their data and to improve the quality of their software. An extension of the data with runtime information (*i.e.*, performance engineering) and live user feedback that is integrated into the integrated development environment could be beneficial.

Long-term effect on software quality. An interesting opportunity for future work is to explore the long-term effect of rapid releases. By analyzing longitudinal data of quality measurements before and after teams switched to rapid releases, it can be measured how metrics relate to release cycle length over time. What is the long-term effect of a shorter release cycle length on the code quality and user-perceived quality of releases? How do issues related to design debt and time pressure develop in the long run? Such insights could help organizations better understand the long-term implications of rapid releases for software quality and the experience of end users.

Overall perspective of delays. Although our results show that rapid releases are more often delayed than non-rapid releases, it is unclear how a reduced release cycle length affects the timing of a project as a whole, given that non-rapid releases are correlated with longer delays. In this study, we focused on the number of delay days per release, however, future work should examine the number of delay days over the duration of a project. How do release delays evolve throughout different phases of a project? This could provide us a better insight into the total delay of rapidly versus non-rapidly releasing projects. Moreover, we were not able to perform data triangulation for RQ2, “What factors are perceived to cause delay in rapid releases?”, since quantitative data on release delay factors is not being collected by ING. It is therefore an interesting opportunity for future work to consider proxy quantitative measurements for delay factors and perform triangulation.

Customer value delivery. An interesting direction for future research is related to the impact of rapid releases on the delivery of customer value. How do rapid releases impact the overall perspective of features developed over time? Do rapid releases (despite frequent delays) help companies to deliver more customer value in a timely manner? A comparison of the customer value delivered in rapidly and non-rapidly releasing projects could give us a better insight into the effectiveness of shorter release cycles. This also raises the question of how customers experience rapid releases. Do rapid releases help companies to manage customer expectations as they evolve? Can customers keep up with a continuous stream of updates?

Appendices



Survey Questions

The following is an overview of the questions that were asked in the survey.

A.1. Demographics

1. What is your squad's name? (open-ended)

2. Which if the following best describes your role at ING?
   a. (lead) developer
   b. data analyst
   c. IT manager
   d. test engineer
   e. architect/ design
   f. other

3. How many years of work experience do you have in the software development industry?
   a. Less than 1 year
   b. 1 to 5 years
   c. 5 to 10 years
   d. 10 to 20 years
   e. More than 20 years

4. How long have you been working at ING?
   a. Less than 1 year
   b. 1 to 5 years
   c. 5 to 10 years
   d. 10 to 20 years
   e. More than 20 years

A.2. Timing

5. My squad releases every...
   a. 1 week
   b. 2 weeks
   c. 3 weeks
   d. 4 weeks
   e. 5 weeks
   f. 6 weeks
   g. 7 weeks
   h. 2 months
   i. 3 months
   j. 4 months
   k. 5 months
   l. 6 months
   m. > 6 months

6. How often are your squad's releases on track?
   a. 0 - 25% of the time
   b. 25 - 50% of the time
   c. 50 - 75% of the time
   d. 75 - 100% of the time

A.3. Release Delays

7. In your experience, what is the most common reason for release delays? (open-ended)

8. What do you, as a squad, do when a release is delayed? (open-ended)

A.4. Quality Monitoring

9. Apart from SonarQube and Fortify, do you use any other static code analysis tools to assess your code quality?
   a. Yes
   b. No, I only use SonarQube and/or Fortify
   c. No, I don't use any static code analysis tools

10. Apart from SonarQube and Fortify, which static code analysis tools do you use to assess your code quality? (open-ended, if answered 'yes' to the previous question)

11. Does your squad actively monitor the technical debt in your releases?
    a. Yes
    b. No

A.5. Code Quality & Technical Debt

Code Quality

12. In your experience, how do rapid release cycles affect your project's source code (technical) quality? (open-ended)

13. In your experience, how do rapid release cycles affect the user-perceived quality of your squad's releases? (open-ended)

14. "*Rapid release cycles result in accumulated technical debt.*" Do you agree or not? Please explain your answer. (open-ended)

Technical debt (Likert scale)

15. "*Rapid release cycles result in accumulated technical debt.*"

    (Design debt is caused by poor design and architecture)

16. "*Rapid release cycles result in accumulated documentation debt.*"

    (Documentation debt is caused by a lack of documentation for code logic or by poor documentation quality)

17. "*Rapid release cycles result in accumulated testing debt.*"

    (Testing debt is caused by a lack of testing or by poor testing quality)

18. "*Rapid release cycles result in accumulated coding debt.*"

    (Coding debt, or programming debt, is caused by low code quality)

A.6. Related Work (Likert scale)

19. "*I often experience time pressure to deliver fast to the market.*"

20. "*I feel that in my environment development speed is prioritized over software quality.*"

21. "*Rapid release cycles prioritize the integration of issues addressed during the current cycle.*"

22. "*In my experience, bugs are fixed faster with rapid release cycles.*"

23. "*Rapid releases allow for less time for refactoring activities.*"

24. "*Rapid release cycles result in more focused testing and a less diverse test suite.*"

Bibliography

[1] Bram Adams and Shane McIntosh. Modern release engineering in a nutshell–why researchers should care. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 5, pages 78–90. IEEE, 2016.

[2] Devdatta Akhawe and Adrienne Porter Felt. Alice in warningland: A large-scale field study of browser security warning effectiveness. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 257–272, 2013.

[3] Olga Baysal, Ian Davis, and Michael W Godfrey. A tale of two browsers. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 238–241. ACM, 2011.

[4] Kent Beck and Erich Gamma. *Extreme programming explained: embrace change*. addison-wesley professional, 2000.

[5] Andrew Begel and Nachiappan Nagappan. Usage and perceptions of agile software development in an industrial context: An exploratory study. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 255–264. IEEE, 2007.

[6] Stephany Bellomo, Robert L Nord, and Ipek Ozkaya. A study of enabling factors for rapid fielding: combined practices to balance speed and stability. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 982–991. IEEE Press, 2013.

[7] C Bonferroni. Teoria statistica delle classi e calcolo delle probabilita. *Pubblicazioni del R Istituto Superiore di Scienze Economiche e Commericiali di Firenze*, 8:3–62, 1936.

[8] Elke Bouillon, Patrick Mäder, and Ilka Philippow. A survey on usage scenarios for requirements traceability in practice. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 158–173. Springer, 2013.

[9] V. Bregar. What is the difference between canary, beta, rc and stable releases in android studio?, 2017. URL `https://android.jlelse.eu/what-is-the-difference-between-canary-beta-rc-and-stable-releases-in-\the-android-studio-bbbb77e7c3cf`.

[10] Maëlick Claes, Mika Mäntylä, Miikka Kuutila, and Bram Adams. Abnormal working hours: effect of rapid releases and implications to work content. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 243–247. IEEE, 2017.

[11] Sandy Clark, Michael Collis, Matt Blaze, and Jonathan M Smith. Moving targets: Security and rapid-release in firefox. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1256–1266. ACM, 2014.

[12] Sandy Clark, Michael Collis, Matt Blaze, and Jonathan M Smith. Moving targets: Security and rapid-release in firefox. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1256–1266. ACM, 2014.

[13] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494, 1993.

[14] Zadia Codabux and Byron Williams. Managing technical debt: An industrial case study. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, pages 8–15. IEEE Press, 2013.

[15] Jacob Cohen. *Statistical power analysis for the behavioral sciences*. Routledge, 2013.

[16] Juliet M Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative sociology*, 13(1):3–21, 1990.

[17] John W Creswell and J David Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

[18] Daniel Alencar da Costa, Surafel Lemma Abebe, Shane McIntosh, Uirá Kulesza, and Ahmed E Hassan. An empirical study of delays in the integration of addressed issues. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 281–290. IEEE, 2014.

[19] Daniel Alencar da Costa, Shane McIntosh, Uirá Kulesza, and Ahmed E Hassan. The impact of switching to a rapid release cycle on the integration delay of addressed issues-an empirical study of the mozilla firefox project. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 374–385. IEEE, 2016.

[20] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in oss packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 2–12. IEEE, 2017.

[21] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, pages 1–36, 2018.

[22] Andrej Dyck, Ralf Penners, and Horst Lichter. Towards definitions for release engineering and devops. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*, pages 3–3. IEEE, 2015.

[23] Addison-Wesley Pearson Education. *Software Project Management, Walker Royce, 1998: Project Management*, volume 1. Bukupedia, 1998.

[24] Robert J Eisenberg. A threshold based approach to technical debt. *ACM SIGSOFT Software Engineering Notes*, 37(2):1–6, 2012.

[25] T. Fernandes. Spotify squad framework, 2017. URL `https://medium.com/productmanagement101/spotify-squad-framework-part-i-8f74bcfcd761`.

[26] Uwe Flick. *An introduction to qualitative research*. Sage Publications Limited, 2018.

[27] M. Fowler. Feature toggles (aka feature flags), 2017. URL `https://martinfowler.com/articles/feature-toggles.html`.

[28] Adrian Furnham. Response bias, social desirability and dissimulation. *Personality and individual differences*, 7(3):385–400, 1986.

[29] Volker Gruhn and Clemens Schäfer. Bizdevops: because devops is not the end of the story. In *International Conference on Intelligent Software Methodologies, Tools, and Techniques*, pages 388–398. Springer, 2015.

[30] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 101–104. ACM, 2018.

[31] Hadi Hemmati, Zhihan Fang, and Mika V Mantyla. Prioritizing manual test cases in traditional and rapid release environments. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, pages 1–10. IEEE, 2015.

[32] Hennie Huijgens, Arie Van Deursen, and Rini Van Solingen. The effects of perceived value and stakeholder satisfaction on software project impact. *Information and Software Technology*, 89:19–36, 2017.

[33] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader)*. Pearson Education, 2010.

[34] ING. 2017 annual report ing group n.v., 2018. URL https://www.ing.com/About-us/Annual-reporting-suite/Annual-Report/2017-Annual-Report-Empowering-people.htm.

[35] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. Real challenges in mobile app development. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 15–24. IEEE, 2013.

[36] Noureddine Kerzazi and Foutse Khomh. Factors impacting rapid releases: an industrial case study. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, page 61. ACM, 2014.

[37] Foutse Khomh, Tejinder Dhaliwal, Ying Zou, and Bram Adams. Do faster releases improve software quality?: an empirical case study of mozilla firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 179–188. IEEE Press, 2012.

[38] Foutse Khomh, Bram Adams, Tejinder Dhaliwal, and Ying Zou. Understanding the impact of rapid releases on software quality. *Empirical Software Engineering*, 20(2):336–373, 2015.

[39] Heinz K Klein and Michael D Myers. A set of principles for conducting and evaluating interpretive field studies in information systems. *MIS quarterly*, 23(1):67–94, 1999.

[40] H. Kniberg. Spotify engineering culture, 2014. URL https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1.

[41] H.I. Kniberg. Scaling agile@ spotify with tribes, squads, chapters & guilds., 2012. URL https://medium.com/productmanagement101/spotify-squad-framework-part-i-8f74bcfcd761.

[42] Sue Kong, Julie E Kendall, and Kenneth E Kendall. The challenge of improving software quality: Developers' beliefs about the contribution of agile practices. *AMCIS 2009 Proceedings*, page 148, 2009.

[43] Eero Laukkanen, Maria Paasivaara, Juha Itkonen, and Casper Lassenius. Comparison of release engineering practices in a large mature company and a startup. *Empirical Software Engineering*, pages 1–43, 2018.

[44] Thierry Lavoie and Ettore Merlo. How much really changes?: a case study of firefox version evolution using a clone detector. In *Proceedings of the 7th International Workshop on Software Clones*, pages 83–89. IEEE Press, 2013.

[45] Jingyue Li, Nils B Moe, and Tore Dybå. Transition from a plan-driven process to scrum: a longitudinal case study on software quality. In *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*, page 13. ACM, 2010.

[46] Zengyang Li, Paris Avgeriou, and Peng Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[47] Patrick Mader, Orlena Gotel, and Ilka Philippow. Motivation matters in the traceability trenches. In *2009 17th IEEE International Requirements Engineering Conference*, pages 143–148. IEEE, 2009.

[48] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[49] Mika V Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, 20(5):1384–1425, 2015.

[50] Matthias Marschall. Transforming a six month release cycle to continuous flow. In *Agile Conference (AGILE), 2007*, pages 395–400. IEEE, 2007.

[51] Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 2(4):308–320, 1976.

[52] Sam G McFarland. Effects of question order on survey responses. *Public Opinion Quarterly*, 45(2):208–215, 1981.

[53] Martin Michlmayr and Brian Fitzgerald. Time-based release management in free and open source (foss) projects. *International Journal of Open Source Software and Processes (IJOSSP)*, 4(1):1–19, 2012.

[54] Martin Michlmayr, Francis Hunt, and David Probert. Release management in free software projects: Practices and problems. In *IFIP International Conference on Open Source Systems*, pages 295–300. Springer, 2007.

[55] Martin Michlmayr, Brian Fitzgerald, and Klaas-Jan Stol. Why and how should open source projects adopt time-based releases? *IEEE Software*, 32(2):55–63, 2015.

[56] Shubhangi Nagpal and Anam Shadab. Literature review: Promises and challenges of devops. *University of Waterloo.[S22]*, 2014.

[57] Maleknaz Nayebi, Bram Adams, and Guenther Ruhe. Release practices for mobile apps–what do users and developers think? In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)*, volume 1, pages 552–562. IEEE, 2016.

[58] Kai Petersen and Claes Wohlin. The effect of moving from a plan-driven to an incremental software development approach with agile practices. *Empirical Software Engineering*, 15 (6):654–693, 2010.

[59] Rashmi Popli and Naresh Chauhan. Cost and effort estimation in agile software development. In *Optimization, Reliabilty, and Information Technology (ICROIT), 2014 International Conference on*, pages 57–61. IEEE, 2014.

[60] Adam Porter, Cemal Yilmaz, Atif M Memon, Arvind S Krishna, Douglas C Schmidt, and Aniruddha Gokhale. Techniques and processes for improving the quality and performance of open-source software. *Software Process: Improvement and Practice*, 11(2): 163–176, 2006.

[61] T. Preston-Werner. Semantic versioning 2.0.0., 2013. URL `http://semver.org/spec/v2.0.0.html`.

[62] S. Prince. The product managers' guide to continuous delivery and devops, 2016. URL `https://www.mindtheproduct.com/2016/02/what-the-hell-are-ci-cd-and-devops-a-cheatsheet-for-the-rest-of-us/`.

[63] Teade Punter, Marcus Ciolkowski, Bernd Freimut, and Isabel John. Conducting on-line surveys in software engineering. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, pages 80–88. IEEE, 2003.

[64] Md Tajmilur Rahman and Peter C Rigby. Release stabilization on linux and chrome. *IEEE Software*, 32(2):81–88, 2015.

[65] Patrick Rempel, Patrick Mçder, and Tobias Kuschke. An empirical study on project-specific traceability strategies. In *2013 21st IEEE International Requirements Engineering Conference (RE)*, pages 195–204. IEEE, 2013.

[66] Colin Robson. *Real world research: A resource for social scientists and practitioner-researchers*. Wiley-Blackwell, 2002.

[67] C. Rossi. Moving to mobile: The challenges of moving from web to mobile releases., 2014. URL https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1/.

[68] Julia Rubin and Martin Rinard. The challenges of staying together while moving fast: An exploratory study. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 982–993. IEEE, 2016.

[69] Günther Ruhe and Des Greer. Quantitative studies in software release planning under risk and resource constraints. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, pages 262–270. IEEE, 2003.

[70] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.

[71] Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on software engineering*, 25(4):557–572, 1999.

[72] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[73] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010. ISBN 0137035152, 9780137035151.

[74] Sabine Sonnentag, Felix C Brodbeck, Torsten Heinbokel, and Wolfgang Stolte. Stressor-burnout relationship in software development teams. *Journal of occupational and organizational psychology*, 67(4):327–341, 1994.

[75] Rodrigo Souza, Christina Chavez, and Roberto A Bittencourt. Do rapid releases affect bug reopening? a case study of firefox. In *2014 Brazilian Symposium on Software Engineering*, pages 31–40. IEEE, 2014.

[76] Rodrigo Souza, Christina Chavez, and Roberto A Bittencourt. Rapid releases and patch backouts: A software analytics approach. *IEEE Software*, 32(2):89–96, 2015.

[77] Stackify. Dev leaders compare continuous delivery vs. continuous deployment vs. continuous integration, 2017. URL https://stackify.com/continuous-delivery-vs-continuous-deployment-vs-continuous-integration/.

[78] Daniel Ståhl, Kristofer Hallén, and Jan Bosch. Achieving traceability in large scale continuous integration and delivery deployment, usage and validation of the eiffel framework. *Empirical Software Engineering*, 22(3):967–995, 2017.

[79] Chandrasekar Subramaniam, Ravi Sen, and Matthew L Nelson. Determinants of open source software project success: A longitudinal study. *Decision Support Systems*, 46(2): 576–585, 2009.

[80] LG Thomas, Stephen R Schach, Gillian Z Heller, and Jeff Offutt. Impact of release intervals on empirical research into software evolution, with application to the maintainability of linux. *IET software*, 3(1):58–66, 2009.

[81] Richard Torkar, Pau Minoves, and Janina Garrigós. Adopting free/libre/open source software practices, techniques and methods for industrial use. *Journal of the AIS*, 12(1), 2011.

[82] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 403–414. IEEE Press, 2015.

[83] Jennifer Wisdom and John W Creswell. Mixed methods: integrating quantitative and qualitative data collection and analysis while studying patient-centered medical home models. *Rockville: Agency for Healthcare Research and Quality*, 2013.