Releasing Fast and Slow

An Exploratory Case Study at ING

Kula, E.; Rastogi, Ayushi; Huijgens, Hennie; van Deursen, Arie; Gousios, Georgios

DOI
10.1145/3338906.3338978

Publication date
2019

Document Version
Accepted author manuscript

Published in
ESEC/FSE 2019

Releasing Fast and Slow: An Exploratory Case Study at ING

Elvan Kula
Delft University of Technology
Delft, The Netherlands
e.kula@student.tudelft.nl

Ayushi Rastogi
Delft University of Technology
Delft, The Netherlands
a.rastogi@tudelft.nl

Hennie Huijgens
ING
Amsterdam, The Netherlands
hennie.huijgens@ing.com

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
arie.vandeursen@tudelft.nl

Georgios Gousios
Delft University of Technology
Delft, The Netherlands
g.gousios@tudelft.nl

ABSTRACT

The appeal of delivering new features faster has led many software projects to adopt rapid releases. However, it is not well understood what the effects of this practice are. This paper 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 2 years of code quality data and 1 year of release delay data. Our research shows that: rapid releases are more commonly delayed than their non-rapid counterparts, however, rapid releases have shorter delays; rapid releases can be beneficial in terms of reviewing and user-perceived quality; rapidly released software tends to have a higher code churn, a higher test coverage and a lower average complexity; challenges in rapid releases are related to managing dependencies and certain code aspects, *e.g.* design debt.

CCS CONCEPTS

• **Software and its engineering → Software development process management**;

KEYWORDS

rapid release, release delay, software quality, technical debt

ACM Reference Format:
Elvan Kula, Ayushi Rastogi, Hennie Huijgens, Arie van Deursen, and Georgios Gousios. 2019. Releasing Fast and Slow: An Exploratory Case Study at ING. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338978

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338978

1 INTRODUCTION

In today's competitive business world, software companies must deliver new features and bug fixes *fast* to maintain sustained user involvement [1]. The appeal of delivering new features more quickly has led many software projects to change their development processes towards rapid release models [2]. Instead of working for months or years on a major new release, companies adopt rapid releases, *i.e.*, releases that are produced in relatively short release cycles that last a few days or weeks. The concept of rapid releases (RRs) [3] is a prevalent industrial practice that is changing how organizations develop and deliver software. Modern applications like Google Chrome [4], Spotify [5] 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 [6].

Previous work on RRs has analyzed the benefits and challenges of adopting shorter release cycles. RRs are claimed to offer a reduced time-to-market and faster user feedback [3]; releases become easier to plan due to their smaller scope [7]; end users benefit because they have faster access to functionality improvements and security updates [2]. Despite these benefits, previous research in the context of open source software (OSS) projects shows that RRs can negatively affect certain aspects of software quality. RRs often come at the expense of reduced software reliability [3, 8], accumulated technical debt [9] and increased time pressure [10].

As RRs are increasingly being adopted in open-source and commercial software [2], it is vital to understand their effects on the quality of released software. It is also important to examine how RRs relate to timing aspects in order to understand the cases in which they are appropriate. Therefore, the overall goal of our research is to explore the timing and quality characteristics of rapid release cycles in an industrial setting. By exploring RRs in industry, we can obtain valuable insights in what the urgent problems in the field are, and what data and techniques 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 the industry today.

We performed an exploratory case study of rapid releases at ING, a large Netherlands-based internationally operating bank that develops software solutions in-house. We identified 433 teams out of 611 software development teams at ING as *rapid teams*, *i.e.*, teams that release more often than others (release cycle time $\leq$ 3 weeks). The remaining 178 teams with a release cycle $>$ 3 weeks are termed

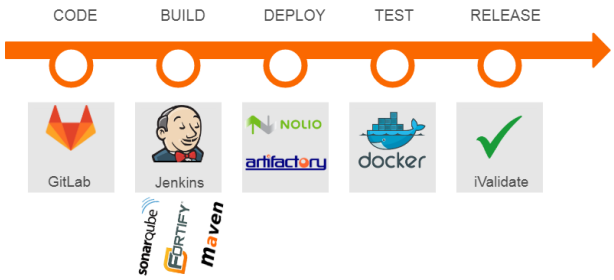

Figure 1: Continuous Delivery Pipeline at ING

as *non-rapid* teams, *i.e.*, teams that release less often than others. The large scale and mixed environment of ING allowed us to make a comparison between RRs and non-rapid releases (NRs) to explore how release cycle lengths relate to time and quality aspects of releases. We followed a mixed-methods approach in which we conducted a survey with 461 software engineers and corroborated their perceptions with 2 years of code quality data and 1 year of release delay data. To the best of our knowledge, this is the first exploratory study in the field of RRs. It is also the first study to analyze RRs at a scale of over 600 teams, contrasting them with NRs.

Developer answers to our survey indicate mixed perceptions of the effect of RRs on code quality. On one hand, RRs are perceived to simplify code reviewing and to improve the developers' 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 number of coding issues and a lower average complexity. Regarding release delays, our data analysis corroborates the belief of developers that RRs are more often delayed than NRs. However, RRs are correlated with a lower number of delay days per release than NRs. A prominent factor that is perceived to cause delay is related to dependencies, including infrastructural ones.

2 CONTEXT

ING is a large multinational financial organization with about 54,000 employees and over 37 million customers in more than 40 countries [11]. 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 2 to 3 months between major version releases. However, ING wanted to cut the cycles down to less than a month to stay ahead of competition. In 2011, the bank introduced DevOps teams to get developers and operators to collaborate in a more streamlined manner. Currently, ING has 611 globally distributed DevOps teams that work on various internal and external applications written in Java, JavaScript, Python, C and C#.

2.1 Time-based Release Strategy

All teams at ING work with a time-based release strategy, in which releases are planned for a specific date. In general, the teams deliver releases at regular week intervals. However, the release time interval differs across teams and occasionally within a team. The latter can appear in case of a release delay.

Defining RRs versus NRs. Although ING envisioned to cut release cycles down to less than a month, 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 that release at different speeds. Figure 2 presents an overview of the teams' release frequencies in the period from June 01, 2017 to June 01, 2018. The distribution shown is not fixed as teams intend to keep reducing their release cycle times in the future.

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.

Classification threshold. As all teams at ING are expected to follow the rapid release model, there is no specific culture of RRs versus NRs that enabled us to make a differentiation between the two (*e.g.*, letting teams self-identify). We decided to use the median release cycle time (3 weeks) as a point of reference since the distribution shown in Figure 2 contains outliers. Using the median as a point of reference, we classified teams as either *rapid* (release duration of $\leq$ 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. In the same manner, we classified releases as either *rapid* (time interval between release date and start of development phase $\leq$ 3 weeks) or *non-rapid* otherwise. 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.

Demographics of teams. Teams selected for analysis are similar in size (both rapid and non-rapid: 95% CI of 5 to 9 members) and number of respondents (both rapid and non-rapid: 95% CI of 1 to 2 respondents). Teams are also similar in their distribution of experience in software development (both rapid and non-rapid: 95% CI of 10 to 20 years). The projects are similar in size and domain.[1]

[1] A replication package containing survey questions and demographics data is publicly available at https://figshare.com/s/4b99fd1b849e4728c6ef.

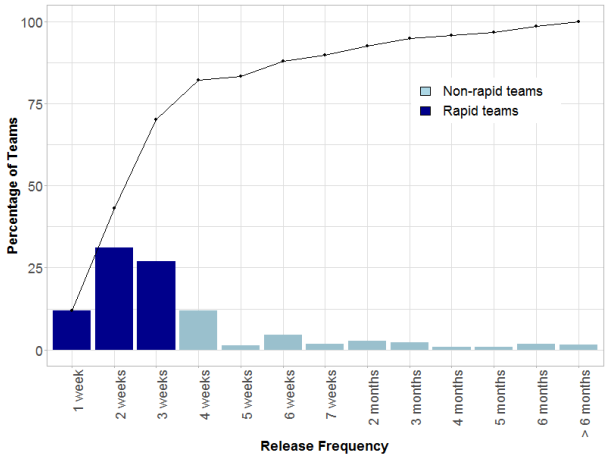

Figure 2: Distribution of release frequencies at ING: the black line represents the cumulative percentage of the teams.

2.2 DevOps and Continuous Delivery

To make shorter release cycles practical, ING introduced the *Continuous Delivery as a Service* (CDaaS) project in 2015 to automate the complete software delivery process. ING put a continuous delivery pipeline in place for all teams to enforce an agile development process, and reduce their testing and deployment effort. Figure 1 depicts the pipeline. *Jenkins*, the CI server, is responsible for monitoring the quality of the source code with the static analysis tool *SonarQube*.[2] Vassallo et al. [12] performed a case study at ING about the adoption of the delivery pipeline during development activities.

3 RESEARCH METHOD

The goal of this study is *to explore the timing and quality characteristics of rapid release cycles in an industrial setting.*

Timing characteristics. Because RRs are driven by the idea of reducing release cycle time, timing aspects are intrinsic to rapid 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. Teams at ING work with a time-based release strategy, in which releases are planned for a specific date. The only time teams deviate from their fixed cycle length is in case of a release delay. Here we want to find out how often projects deviate from their regular cycle length and why. This leads to our first two research questions:

> **RQ1:** *How often do rapid and non-rapid teams release software on time?*
>
> **RQ2:** *What factors are perceived to cause delay in rapid releases?*

Quality characteristics. It is important to examine the quality characteristics of rapid releases to get an insight into the way shorter release cycles affect the internal code quality and user-perceived quality of software in organizations. By exploring the quality characteristics of RRs, we may better understand their long-term consequences and 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:

> **RQ3:** *How do rapid release cycles affect code quality?*

3.1 Study Design

We conducted an exploratory case study [13] of rapid releases at ING with two units of analysis: the group of rapid teams and the group of non-rapid teams. Our case study combines characteristics from interpretive and positivist type case studies [13]. From an interpretive perspective, our study explores RRs through participants' interpretation of the development context at ING. From a positivist perspective, we draw inferences from a sample of participants to a stated population.

Data triangulation. To get a better understanding of RRs, we addressed the research questions applying data triangulation [14]. We combined qualitative survey data with quantitative data on release delays and code quality to present detailed insights. This is also known as a *mixed-methods* approach [15, 16]. 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 [17]. In the first phase, we ran a pilot study with two rapid and two non-rapid teams at ING. This allowed us to refine the survey questions. In the second phase, we sent the final survey to all teams at ING Netherlands (ING NL). In addition, we analyzed quantitative data stored in ServiceNow and SonarQube to examine the timing and quality characteristics of rapid releases, respectively.[3] We compared the perceptions of developers with release delay data and code quality data for rapid and non-rapid teams. An overview of our study set-up is shown in Figure 3. For RQ2, we only analyzed survey data because quantitative (proxy) data on release delay factors is not being collected by ING.

The quantitative data on release delays and code quality was aggregated at the release level (unless stated otherwise), 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.

3.2 Collecting Developers' Perceptions

We sent the survey to members of both rapid teams and non-rapid teams. To ensure that we collected data from a large number of diverse projects, we selected the members of all teams at ING NL as our population of candidate participants. In total, we contacted 1803 participants in more than 350 teams, 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 team, total years of work experience and total years at ING). The five sections were composed of open-ended questions, intermixed with multiple choice or Likert scale questions. To address RQ1, we asked respondents to fill in a compulsory multiple choice question on how often their team releases on time. To get deeper insights into how rapid versus non-rapid teams deal with release delays, we included an open-ended question asking respondents what their teams do when a release is delayed. For RQ2, we provided respondents with an open-ended question to gather unbounded and detailed responses on delay factors. For RQ3, 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 added a set of four optional 4-level Likert scale questions (each addressing the

[2] https://www.sonarqube.org/

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

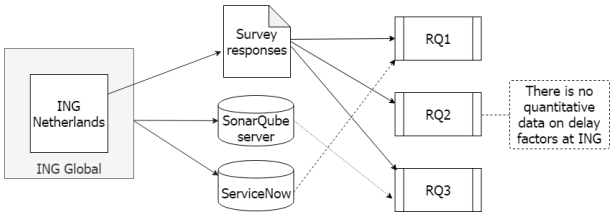

Figure 3: Overview of our mixed-methods study set-up

impact of RRs on testing debt, design debt, documentation debt and coding debt). In addition, we included a mandatory multiple choice question about the respondent team's release frequency and a few optional questions on how they perform quality monitoring.

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 and two non-rapid teams. We e-mailed the 24 employees in the four teams and received 7 responses (29% response rate). For the final survey, we e-mailed 1803 team members and obtained 461 responses (26% response rate). 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.

Demographics of respondents. Out of the 461 responses we received, 296 respondents were from rapid teams (64%) and the remaining 165 respondents were from non-rapid teams (36%) . A majority (70%) of our respondents self-identified as software engineer, while the rest identified themselves as managers (6%), analysts (23%) or other (1%) role at the IT department of ING. Most participants (77%) reported to have more than ten years of software development experience and more than five years of experience at ING (59%). For RQ3, we filtered out 259 respondents who did not identify as a software engineer in a rapid team.

Survey analysis. We applied manual coding [18] to summarize the results of the four open-ended questions during two integration rounds. We coded by statement and codes continued to emerge till the end of the process. In the first round, the first and the last author 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 first author then applied the integrated codes to 90% of the answers and the second author did this for the remaining 10% of the responses. In the second round, the first two authors 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.

3.3 Collecting Software Metrics

To analyze the quality of software written by non-rapid teams in comparison with rapid teams, we extracted the SonarQube measurements of releases that were shipped by teams that actively use SonarQube as part of the CDaaS pipeline. Although all teams at ING have access to SonarQube, 190 of them run the tool each time they ship a new release. We analyzed the releases shipped by these 190 teams 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 ($\leq$ 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.

Processing software metrics. 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 only considered the subset of metrics that are analyzed by all teams at ING. For each release, we extracted the metrics that all teams at ING measure to assess their coding performance. Out of these metrics, *code churn* is used to assess the level of coding activity within a code base, and the remaining metrics are seen as indicators for coding 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 [19]. As part of this class of metrics, we looked more specifically into the *Cyclomatic Complexity* [20] 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 indicate testing debt in the system [21].
(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 [21].
(4) *Code Churn:* the number of changed lines of code between two consecutive releases. Since in RRs code is released in smaller batches, it is expected that the absolute code churn is lower in rapid teams but it is not clear how the normalized code churn is influenced.

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. Since code churn is calculated over the time between releases and this differs among teams, 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. Finally, we performed a statistical comparison of the metrics between the group of RRs and the group of NRs.

3.4 Collecting Release Delay Data

To compare the occurrence and duration of delays in releases of rapid and non-rapid teams, we extracted log data from *ServiceNow*, a backlog management tool used by most teams at ING NL. We received access to the log data of 102 teams for releases shipped between October 01, 2017 and October 01, 2018. First, we checked the releases of each team in the system, and extracted their 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. We acknowledge that the development of a release might start before the planned release date of the previous release. This should not affect our distinction between releases as they are classified based

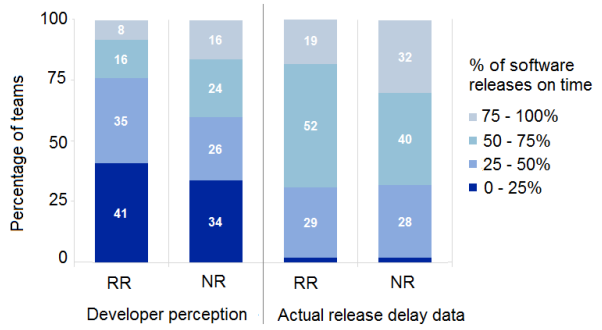

Figure 4: Percentage distribution of rapid and non-rapid teams based on the percentage of times they release software on time

on whether they were *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 to perform a statistical comparison of delays between rapid teams and non-rapid teams.

4 RESULTS

This section presents results on timing and quality characteristics of rapid releases derived from survey responses, release delay data and code quality data for rapid and non-rapid teams. Example quotes from the survey are marked with a [rX] notation, in which X refers to the corresponding respondent's identification number. The codes resulting from our manual coding process are underlined.

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

For this research question we looked into perceived delay data from survey responses and actual delay data from ServiceNow. The results are summarized in Figure 4. This figure shows a percentage distribution based on the percentage of times rapid and non-rapid teams perceive to and actually release software on time. Both the survey responses and delay data are aggregated at the team level.

A. Developers' Perceptions

For this survey question 85% of teams had 1 respondent, 9% had 2 respondents (both responses accepted) and remaining had 3 respondents (responses aggregated using majority vote). Using the Mann-Whitney U test [22], 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.

Figure 4 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. The percentage doubles to 16% 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%).

B. 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.

Figure 4 shows that a majority of both rapid and non-rapid teams release software more often on time than our respondents believe. We could not find any rapid or non-rapid team that releases software on time only 0 - 25% of the time. The data corroborates the perception of respondents that rapid teams are always 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.

Delay duration. Although rapid teams are more often delayed than non-rapid teams, analysis of the delays at release level shows that delays in RRs take a median time of 6 days, while taking 15 days (median) in NRs. 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.

Application domains. Further analysis of the data showed a similar trend as observed 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). Using the Mann-Whitney U test, we did not find any significant difference in project domains for the rapid teams that are on track more than 75% of the time.

How do teams address release delays? In the responses to the open-ended question on what teams do when a release gets delayed, we distinguished two main approaches that both rapid and non-rapid teams undertake. 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. Rapid teams mentioned both approaches equally often, while a majority (76%) of non-rapid teams report to reschedule. This suggests that rapid teams are more flexible regarding release delays.

> Rapid teams perceive to be, and in reality are, more commonly delayed than their non-rapid counterparts.

RQ2: What factors are perceived to cause delay 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. A list of these factors arranged in decreasing order of their occurrence in responses of rapid teams is shown in Figure 5.

Figure 5 shows that dependencies and infrastructure (which can be seen as a specific type of dependency) are the most prominent factors that are perceived to cause delay in rapid teams. Other factors which were listed in at least 10% of responses are testing (in general and for security), following mandatory procedures (such as for quality assurance) prior to every release, fixing bugs, and scheduling the release, including planning effort and resources. Non-rapid teams experience similar issues. 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 the respondents are scheduling, procedure, and security testing.

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. At a workflow level, developers mention to be hindered by task dependencies. Inconsistent schedules and unaligned priorities are perceived to cause delays in dependent teams. Many developers seem to struggle with estimating the impact of both types of dependencies in the release planning.

Another factor which is perceived to prominently affect rapid and non-rapid teams is security testing. For rapid teams, developers report that security tests are almost always delayed because of an unstable acceptance environment or missing release notes. They further add that any software release 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].

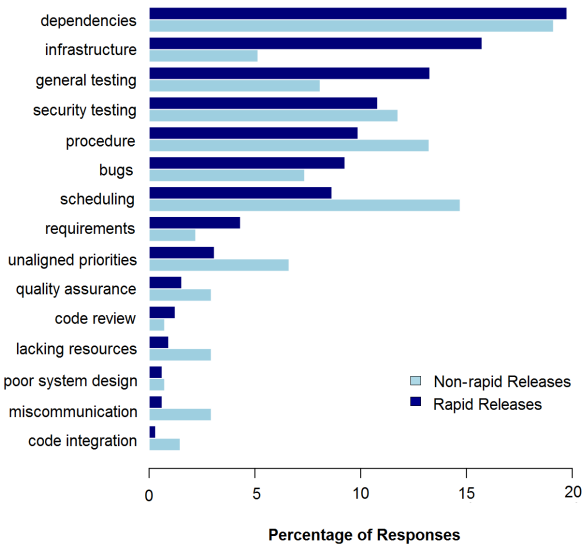

Figure 5: Factors perceived to cause delays in rapid and non-rapid teams

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*". Regarding testing, developers mention that the unavailability or 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*".

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 report 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.

> Dependencies, especially in infrastructure, and testing are the top mentioned delay factors in rapid releases.

RQ3: How do rapid release cycles affect code quality?

For this research question, we considered 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.

A. Developers' Perceptions

Developers have mixed opinions on how RRs affect code quality. A distribution of the effect of RRs (improve, degrade, no effect) on different factors related to code as perceived by developers is shown in Figure 6. It shows responses suggesting improvements in quality in green, degradation in quality in red and no effect in grey.

Quality improvement. A majority of developers perceive that the small changes in RRs 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 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 RRs motivate them to write modular and understandable code.

A large number of developers mention the benefits of rapid feedback in RRs. 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.*, "*[RRs] give more insight in bugs and issues after releasing. [They] enable us to respond more quickly to user requirements*" [r232], "*We can better monitor*

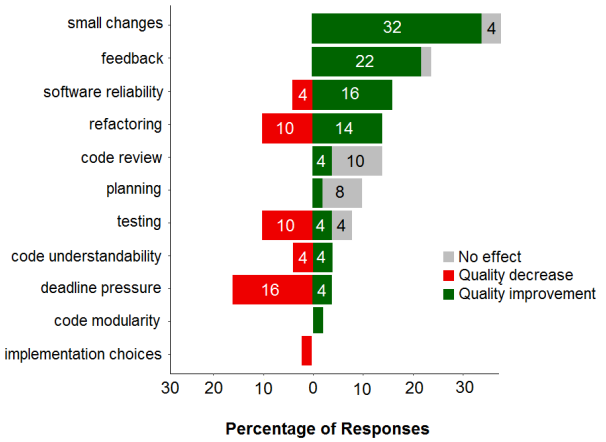

Figure 6: Developer perception of the impact of rapid releases on code quality aspects

the feedback of the customers which increased [with RRs]." [r130]). This enables teams to deliver customer value at a faster and more steady pace (*e.g.*, "*[With RRs] we can provide more value more often to end users.*" [r65], "*Features are delivered at a more steady pace*" [r16]).

> Developers perceive the smaller changes and rapid feedback in rapid releases to improve code quality.

Quality degradation. Many developers report to experience an increased deadline pressure in RRs, 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*" [143]. 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. [19]: *coding debt*, *design debt*, *testing debt* and *documentation debt*. An overview of the responses to the Likert scale questions is shown in Figure 7. According to a majority of the developers, RRs do not result in accumulated debt of any type. Since the developers' explanations on coding debt were similar to the factors mentioned in Figure 6, we will now focus on other types of debt:

Design debt. Many developers report that the short term focus of RRs 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, RRs 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 RRs to have both positive and negative effects on their testing effort. RRs 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 RRs 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 RRs.

Documentation debt. A majority of the developers do not perceive a decrease in the amount of documentation in RRs. However, developers report that the short-term focus in RRs 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*" [155], "*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.

> 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.

B. Software Quality Measurements

To gain a more detailed insight into the code quality of teams, we performed a comparative analysis of SonarQube measurements for RRs and NRs. To account for differences in project size, we normalized all metrics by SLOC. The Shapiro-Wilk test [23] shows that the data is not normally distributed. Therefore, we use the non-parametric statistical Mann-Whitney U test [22] to check whether the differences observed between NRs and RRs are statistically significant. To adjust for multiple comparisons we use the Bonferroni-corrected significance level [24] by dividing the conventional significance level of 0.05 by 5 (the total number of tests), giving a corrected significance level of 0.01. This means that the p-value needs to be < 0.01 to reject the null hypothesis. The effect size is measured as Cliff's delta [25]. The results of our analysis are summarized in Table 1, and presented below:

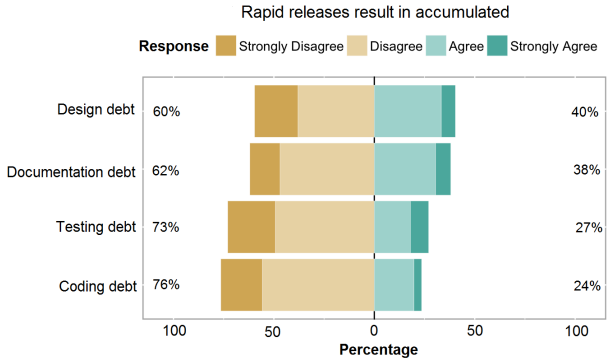

Figure 7: The impact of rapid release cycles on different types of technical debt

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
Cyclomatic Complexity*	0.14	0.17	0.15	0.16	0.00418	[0.13, 0.16]	[0.14, 0.20]	-0.336
Branch Coverage*	57.51	43.02	68.40	49.15	0.00016	[48.90, 76.70]	[27.40, 59.00]	0.286
Comment Density	10.13	11.07	7.20	8.30	0.04533	[4.79, 12.50]	[5.11, 15.00]	-0.109
Code Churn*	0.05	0.03	0.03	0.02	0.00782	[0.03, 0.07]	[0.02, 0.05]	0.263
SLOC	67066	83983	7447	9286	0.09045	[63600, 98900]	[55600, 78600]	-0.181

Table 1: Effects of RRs on software metrics. SLOC is used as a normalization factor. *P*-values are based on the Mann-Whitney U Test. * indicates statistical significance (p-value < 0.01, Bonferroni-corrected).

(1) ***The cyclomatic complexity and coding issues are significantly lower in software built with RRs (medium and large effect).*** 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 RRs. This makes it likely for them to write less complex code and fix issues more quickly.
(2) ***The branch coverage is significantly higher in software built with RRs (small effect).*** This result corresponds with the perceptions of developers on the testing effort. Developers report the test process in RRs to become more continuous and to allow for more complete testing of new features. As a consequence, RRs are likely to exhibit a higher test coverage.
(3) ***The code churn is significantly higher in software built with RRs (small effect).*** 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 RRs.
(4) ***We did not find a significant difference in the comment density of RRs and NRs.*** While not statistically significant, a lack of difference is consistent with the perceptions of developers on documentation debt. Developers perceive that RRs have an impact on the quality of documentation, but not on the amount (density) of documentation.

5 DISCUSSION

We now discuss our main findings and consider implications for practice. We also discuss several future research directions.

5.1 Main Findings

Delay factors in rapid releases. We found that testing and dependencies are the top mentioned delay factors in rapid teams. In addition, RRs in API and mobile app development are more often delayed than other application domains. These findings suggest that project type, or perhaps certain inherent characteristics in different project types, are a delay factor in RRs. A study of project properties that delay RRs could help the field determine when RRs are appropriate to use. Which project types and organizations fit well with RRs? Initial work in this direction has been carried out by Kerzazi & Khomh [26] and Bellomo et al. [27].

Total release delay and customer value. Our results show that RRs are more commonly delayed than NRs. However, it is not clear what this means for the project as a whole, given that NRs are correlated with a longer delay duration. How do RRs impact the total delay of a project? 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 also raises the question whether RRs (despite those delays) help companies to deliver more customer value in a timely manner. Our respondents report that RRs 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 in this direction could give us a better insight into the effectiveness of RRs in terms of customer experience.

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 lead time of software releases? In this context, Clark et al. [28] studied security vulnerabilities in Firefox and showed that RRs do not result in higher vulnerability rates. Further research in this direction is needed to clear up the interaction between both factors.

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 RRs.

Dependency management. We found that the timing of both RRs and NRs is perceived to be influenced by dependencies in the ecosystem of the organization. Our respondents report the difficulty of 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. [29, 30] 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. [31] 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 RRs can be beneficial in terms of code reviewing and user-perceived quality, even in large organizations working with hundreds of software development teams. This complements the findings reported by [2, 32, 33] on the ease of quality monitoring in RRs. Our quantitative data analysis shows that software built with RRs tends to have a higher branch coverage, a lower number of coding issues and a lower average complexity. Previous research [34] 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 RRs impact code quality metrics in other types of organizations and projects.

Challenges related to code aspects in RRs concern design debt and the risk of poor implementation choices due to deadline pressure and a short-term focus. This is in line with previous work [9, 10] 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 RRs, instead of RRs 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 RRs.

5.2 Implications for Practitioners

Here we present a set of 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 RRs on code quality. We noticed that most self-reported 'good' teams are doing regular code reviews and dedicate 25% of the time per sprint on refactoring. Teams that experience the downsides of RRs 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 analysis is 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.

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 RRs: [35, 36]), 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 and *Infrastructure as Code*, to the problem of delays in RRs.

Releasing fast and responsibly. Developers report to feel more deadline pressure in RRs, which can result in poor implementation choices and an increase in workarounds. This is also reported by previous work [2, 10]. Organizations should not view RRs 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 attempt to enhance the ways that software development teams communicate, coordinate and assess coding performance to enable organizations to release fast while maintaining high code quality.

5.3 Future Work

Our work reveals several future research directions.

Fine-grained analysis of release cycle length. An interesting opportunity for future work is to explore whether our findings still hold for more fine-grained groupings of weekly release intervals. Another promising opportunity is to explore possible confounding factors. Although we explored the role of several factors that are likely to affect release cycle time (see Section 2.1), further analysis is required to explore confounding factors. Participant observations suggest that customers' 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).

Long-term impact of RRs. An interesting opportunity for future work is to explore the long-term effect of RRs. By analyzing longitudinal data of quality measurements before and after teams switched to RRs, it can be measured how metrics relate to release cycle length over time. What is the long-term effect of RRs on code quality and user-perceived quality of releases? How do issues related to design debt and time pressure develop in the long run?

Feedback-driven development. Our results show that the rapid feedback in RRs 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 cycle length. An exploration of these opportunities would help organizations 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.

6 LIMITATIONS

Internal validity. One factor that can affect the qualitative analysis is the bias induced by the involvement of the authors with the studied organization. The first author interned at ING at the time of this study while the third author works at ING. To counter the biases which might have been introduced by the first and third authors, the last author (from Delft University of Technology) helped in designing survey questions. The observations and interpretation of the findings were cross-validated by the other two authors. Another 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 order effects. Social desirability bias [37] may have influenced the responses. To mitigate this risk, we made the survey anonymous and let the participants know that the responses would be evaluated statistically.

We cannot account for confounding variables that might have affected our findings. Even though all teams at ING are encouraged to release faster, not all teams have been able to reduce their release cycle length to 3 weeks or less. This suggests that there are confounding factors that differentiate rapid and non-rapid teams. Examples of potential factors are project difficulty and security requirements. It is also a possibility that rapid teams at ING work on software components that are more easy to release rapidly. This might have led to too optimistic results for rapid teams.

External validity. As our study only considers one organization, external threats are concerned with our ability to generalize our results. Although the results are obtained from a large, global organization and we control for variations using a large number of participants and projects spanning a time period of two years, 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 RRs. Our findings are likely applicable to organizations that are similar to ING in scale and security level. 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 organizations and projects relates to the findings. Our findings indicate a trade-off between rapid delivery and software security. 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 likely applicable to organizations with similar business- or safety-critical systems. Replication of this study in organizations of different scale, type and security level is therefore required.

7 RELATED WORK

Early studies on RRs focused on the motivations behind their adoption. Begel and Nagappan [38] found that the main motivations relate to easier planning, more rapid feedback and a greater focus on software quality. Our study and others [2, 32–34, 39, 40] found similar benefits. We also found that RRs are perceived to enable a faster and more steady delivery of customer value.

Recent efforts have examined the impact of switching from NRs to RRs on the *time* and *quality* aspects of the development process:

Time aspects. Costa et al. [8] found that issues are fixed faster in RRs but, surprisingly, RRs take a median of 54% longer to deliver fixed issues. This may be because NRs prioritize the integration of backlog issues, while RRs prioritize issues that were addressed during the current cycle [41]. Kerzazi and Khomh [26] studied the factors impacting the lead time of software releases and found that testing is the most time consuming activity along with socio-technical coordination. Our study complements prior work by exploring how often rapid teams release software on time and what the perceived causes of delay are. In line with [26], we found that testing is one of the top mentioned delay factors in RRs.

The strict release dates in RRs are claimed to increase the time pressure under which developers work. Rubin and Rinard [10] found that most developers in high-tech companies work under significant pressure to deliver new functionality quickly. Our study corroborates the finding that developers experience increased deadline pressure in RRs.

Quality aspects. Tufano et al. [9] found that deadline pressure for an approaching release date is one of the main causes for code smell introduction. Industrial case studies of Codabux and Williams [42], and Torkar et al. [43], showed that a rapid development speed is perceived to increase technical debt. We found that the deadline pressure in RRs is perceived to result in a lack of focus and an increase in workarounds. Our study complements prior work by analyzing the impact of RRs on certain code quality metrics and different types of debt.

In the OSS context, multiple studies [2, 32, 33, 39] have shown that RRs ease the monitoring of quality and motivate developers to deliver quality software. Khomh et al. [3, 44] found that less bugs are fixed in RRs, proportionally. Mäntylä et al. [2] showed that in RRs testing has a narrower scope that enables a deeper investigation of features and regressions with the highest risk. This was also found by our study and others [34, 39]. [34, 45] showed that testers of RRs lack time to perform time-intensive performance tests. In line with previous work, our respondents reported to spend less time on creating regression tests. Our study complements aforementioned studies by comparing the branch coverage of RRs to that of NRs.

Overall, studies that focus on RRs as main study target are explanatory and largely conducted in the context of OSS projects. In this paper, we present new knowledge by performing an exploratory case study of RRs in a large software-driven organization.

8 CONCLUSIONS

The goal of our paper is to deepen our understanding of the practices, effectiveness, and challenges surrounding rapid software releases in industry. To that end, we conducted an industrial case study, addressing timing and quality characteristics of rapid releases. Our contributions include the reusable setup of our study (Section 3) and the results of our study (Section 4).

The key findings of this study are: (1) Rapid teams are more often delayed than their non-rapid counterparts. However, rapid releases are correlated with a lower number of delay days per release. (2) Dependencies, especially in infrastructure, and testing are the top mentioned delay factors in rapid releases. (3) Rapid releases are perceived to make it easier to review code and to strengthen the developers' focus on user-perceived quality. The code quality data shows that rapid releases are correlated with a higher test coverage, a lower average complexity and a lower number of coding standard violations. (4) Developers perceive rapid releases to negatively impact implementation choices and design due to deadline pressure and a short-term focus.

Based on our findings we identified challenging areas calling for further attention, related to the applicability of rapid releases, the role of security concerns, the opportunities for rapid feedback, and management of code quality and dependencies. Progress in these areas is crucial in order to better realize the benefits of rapid releases in large software-driven organizations.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] H. Kniberg. Spotify engineering culture. *Available at https://labs.spotify.com/2014/03/27/spotify-engineering-culture-part-1*, 2014.

[6] C Rossi. Moving to mobile: The challenges of moving from web to mobile releases. *Keynote at RELENG*, 2014.

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

[8] 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.

[9] 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.

[10] 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.

[11] ING. 2017 Annual ING Group N.V. *Available at https://www.ing.com/About-us/Annual-reporting-suite/Annual-Report/2017-Annual-Report-Empowering-people.htm*, 2018.

[12] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. Continuous delivery practices in a large financial organization. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 519–528. IEEE, 2016.

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

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

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

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

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

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

[19] 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.

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

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

[22] 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.

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

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

[38] 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.

[39] 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.

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

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.