Empir Software Eng (2018) 23:835–904
https://doi.org/10.1007/s10664-017-9548-7

The impact of rapid release cycles on the integration delay of fixed issues

Daniel Alencar da Costa[1] · **Shane McIntosh**[2] · **Christoph Treude**[3] · **Uirá Kulesza**[4] · **Ahmed E. Hassan**[5]

Published online: 4 November 2017

Abstract The release frequency of software projects has increased in recent years. Adopters of so-called rapid releases—short release cycles, often on the order of weeks, days, or even hours—claim that they can deliver fixed issues (i.e., implemented bug fixes and new features) to users more quickly. However, there is little empirical evidence to support these claims. In fact, our prior work shows that code integration phases may introduce delays for rapidly releasing projects—98% of the fixed issues in the rapidly releasing Firefox project had their integration delayed by at least one release. To better understand the impact that rapid release cycles have on the integration delay of fixed issues, we perform

Communicated by: Romain Robbes, Christian Bird, and Emily Hill

✉ Daniel Alencar da Costa
daniel.alencar@queensu.ca

Shane McIntosh
shane.mcintosh@mcgill.ca

Christoph Treude
christoph.treude@adelaide.edu.au

Uirá Kulesza
uira@dimap.ufrn.br

Ahmed E. Hassan
ahmed@cs.queensu.ca

[1] Department of Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada

[2] Department of Electrical and Computer Engineering, McGill University, Montreal, QC, Canada

[3] School of Computer Science, University of Adelaide, Adelaide, South Australia, Australia

[4] Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte, Natal, Brazil

[5] Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, ON, Canada

a comparative study of traditional and rapid release cycles. Our comparative study has two parts: (i) a quantitative empirical analysis of 72,114 issue reports from the Firefox project, and a (ii) qualitative study involving 37 participants, who are contributors of the Firefox, Eclipse, and ArgoUML projects. Our study is divided into quantitative and qualitative analyses. Quantitative analyses reveal that, surprisingly, fixed issues take a median of 54% (57 days) longer to be integrated in rapid Firefox releases than the traditional ones. To investigate the factors that are related to integration delay in traditional and rapid release cycles, we train regression models that model whether a fixed issue will have its integration delayed or not. Our explanatory models achieve good discrimination (ROC areas of 0.80–0.84) and calibration scores (Brier scores of 0.05–0.16) for rapid and traditional releases. Our explanatory models indicate that (i) traditional releases prioritize the integration of backlog issues, while (ii) rapid releases prioritize issues that were fixed in the current release cycle. Complementary qualitative analyses reveal that participants' perception about integration delay is tightly related to activities that involve decision making, risk management, and team collaboration. Moreover, the allure of shipping fixed issues faster is a main motivator for adopting rapid release cycles among participants (although this motivation is not supported by our quantitative analysis). Furthermore, to explain why traditional releases deliver fixed issues more quickly, our participants point out the rush for integration in traditional releases and the increased time that is invested on polishing issues in rapid releases. Our results suggest that rapid release cycles may not be a silver bullet for the rapid delivery of new content to users. Instead, our results suggest that the benefits of rapid releases are increased software stability and user feedback.

Keywords Integration delay · Rapid releases · Mining software repositories · Release engineering

1 Introduction

To achieve sustained success, software projects must attract and retain the interest of users (Subramaniam et al. 2009). Since users will quickly lose interest in a stagnant software project, successful projects need to continuously provide exciting new features and fix bugs that are frustrating users.

Within the context of constantly evolving requirements (e.g., in *agile* development), approaches like eXtreme Programming (XP) and Scrum[1] have arisen to foster faster software delivery (Beck 2000). Those methodologies claim to better embrace a constantly evolving requirements context by shortening release cycles. Indeed, modern release cycles are on the order of days or weeks rather than months or years (Baskerville and Pries-Heje 2004). Such rapid releasing enables faster user feedback and a smoother roadmap for user adoption.

The allure of delivering new features faster has led many large software projects to shift from a more traditional release cycle (e.g., 12–18 months to ship a major release), to shorter release cycles (e.g., weeks). For example, Google Chrome, Mozilla Firefox, and Facebook teams have each adopted shorter release cycles (Adams and McIntosh 2016). In this paper, we use the term *rapid releases* to refer to releases that are produced in release cycles that last for weeks or days (such as a *sprint* in the Scrum agile process (Schwaber 1997)). Conversely, we use the term *traditional releases* to refer to releases that are produced in cycles that last for months or years.

[1] http://www.scrumguides.org/.

Prior research has investigated the impact of adopting rapid releases (Mäntylä et al. 2014; Souza et al. 2014, 2015; Baysal et al. 2011; Khomh et al. 2012). For example, Khomh et al. (2012) found that bugs that are related to crash reports tend to be fixed more quickly in the rapid Firefox releases than the traditional ones. Mäntylä et al. (2014) found that the Firefox project's shift from a traditional to a rapid release cycle has been accompanied by an increase in the testing workload.

To the best of our knowledge, little prior research has empirically studied the impact that a shift from a traditional to a rapid release cycle has on the speed of integration of fixed issues. Such an investigation is important to empirically check if adopting a rapid release cycle really does lead to the quicker delivery of fixed issues. In our previous work (da Costa et al. 2014), we studied the delay that is introduced by the integration phase of a software project. We found that 98% of the bug-fixes and new features in the rapid releases of Firefox were delayed by at least one release. Such delayed integration hints that even though rapid releases are consistently delivered every 6 weeks, they may not be delivering fixed issues as quickly as its proponents purport.

Hence, in this paper, we perform a two-part empirical study to compare traditional and rapid release cycles with respect to integration delay. The first part is a quantitative analysis of 72,114 issue reports from the Firefox project (34,673 for traditional releases and 37,441 for rapid releases). These issue reports refer to bugs, enhancements, and new features (Antoniol et al. 2008). In the second part, we set out to qualitatively analyze the integration delay of fixed issues by surveying 37 participants from the Firefox, Eclipse, and ArgoUML projects.

This paper is an extended version of our prior work (da Costa et al. 2016). We extend our prior study to add a new qualitative analysis that is comprised of:

- An analysis of survey data that we collect from 37 participants from the Firefox, Eclipse, and ArgoUML projects (RQ4-RQ6).
- An open-coding analysis of the open-ended questions of our survey (RQ4-RQ6).
- A quantitative analysis using the responses to the Likert-scale questions of our survey (RQ4).
- An analysis of the extent to which the perceived integration delay of our participants are in accordance with the collected quantitative data from our prior work (da Costa et al. 2014; da Costa et al. 2016) (RQ5).
- Follow-up interviews with participants who were willing to clarify and provide deeper explanations about their survey responses (RQ4-RQ6).

1.1 Quantitative Study

Our quantitative analysis focuses on the following research questions.

- ***RQ1: Are fixed issues integrated more quickly in rapid releases?*** Interestingly, we find that although issues are fixed more quickly in rapid releases, they tend to require a longer time to be integrated and released to users.
- ***RQ2: Why can traditional releases integrate fixed issues more quickly?*** We find that minor-traditional releases (i.e., releases of smaller scope that are shipped after a major version of the software) are a key reason as to why fixed issues tend to be integrated more quickly in traditional releases. In addition, we find that the length of the release cycles are roughly the same between traditional and rapid releases when considering both minor and major releases, with medians of 40 and 42 days, respectively.

– ***RQ3: Did the change in release strategy have an impact on the characteristics of delayed issues?*** Our models suggest that issues are queued up as a project backlog in traditional releases, while issues in rapid releases are queued up on a per release basis (i.e., a backlog per release cycle). Issues that are fixed early either in a project or release cycle backlog are less likely to be delayed.

1.2 Qualitative Study

Next, we survey 37 developers from the Firefox, Eclipse, and ArgoUML projects to study the perceived impact of integration delay and the shift from a traditional to a rapid release cycle. More specifically, we address the following research questions:

– ***RQ4: What are developers' perceptions as to why integration delays occur?*** The perceived reasons for the integration delay of fixed issues are related to decision making, team collaboration, and risk management activities. Moreover, integration delay will likely lead to user/developer frustration according to our participants.
– ***RQ5: What are developers' perceptions of shifting to a rapid release cycle?*** The allure of delivering fixed issues more quickly to users is the most recurrent motivator of switching to a rapid release cycle. Moreover, the allure of improving the flexibility and quality of fixed issues is another advantage that are perceived by our participants.
– ***RQ6: To what extent do developers agree with our quantitative findings about integration delay?*** The dependency of fixed issues on other projects and team workload are the main perceived explanations of our findings about integration delay in general. Integration rush and increased time spent on polishing fixed issues (during rapid releases) emerge as main explanations as to why traditional releases may achieve shorter integration delays.

Paper Organization The remainder of this paper is organized as follows. In Section 2, we present the necessary background and definitions to the reader. In Section 3, we describe the design of our quantitative and qualitative studies. In Sections 4 and 5, we present the results of our quantitative and qualitative studies, respectively. In Section 6, we analyze potential confounding factors that are related to our quantitative analysis. In Section 7, we suggest practical guidelines based on the results of our two studies. Section 8 discloses the threats to the validity of our studies, while we discuss the related work in Section 9. Finally, we draw conclusions in Section 10.

2 Background & Definitions

2.1 Issue Reports

An *issue report* describes a new feature, enhancement, or bug. Modern software projects use *Issue Tracking Systems* (ITSs, e.g., Bugzilla) to manage issues as they transition from being reported to being fixed.[2]

Each issue report has a unique identifier (issue ID), a description of the nature of the issue, and a variety of other metadata (e.g., the severity and priority of the issue).[3] Large

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

[3] https://bugzilla.readthedocs.org/en/5.0/using/understanding.html.

software projects receive plenty of issue reports on a daily basis. For example, our data shows that a median of 124 Firefox issues were opened on a daily basis from 1999 to 2010.

When developers start working on issue reports, they use the *issue status* to track progress throughout the lifetime of an issue. An issue is first (1) reported (*new* status), (2) triaged to an appropriate developer (*assigned* status), and (3) finally fixed (*fixed* status). A more detailed description of the life cycle of an issue report in the Firefox project is provided in the Bugzilla documentation.[4]

In this paper, we study *fixed issues*, which are issues that are resolved with the *fixed* status (i.e., the RESOLVED-FIXED status in the Bugzilla ITS) and integrated into traditional or rapid releases of the Firefox project.

2.2 Release Cycles

In this paper, we use the term *rapid releases* to refer to releases that are produced in *rapid release cycles*, i.e., cycles that last for weeks, days, or even hours. For example, the Scrum agile process uses the term *sprint* to refer to a rapid release cycle (Schwaber 1997). Also, we use the term *traditional releases* to refer to releases that are produced in cycles that last for several months or even years. Traditional release cycles resemble the classic *Spiral* model (Boehm 1988), in which the scope of a release is firmly fixed at the beginning of the release cycle the tasks within the scope should be completed by a fixed date. Conversely, rapid releases have a more flexible scope by shortening the duration of their release cycles (Schwaber 1997). Since a shorter release cycle exposes the release more quickly to customers, there are more windows for changing and re-prioritizing the release scope. In this way, shorter release cycles are claimed to better embrace changes in scope that occur throughout the development process.

In our quantitative study, we examine the popular Firefox web browser.[5] Firefox has approximately 18% of the worldwide market share of web browsers at the time that this paper was written.[6] Firefox is a fitting subject for our study because it shifted from a traditional release cycle to a rapid release cycle (Khomh et al. 2012; Souza et al. 2014, 2015; Mäntlä et al. 2014).

2.2.1 Firefox Traditional Releases

The traditional release cycle of Firefox was applied to major releases from 1.0 to 4.0. These traditional major releases took 12-18 months to be shipped.[7] In traditional release cycles, major traditional releases have subsequent minor releases containing bug fixes. These minor releases can be developed and released in parallel with major releases. Indeed, the final minor traditional release (3.6.24) was released in tandem with major rapid release 8.

2.2.2 Firefox Rapid Releases

Firefox began adopting a rapid release cycle in March 2011. The first official rapid release was shipped in June 2011. The development process of rapid releases differs from that

[4] https://bugzilla.readthedocs.org/en/5.0/using/editing.html#life-cycle-of-a-bug.

[5] https://www.mozilla.org/en-US/firefox/new/.

[6] https://clicky.com/marketshare/global/web-browsers/.

[7] https://en.wikipedia.org/wiki/Firefox_release_history.

of traditional releases. Rapid release cycles last for 6 weeks (42 days) in Firefox, while traditional release cycles last for 12 to 18 months. Also, the rapid release process consists of a *pipeline process*, i.e., a release is developed (or *trained*) through four stabilization channels. These channels are the NIGHTLY, AURORA, BETA, and RELEASE channels, respectively—each channel yielding more stable releases than its prior counterpart.

In the Firefox rapid release strategy, a release is shipped into the NIGHTLY channel every night. This NIGHTLY release incorporates the fixed issues that were integrated into the main code repository (`mozilla-central`).[8] Then, a release candidate in the NIGHTLY channel migrates to the AURORA and BETA channels to be stabilized. Once stabilized, an official release is broadcasted on the RELEASE channel. In the AURORA and BETA channels, the *Quality Assurance* team (QA) makes decisions about whether the code that was stabilized in these channels should be pushed to the next channel.[9] Code that was further stabilized in the BETA channel is pushed to the RELEASE channel. The rapid release strategy is able to produce new official releases (on the RELEASE channel) every six weeks because it allows for the development of consecutive releases that are migrated from one channel to another on a regular basis.

2.3 Major and Minor Releases

We also analyze *major* and *minor* releases in our study. Major releases are releases that are produced in official release cycles, which produce the official versions of a software project. For example, Firefox versions 25, 26, and 27 were produced by major release cycles. On the other hand, minor releases are off-schedule releases that are usually produced to fix specific bugs (e.g., security bugs) or provide specific enhancements (e.g., stability improvements). For example, Firefox 27.0.1 is a minor release that was shipped after Firefox had adopted rapid release cycles.[10]

The rapid release cycle of the Firefox project also includes minor releases that contain bug fixes and *Extended Support Releases* (ESR). ESRs are shipped to organizations/customers who cannot update their Firefox installations at the same pace at which the rapid releases are shipped.[11]

2.4 Integration Delay

Figure 1 shows how we compute integration delays. *Integration delay* is the time between the moment at which an issue is fixed ($T3$) and the moment at which this fix is released to end users ($T5$). We consider that an issue is fixed when it reaches the RESOLVED-FIXED status. We define two kinds of integration delay that we study in this work below.

Definition 1—Time Delay The *time delay* is the number of days between $T3$ and $T5$. For example, if the time in days between each T of Fig. 1 is 30 days, the integration delay in terms of days for issue-#1 is 60 days. We use Definition 1 to perform our analyses in RQ1-RQ2 and to gather developer feedback about these analyses in RQ6.

[8] https://hg.mozilla.org/mozilla-central/.

[9] http://mozilla.github.io/process-releases/draft/development_overview/.

[10] https://en.wikipedia.org/wiki/History_of_Firefox#Version_27.

[11] https://www.mozilla.org/en-US/firefox/organizations/faq/.

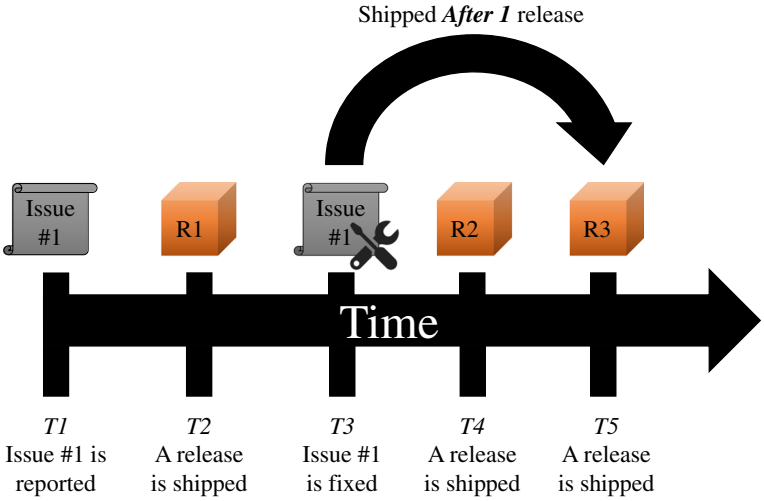


Fig. 1 An illustrative example of how we compute integration delays

Definition 2—Release Delay Instead of counting the number of days between $T3$ and $T5$ as we specify in Definition 1, we count the number of releases between $T3$ and $T5$. For example, the *release delay* for issue-#1 is *one release*. We use this definition to fit our statistical models in RQ3 and to understand the perceived reasons of why fixed issues are not integrated in RQ4-RQ6.

3 Empirical Study Design

We perform two studies: a quantitative and a qualitative study. In this section, we provide information about the subject projects, the motivation, and approach of the research questions for each of our studies.

3.1 Quantitative Study (Study I)

In Study I, we set out to comparatively analyze the integration delay of fixed issues that were shipped in traditional versus the ones that were shipped in rapid releases.

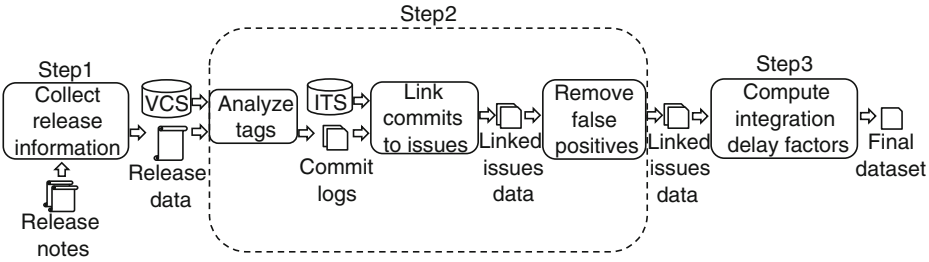


Fig. 2 Overview of the process to construct the dataset that is used in our Study I

Table 1 The studied traditional and rapid Firefox releases

Strategy	Version range	Time period	# of majors	# of minors
Trad.	1.0–4.0	Sep/2004–Mar/2012	7	104
Rapid	5–27	Jun/2011–Sep/2014	23	50

3.1.1 The Firefox Subject Project

We choose to study the Firefox project because it offers a unique opportunity to investigate the impact of shifting from a traditional release cycle to a rapid release cycle using rich, publicly available ITS and *Version Control System* (VCS) data. Although other open source projects may have ITS and VCS data available, they do not provide the opportunity to investigate the transition between traditional releases and rapid releases. In addition, comparing different projects that use traditional and rapid releases poses a great challenge, since one has to distinguish to what extent the results are due to the release strategy and not due to intricacies of the projects themselves. Therefore, we highlight that the choice to investigate Firefox is not accidental, but based on the specific analysis constraints that such data satisfies, and the very unique nature of such data.

3.1.2 Data Collection

Figure 2 shows an overview of our data collection approach. Each step of the process is described below.

Step 1: Collect Release Information We collect the date and version number of each Firefox release (minor and major releases of each release strategy) using the Firefox release history wiki.[12] Table 1 shows: (i) the range of versions of releases that we investigate, (ii) the investigated time period of each release strategy, and (iii) the number of major and minor studied releases in each release strategy. Release versions 1.0 to 4.0 are considered as traditional releases (i.e., they have a release cycle duration of 12 to 18 months), while release versions from 5 to 27 are considered as rapid releases (i.e., their release cycles are only 6 weeks long). The release version 4.0 was the last release that was produced in the traditional release cycle (Khomh et al. 2012; Souza et al. 2014, 2015; Mäntylä et al. 2014).

Step 2: Link Issues to Releases Once we collect the release information, we use the *tags* within the VCS to link issue IDs to releases. First, we analyze the tags that are recorded within the VCS. Since Firefox migrated from CVS to Mercurial during release 3.5, we collect the tags of releases 1.0 to 3.0 from CVS, while we collect the tags of releases 3.5 to 27 from Mercurial.[13,14] By analyzing the tags, we extract the commit logs within each tag. The extracted commit logs are linked to their respective tags. We then parse the commit logs to collect the issue IDs that are being fixed in these commits. By inspecting the commit logs, we notice that they mention issue IDs using the following patterns:

1. "Bug <ID>" or "bug <ID>" followed by a description of the fix.
2. "b=<ID>" followed by a description of the fix.

[12] https://en.wikipedia.org/wiki/Firefox_release_history.

[13] http://cvsbook.red-bean.com/cvsbook.html.

[14] https://mercurial.selenic.com/.

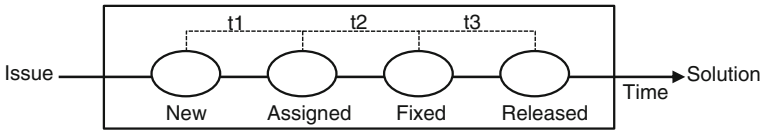


Fig. 3 A simplified life cycle of an issue

For example, the commits with IDs *bd0fdb3585c6* and *2e06eade69ce* are instances of the aforementioned patterns.[15,16]

In order to mitigate false positives, i.e., links between commit logs and issue IDs that should not exist, we discard the following patterns.

1. Potential IDs that have less than five digits, since the issue IDs of the investigated releases should have at least five digits (2,559 issues were discarded).
2. Commit logs that follow the pattern: "Bug <ID> reftest", "Bug <ID> JavaScript Tests", or "Bug <ID> crash tests", which refer to tests and not issue fixes (441 issues were discarded). We also include the "b=<ID>" variations in these cases.
3. Backout commits, i.e., commit logs that follow the pattern: "back out of bug <ID>" or "back out of b=<ID>" because these are reverting commits that are related to the specified IDs rather than fixes (168 issues were discarded) (Shimagaki et al. 2016; Souza et al. 2015).
4. Any potential ID that is the name of a file, e.g., "159334.js" (607 issues were discarded).

In total, we link 77% $\left(\frac{168,153}{217,245}\right)$ of the commit logs that are related to the traditional releases data, while we link 97% $\left(\frac{127,254}{130,136}\right)$ of the commit logs for the rapid releases data. These linkage rates suggest that the practice of providing issue IDs in commit logs has been more broadly adopted as the Firefox community has matured.

Since the commit logs are linked to VCS tags, we are also able to link the issue IDs found within these commit logs to the releases that correspond to those tags. For example, since we find the fix for issue 529404 in the commit log of tag 3.7a1, we link this issue ID to that release. We also merge together the data of development releases like 3.7a1 into the nearest minor or major release. For example, release 3.7a1 would be merged with release 4.0, since it is the next user-intended release after 3.7a1. In the case that a particular issue is found in the commit logs of multiple releases, we consider that particular issue to pertain to the earliest release that contains the last fix attempt (commit log), since that release is the first one to contain the complete fix for the issue. Finally, we collect the issue report information of each remaining issue (e.g., opening date, fix date, severity, priority, and description) using the ITS. Moreover, since the minor-rapid releases are *off-cycle releases*, in which fixed issues may skip being integrated into `mozilla-central` (i.e., NIGHTLY) tags, we manually collect the fixed issues that were integrated into those releases using the Firefox release notes (i.e., 247 fixed issues).[17] We add the manually collected fixed issues from ESR releases within the rapid releases data, since they also represent data from a rapid release strategy.

[15] https://hg.mozilla.org/mozilla-central/rev/bd0fdb3585c6.

[16] https://hg.mozilla.org/mozilla-central/rev/2e06eade69ce.

[17] https://www.mozilla.org/en-US/firefox/releases/.

Table 2 Metrics that are used in our explanatory models (reporter and resolver dimensions)

Family	Metrics	Value	Definition (d)|Rationale (r)
Reporter	Experience	Numerical	**d:** the number of previously integrated issues that were reported by the reporter of a particular fixed issue
			r: The greater the experience of the reporter the higher the quality of his/her reports and the solution to his/her reports might be integrated more quickly (Shihab et al. 2010)
	Reporter integration	Numerical	**d:** The median in days of the previously integrated fixed issues that were reported by a particular reporter
			r: If a particular reporter usually reports issues that are integrated quickly, his/her future reported issues might be integrated quickly as well
Resolver	Experience	Numerical	**d:** the number of previously integrated fixed issues that were fixed by the resolver of a particular fixed issue. We consider the collaborator that changed the status of an issue to RESOLVED-FIXED as the resolver of that issue
			r: The greater the experience of the resolver, the greater the likelihood that his/her code will be integrated faster (Shihab et al. 2010)
	Resolver integration	Numerical	**d:** The median in days of the previously integrated fixed issues that were fixed by a particular resolver
			r: If a particular resolver usually fixes issues that are integrated quickly, his/her future fixed issues might be integrated quickly as well

Step 3: Compute Metrics and Perform Analyses We use the data from Step 2 to compute the metrics that we use in our analyses. We select these metrics (which are described in Section 3.1.4) because we suspect that they share a relationship with integration delay (i.e., *time delay* and *release delay*). Finally, we use the collected dataset to perform the analyses of our study.

3.1.3 Research Questions

In our quantitative study, we address three research questions about the shift from a traditional to a rapid release cycle. The motivation of each research question is detailed below.

- **RQ1: Are fixed issues integrated more quickly in rapid releases?** Since there is a lack of empirical evidence to indicate that rapid release cycles integrate fixed issues more quickly than traditional release cycles, we compare the integration delay of fixed issues in traditional releases against the integration delay in rapid releases in RQ1.
- **RQ2: Why can traditional releases integrate fixed issues more quickly?** In RQ1, we surprisingly find that traditional releases tend to integrate fixed issues more quickly than rapid releases. This result raises the following question: why can a traditional release strategy, which has a longer release cycle, integrate fixed issues more quickly than a rapid release strategy?
- **RQ3: Did the change in the release strategy have an impact on the characteristics of delayed issues?** In RQ1 and RQ2, we study the differences between rapid and traditional releases with respect to integration delay. We find that although issues tend to be

fixed more quickly in rapid releases, they tend to wait longer to be integrated. We also find that the use of minor releases is the main reason as to why traditional releases may integrate fixed issues more quickly. In RQ3, we investigate what are the characteristics of each release strategy that are associated with integration delays. This important investigation sheds light on what may generate integration delays in each release strategy, so that projects are aware of the characteristics of rapid releases versus traditional releases before choosing to adopt one of these release strategies.

3.1.4 Research Approach

Figure 3 shows a simplified life cycle of an issue, which includes the triaging phase ($t1$), the fixing phase ($t2$), and the integration phase ($t3$). We compute the *time delay* in $t3$. The *lifetime* of an issue is composed of all three phases (from *new* to *released*). For RQ1, we first observe the lifetime of the issues of traditional and rapid releases. Next, we look at the time span of the *triaging*, *fixing*, and *integration* phases within the lifetime of an issue. In RQ2, we group traditional and rapid releases into major and minor releases and study their *time delay*.

Table 3 Metrics that are used in our explanatory models (issue dimension)

Family	Metrics	Value	Definition (d)|Rationale (r)
Issue	Stack trace attached	Dichotomous	**d:** We verify if the issue report has a stack trace attached in its description
			r: A stack trace attached may provide useful information regarding the cause of the issue, which may quicken the integration of the fixed issue (Schroter et al. 2010)
	Severity	Nominal	**d:** The severity level of the issue report. Issues with higher severity levels (e.g., blocking) might be integrated faster than other issues
			r: Panjer observed that the severity of an issue has a large effect on its time to be fixed in the Eclipse project (Panjer 2007)
	Priority	Nominal	**d:** The priority level of the issue report. Issues with higher priority levels (e.g., P1) might be integrated faster than other issues
			r: Higher priority issues will likely be integrated before lower priority issues
	Bug type	Dichotomous	**d:** A boolean identifying whether an issue is a security issue. Similar to Zaman et al. (2011), we consult the *Mozilla Foundation Security Advisory* to identify whether an issue is related to security problems.[a]
			r: Since the release history documentation of Firefox shows that minor releases are usually related to stability and security issues We investigate whether short delays are related to security issues
	Description size	Numerical	**d:** The number of words in the description of the issue
			r: Issues that are well described might be more easy to integrate than issues that are difficult to understand

[a]https://www.mozilla.org/en-US/security/advisories/

Table 4 Metrics that are used in our explanatory models (project dimension)

Family	Metrics	Value	Definition (d)|Rationale (r)
Project	Queue rank	Numerical	**d:** A rank number that represents the moment at which an issue is fixed compared to other fixed issues in the backlog. For instance, in a backlog that contains 500 issues, the first fixed issue has a rank of 1, while the last fixed issue has a rank of 500
			r: An issue with a high *queue rank* is a recently fixed issue. A fixed issue might be integrated faster/slower depending of its rank
	Cycle queue rank	Numerical	**d:** A rank number that represents the moment at which an issue is fixed compared to other fixed issues of the same release cycle. For example, in a release cycle that contains 300 fixed issues, the first fixed issue has a rank of 1, while the last one has a rank of 300
			r: An issue with a high *cycle queue rank* is a recently fixed issue compared to the others of the same release cycle. An issue fixed close to the upcoming release might be integrated faster
	Queue position	Continuous	**d:** $\frac{\text{queue rank}}{\text{all fixed issues}}$. The *queue rank* is divided by all the issues that are fixed by the end of the next release. A *queue position* close to 1 indicates that the issue was fixed recently compared to others in the backlog
			r: A fixed issue might be integrated faster/slower depending of its position
	Cycle queue position	Continuous	**d:** $\frac{\text{cycle queue rank}}{\text{fixed issues of the current cycle}}$. The *cycle queue rank* is divided by all of the fixed issues of the release cycle. A *cycle queue position* close to 1 indicates that the issue was fixed recently in the release cycle
			r: An issue that is fixed close to a upcoming release might be integrated faster

We use beanplots (Kampstra et al. 2008) to compare the distributions of our data. The vertical curves of beanplots summarize and compare the distributions of different datasets (see Fig. 10a). The higher the frequency of data within a particular value, the thicker the bean is plotted at that particular value on the y axis. We also use Mann-Whitney-Wilcoxon (MWW) tests (Wilks 2011) and Cliff's delta effect-size measures (Cliff 1993). MWW tests are non-parametric tests of the *null hypothesis* that two distributions come from the same population ($\alpha = 0.05$). On the other hand, Cliff's delta is a non-parametric effect-size measure to verify the difference in magnitude of one distribution compared to another distribution. The higher the value of the Cliff's delta, the greater the difference of values between distributions. For instance, if we obtain a significant p value but a small Cliff's delta, this means that although two distributions do not come from the same population their difference is not that large. A positive Cliff's delta indicates how much larger the values of the first distribution are, while a negative Cliff's delta indicates the inverse. Finally, we use the *Median Absolute Deviation* (MAD) (Howell 2005; Leys et al. 2013) as a measure of the variation of our distributions. The MAD is the median of the *absolute deviations* from one distribution's median. The higher the MAD, the greater is the variation of a distribution with respect to its median.

For RQ3, we build explanatory models (i.e., logistic regression models) for the traditional and rapid releases data using the metrics that are presented in Tables 2, 3, 4 and 5.

Table 5 Metrics that are used in our explanatory models (process dimension)

Family	Metrics	Value	Definition (d)|Rationale (r)
Process	Number of Impacted Files	Numerical	**d:** The number of files that are linked to an issue report
			r: An integration delay might be related to a high number of impacted files because more effort would be required to properly integrate the modifications (Jiang et al. 2013)
	Churn	Numerical	**d:** The sum of added lines plus the sum of deleted lines to fix the issue
			r: A higher churn suggests that a great amount of work was required to fix the issue, and hence, verifying the impact of integrating the modifications may also be difficult (Jiang et al. 2013; Nagappan and Ball 2005)
	Fix time	Numerical	**d:** Number of days between the date when the issue was triaged and the date that it was fixed (Giger et al. 2010)
			r: If an issue is fixed quickly, it may have a better chance to be integrated faster
	Number of activities	Numerical	**d:** An activity is an entry in the issue's history
			r: A high number of activities might indicate that much work was required to fix the issue, which may impact the integration of the issue into a release (Jiang et al. 2013)
	Number of comments	Numerical	**d:** The number of comments of an issue report
			r: A large number of comments might indicate the importance of an issue or the difficulty to understand it Giger et al. (2010), which might impact the integration delay (Jiang et al. 2013)
	Interval of comments	Numerical	**d:** The sum of the time intervals (hour) between comments divided by the total number of comments of an issue report
			r: A short *interval of comments* indicates that an intense discussion took place, which suggests that the issue is important. Hence, such an issue may be integrated faster
	Number of tosses	Numerical	**d:** The number of times that the assignee has changed
			r: Changes in the issue assignee might indicate that more than one developer have worked on the issue. Such issues may be more difficult to integrate, since different expertise from different developers might be required (Jeong et al. 2009; Jiang et al. 2013)

Our metrics are computed based on the date at which a given issue was fixed (i.e., the date of the last RESOLVED-FIXED status). For example, the *resolver experience* metric is the number of integrated fixes by that resolver that were made prior to the analzyed fix date. We model our response variable Y as $Y = 1$ for fixed issues that are delayed, i.e., missed at least one release before integration (da Costa et al. 2014) and $Y = 0$ otherwise (see release delay). Hence, our models are intended to explain why a given fixed issue has its integration delayed (i.e., $Y = 1$).

We follow the guidelines of Harrell (2001) for building explanatory regression models. Figure 4 provides an overview of the process that we use to build our models. First, we estimate the budget of degrees of freedom that we can spend on our models while having a low risk of overfitting (i.e., producing a model that is too specific to the training data to

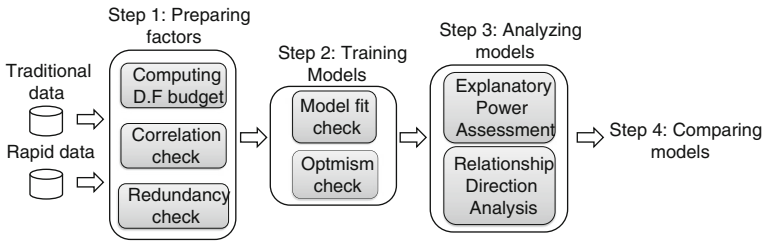


Fig. 4 Overview of the process that we use to build our explanatory models

be useful when applied to other unseen data). Second, we check for metrics that are highly correlated using Spearman rank correlation tests (ρ) and we perform a redundancy analysis to remove any redundant metrics before building our explanatory models.

We then assess the fit of our models using the ROC area and the Brier score. The ROC area is used to evaluate the degree of discrimination that is achieved by a model. The ROC values range between 0 (worst) and 1 (best). An area greater than 0.5 indicates that the explanatory model outperforms naïve random guessing models. The Brier score is used to evaluate the accuracy of probabilistic predictions. This score measures the mean squared difference between the probability of delay assigned by our models for a particular issue I and the actual outcome of I (i.e., whether I is actually delayed or not). Hence, the lower the Brier score, the more accurate the probabilities that are produced by a model.

Next, we assess the stability of our models by computing the *optimism-reduced* ROC area and Brier score (Efron 1986). The optimism of each metric is computed by selecting a bootstrap sample to fit a model with the same degrees of freedom of the original model. The model that is trained using the bootstrap sample is applied both on the bootstrap and original samples (ROC and Brier scores are computed for each sample). The optimism is the difference in the ROC area and Brier score of the bootstrap sample and original sample. This process is repeated 1,000 times and the average optimism is computed. Finally, we obtain the *optimism-reduced* scores by subtracting the average optimism from the initial ROC area and Brier score estimates (Efron 1986).

We evaluate the impact of each metric on the fitted models using Wald χ^2 maximum likelihood tests. The larger the χ^2 value, the larger the impact that a particular metric has on our explanatory models' performance. We also study the relationship that our metrics share with the likelihood of integration delay. To do so, we plot the change in the estimated probability of delay against the change in a given metric while holding the other metrics constant at their median values using the `Predict` function of the `rms` package (Harrell 2001).

We also plot nomograms (Iasonos et al. 2008; Harrell 2001) to evaluate the impact of the metrics in our models. Nomograms are user-friendly charts that visually represent explanatory models. For instance, Figure 15 shows the nomogram of the model that we fit for the rapid release data. The higher the number of points that are assigned to an explanatory metric on the x axis (e.g., 100 points are assigned to *comments* in rapid releases), the larger the effect of that metric in the explanatory model. We compare which metrics are more important in both traditional and rapid releases in order to better understand the differences between these release strategies.

3.2 Qualitative Study (Study II)

In Study II, we qualitatively analyze the integration delay phenomena by surveying and interviewing the team members of our subject projects.

3.2.1 Subject Projects

We analyze the Firefox, ArgoUML, and Eclipse (JDT) projects. We naturally choose Firefox, since the qualitative part of our study is intended to complement the quantitative analyses that we performed in the Firefox project. Furthermore, since data sources in qualitative studies should be selected with a focus on variation rather than representativeness (Sandelowski 1995), we also investigate the Eclipse and ArgoUML projects in this study. In particular, the Eclipse and ArgoUML projects are chosen based on (and to complement) our prior work (da Costa et al. 2014), in which we study general reasons for integration delay in these two projects.

ArgoUML is an open source UML modeling tool. ArgoUML provides support for all of the UML 1.4 diagrams. At the time that we perform this study, ArgoUML was downloaded 80,000 times worldwide.[18] ArgoUML uses the IssueZilla ITS to record its issue reports.[19]

Eclipse is a popular *Integrated Development Environment* (IDE) that is famous for its support for the Java programming language.[20] We study the *Java Development Tools* (JDT) project of the Eclipse Foundation.[21] The JDT project provides the Java perspective for the Eclipse IDE, which includes a number of views, editors, wizards, and builders.

ArgoUML and Eclipse (JDT) adopt a traditional release cycle when compared to the Firefox project. For instance, the median duration of release cycles that we study for the ArgoUML and Eclipse (JDT) projects are 180 and 112 days, respectively (da Costa et al. 2014). For these projects, we study the impressions of team members regarding the impact of a shift to a rapid release cycle on integration delays.

3.2.2 Data Collection

To perform our study, we searched for developers who contributed in the last four years of the studied projects.[22] Inspecting the code commits of the studied projects, we estimate that the population size of the Firefox, Eclipse (JDT), and ArgoUML projects are respectively of 1,011, 194, and 83 contributors in this time period. We contacted a group of these contributors who actively participated on the developer mailing lists of their projects in the last four years. To collect the data, we designed a web-based survey that was sent to 780 developers of the Firefox, Eclipse (JDT), and ArgoUML projects. We sent our survey to 513 Firefox, 184 Eclipse (JDT), and 62 ArgoUML developers whose e-mails existed on the mailing lists. To encourage participation, we provided $100 Amazon.com gift cards to a random subset of the respondents who answered all of the questions of our surveys.

Our survey has two major *themes*. The first *theme* is about integration delay in general, while the second *theme* is focused on the impact of switching to a rapid release cycle on

[18] argouml.tigris.org.

[19] http://argouml.tigris.org/project_bugs.html.

[20] https://eclipse.org/.

[21] https://projects.eclipse.org/projects/eclipse.jdt.

[22] From 2013 to 2016 by the time of this study.

the integration delay. Our complete surveys are available in Appendices A, B and C. The first three questions (#2–#4) collect demographic information. Questions #6–14 belong to the general integration delay *theme*, while questions #5, #17–18 belong to the impact of switching to a rapid release cycle *theme*. We placed one question of the second *theme* early in the survey to mitigate bias in the responses about the motivation to switch to a rapid release cycle. Finally, questions #17–19 are different for the Firefox project, since the other projects did not shift from a traditional to a rapid release cycle.

In total, we received 37 responses (5% response rate), of which 25 responses came from Firefox developers, 9 from Eclipse developers, and 3 from ArgoUML developers. We also conducted follow-up interviews with four of the Firefox participants to gather deeper insights into their responses. Our interviews were semi-structured and our goal was to clarify the responses of our survey and collect more details about specific cases of integration delays for fixed issues. Moreover, we provide our invitation letter and interview script in Appendices F and G, respectively.

3.2.3 Research Questions

We present the three research questions that are addressed in this qualitative *part* of the study below.

- **RQ4: What are developers' perceptions as to why integration delays occur?** To the best of our knowledge, there is no prior work that qualitatively studies integration delay. Qualitative studies are important to detect phenomena that are difficult to uncover quantitatively. Our goal in this RQ is to better understand *why* integration delays happen. This investigation is a starting point to reveal new ways of mitigating integration delays.
- **RQ5: What are developers' perceptions of shifting to a rapid release cycle?** In this research question, we intend to complement our quantitative findings about the comparison between traditional and rapid release cycles regarding integration delays. This investigation is important to gain deeper explanations as to why fixed issues may be integrated more quickly in traditional releases. Additionally, we intend to understand what are the reasons for the perceived success of adopting a rapid release cycle. This is also important to help projects with their decision of adopting a rapid release cycle rather than a traditional one.
- **RQ6: To what extent do developers agree with our quantitative findings about integration delay?** The main motivation for this research question is to solicit feedback about our quantitative findings. More specifically, we aim to understand to what extent our quantitative findings agree with the participants' perception of integration delays.

3.2.4 Research Approach

Given the exploratory nature of our qualitative analysis, we use methods from *Grounded Theory* (Charmaz 2014). The first and third authors independently conducted three sessions of open coding of the responses to open-ended questions (one session for each RQ). In the following, the codes that were generated were shared and merged into a new set of codes. The fourth author reviewed the set of codes and added additional entries to the final set of codes. At the end of the process, we achieved 175 unique codes. Finally, we used axial coding to find higher level conceptual themes to answer our RQs.

When reporting the results of RQ4-RQ5, we indicate in superscript the number of participants that mentioned a particular code that emerged during the qualitative analysis.

Table 6 Participant range per subject project

Project	Participant range
Firefox	F01–F25
Eclipse	E26–E34
ArgoUML	A35–A37

These numbers do not necessarily indicate the importance of a given code, since they were coded based on the received responses rather than scored by participants. Also, we mention quotes from the interviews when necessary to provide more detail about the results. Finally, Table 6 shows the IDs of the participants that we use while reporting results.

Finally, we also performed quantitative analyses of the responses to Likert-scale questions. First, we checked whether the factors that are listed in *question #13* were significantly different using the ranks (responses) that were assigned to each factor. In *question #13*, we present factors that reflect the metrics that we use to build the statistical models of our quantitative study (see Tables 2–5). The goal of this question is to verify whether our metrics are perceived as useful by the participants of our studied systems. We used a Kruskal Wallis test (Kruskal and Wallis 1952) to check whether there was a statistically significant difference between the ranks assigned to the factors. The Kruskal Wallis test is the non-parametric equivalent of the ANOVA test (Fisher 1925) to check whether there are statistically significant differences when comparing three or more distributions. Since Kruskal Wallis does not indicate which factor has statistically different values with respect to others, we use the Dunn test (Dunn 1964) to perform specific comparisons. For example, the Dunn test indicates whether the ranks that are assigned to the *number of comments* metric are statistically different when compared to the ones that are assigned to the *number of modified files* metric. We used the Bonferroni-Holm correction (Holm 1979) on the obtained p values to account for the multiple comparisons that we performed between each of the factors that are listed in *question #13*.

Additionally, we correlated the ranks that were assigned to the factors in *question #13* with the experience of the participants (*question #1*). To do that, we used Spearman rank ρ correlation (Spearman 1904), which is used to measure the statistical dependence between the ranks of two variables. Finally, we also correlated the experience of the participants with the perception of the frequency of integration delay (i.e., release delay) that happens in the studied projects (*question #6*).

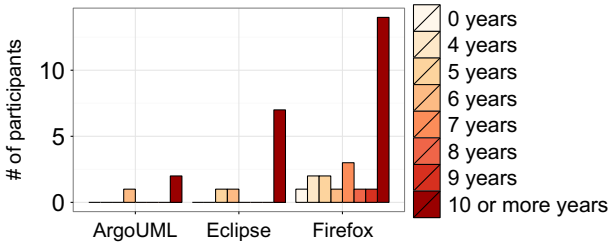


Fig. 5 Software development experience of the participants

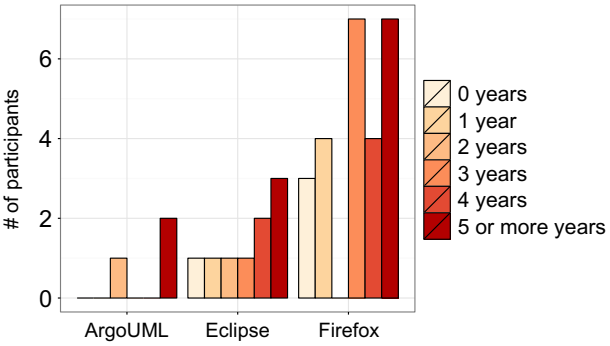


Fig. 6 Development experience of the participants in the respective project

3.2.5 Exploratory Analysis

We present an exploratory analysis of the data that we collect from the responses of the participants. Figure 5 shows the experience of the participants. We collect this data from *question #1*. The options range from "0 years" to "10 or more years". We observe that 62% ($\frac{23}{37}$) of the participants have "10 or more years" of software development experience. Furthermore, Fig. 6 shows the experience of the participants related to the specific project that they are representing. We collect this data from *question #2* and the options range from "0 years" to "5 or more years". 51% ($\frac{19}{37}$) of the participants have 4 or more years of experience. Moreover, Fig. 7 shows how many participants have experience in working on rapid release cycles (*question #14*). We note that 57% ($\frac{21}{37}$) of the participants have experience with rapid release cycles. Nevertheless, feedback from participants who do not have experience with rapid releases is also important, since this group may include prospective future adopters of a rapid release strategy without prior experience in such a strategy.

Figure 8 shows the team roles that the participants classified themselves as (*question #3*). The majority of the participants consider themselves as "developers" and "testers". Since one participant can occupy several roles, the numbers that are shown in Fig. 8 represent the frequency that a role was cited rather than the number of participants. Finally, we observe that the majority of the participants perceive integration delay as an unusual event rather

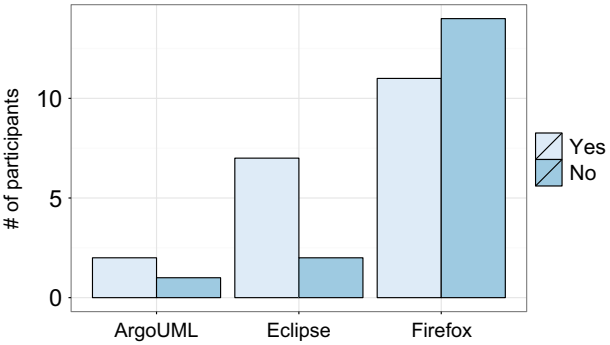


Fig. 7 Experience of the participants with respect to rapid release cycles

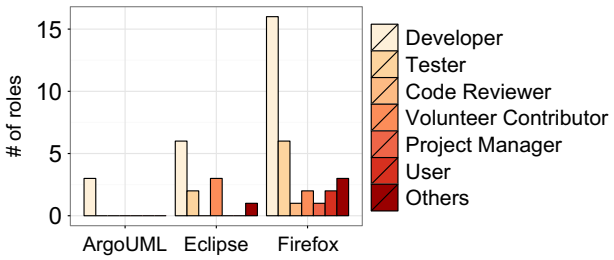


Fig. 8 An overview of the roles of the participants. One participant may have more than one role

than typical (see Fig. 9). For instance, 14 of the Firefox participants think that 90% of the issues are included in the next possible release.

In our analyses to answer RQ4-RQ6, we attempt to correlate the rating of factors that are provided in *question #13* with the data that is presented in this exploratory analysis.

4 Quantitative Study Results

In this section, we present the results of our quantitative study (RQ1-RQ3).

RQ1: are Fixed Issues Integrated More Quickly in Rapid Releases?

Observation 1—There is no Practical Difference Between Traditional and Rapid Releases Regarding Issue Lifetimes Figure 10a shows the distributions of the lifetime of the issues in traditional and rapid releases. We observe a $p < 1.03e^{-14}$ but a $negligible$ difference between the distributions ($delta = 0.03$). We also observe that traditional releases have a greater MAD (154 days) than rapid releases (29 days), which indicates that rapid releases are more consistent with respect to the lifetime of the issues. Our results indicate that the difference in the issues' lifetime between traditional and rapid releases is not as obvious as one might expect. We then look at the triaging, fixing, and integration time spans to better understand the differences between traditional and rapid releases.

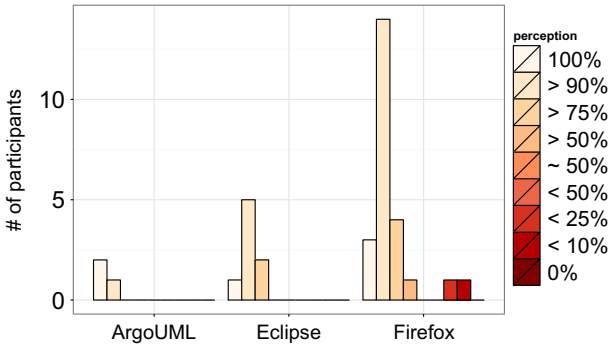


Fig. 9 Participants' perception on how frequent is integration delay. The data is grouped by proportions of how many fixed issues are included in the next possible release. This data refers to the responses to *question #6*

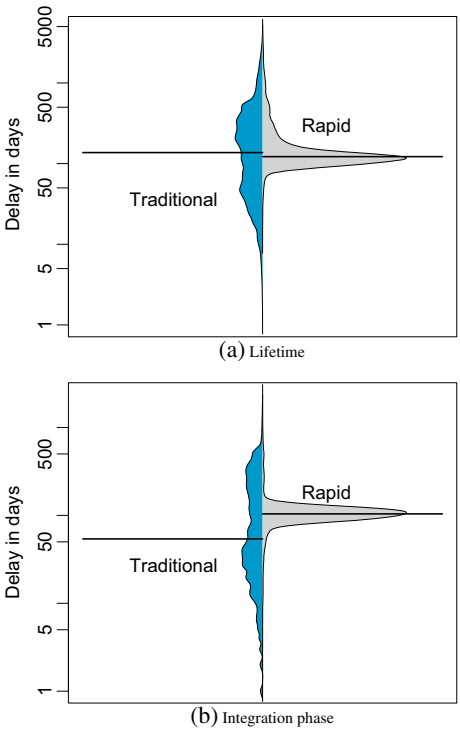


Fig. 10 Distributions of the lifetime and integration phase (*time delay*) of an issue

Observation 2—Fixed Issues are Triaged and Fixed More Quickly in Rapid Releases, but Tend to Wait for a Longer Time Before Being Released Figures 10b and 11a, b, show the triaging, fixing, and integration time spans, respectively. We observe that fixed issues take a median time of 54 days to be integrated into traditional releases, while taking 104 days (median) to be integrated into rapid releases. We observe a $p < 2.2e^{-16}$ with a *small* effect-size ($delta = -0.25$).

Regarding fixing time span, an issue takes 6 days (median) to be fixed in rapid releases, and 9 days (median) in traditional releases. These results are statistically significant $p < 2.2e^{-16}$, but not practically significant, i.e., the difference in magnitude between distributions is *negligible* ($delta = 0.13$).

Our results complement previous research. Khomh et al. (2012) found that post- and pre-release bugs that are associated with crash reports are fixed faster in rapid Firefox releases than in traditional releases. Furthermore, we observe a significant $p < 2.2e^{-16}$ but non-practical *negligible* difference ($delta = 0.11$) between traditional and rapid releases regarding triaging time. The median triaging time for rapid and traditional releases are 11 and 18 days, respectively.

When we consider both pre-integration phases together (triaging $t1$ plus fixing $t2$ in Fig. 3), we observe that an issue takes 11 days (median) to be triaged and fixed in rapid releases, while it takes 19 days (median) in traditional releases. We observe a $p < 2.2e^{-16}$ with a *small* effect-size ($delta = 0.15$). Our results suggest that even though issues have shorter pre-integration phases in rapid releases, they remain "on the shelf" for a longer time on average.

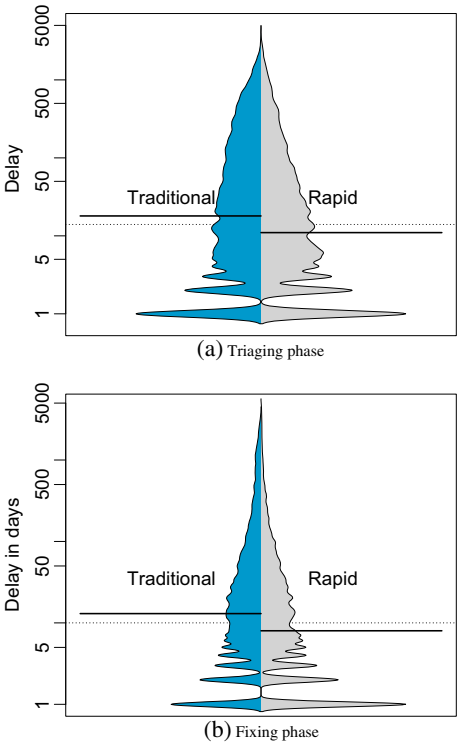


Fig. 11 Fixing and triaging phases of an issue's lifetime

(a) Triaging phase

(b) Fixing phase

Finally, we again observe that rapid releases are more consistent than traditional releases in terms of fixing and integration rate. Rapid releases achieve MADs of 9 and 17 days for fixing and integration, respectively. The values for traditional releases are 13 and 64 days for fixing and integration, respectively.

Although issues are triaged and fixed faster in rapid releases, they take a longer time to be integrated. However, the integration rate of fixed issues is more consistent in rapid releases than in traditional releases.

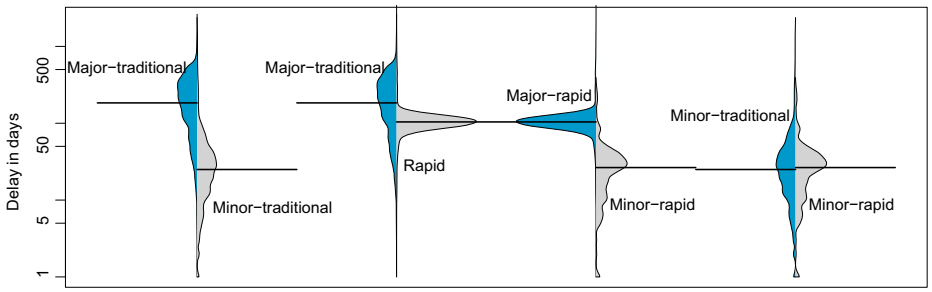
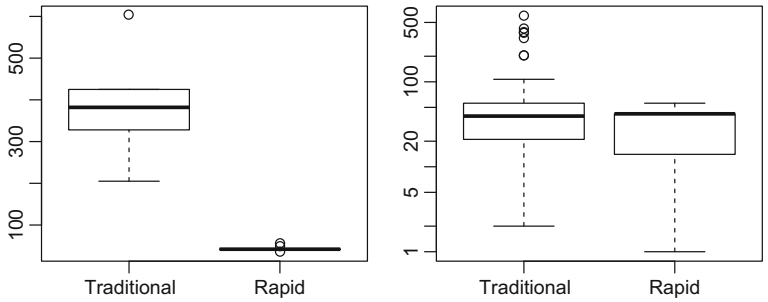


Fig. 12 Distributions of the *time delay* of fixed issues grouped by minor and major releases



(a) Only Major (b) Major and Minor

Fig. 13 Release frequency (in days). The outliers in figure (**b**) represent the major-traditional releases

RQ2: Why can Traditional Releases Integrate Fixed Issues More Quickly?

Observation 3—Minor-Traditional Releases Tend to have Less Integration Delay than Major/Minor-Rapid Releases Figure 12 shows the distributions of integration delay (i.e., *time delay*) grouped by (1) *major-traditional vs. minor-traditional*, (2) *major-traditional vs. rapid*, (3) *major-rapid vs. minor-rapid*, and (4) *minor-traditional vs. minor-rapid*. In the comparison of *major-traditional vs. minor-traditional*, we observe that minor-traditional releases are mainly associated with a shorter integration delay. Furthermore, in the comparison *major-traditional vs. rapid*, rapid releases integrate fixed issues more quickly than major-traditional releases on average ($p < 2.2e^{-16}$ with a $medium$ effect-size, i.e., $delta = 0.40$).

The Firefox rapid release cycle includes ESR releases (see Section 2) and a few minor stabilization and security releases. These releases also integrate fixed issues more quickly than major-rapid releases (*major-rapid vs. minor-rapid*) with a $p < 2.2e^{-16}$ and a $large$ effect-size, i.e., $delta = 0.92$. Furthermore, we do not observe a statistically significant difference between distributions in the comparison of *minor-traditional vs. minor-rapid* ($p = 0.68$).

Minor-traditional releases have the lowest integration delay (median of 25 days). This is likely because they are more focused on a particular set of issues that, once fixed, should be released immediately. For example, the release history documentation of Firefox shows that minor releases are usually related to stability and security issues.[23]

Observation 4—When Considering both Minor and Major Releases, the Time Interval Between Releases in Traditional and Rapid Strategies are Roughly the Same Since we observe that integration delay is shorter on average in traditional releases, we also investigate the length of the release cycles to better understand our previous results (see Observation 2). Figure 13a shows that, at first glance, one may speculate that rapid releases should deliver fixed issues more quickly because releases are produced more frequently. However, if we consider both major and minor releases—as shown in Fig. 13b—we observe that both release strategies deliver releases at roughly the same rate on average (median of 40 and 42 days for traditional and rapid releases, respectively).

[23] https://www.mozilla.org/en-US/firefox/releases/.

Minor-traditional releases are a key reason as to why traditional releases may integrate fixed issues more quickly than rapid releases. Furthermore, the time duration of the release cycles are roughly the same between traditional and rapid releases when minor and major releases are analyzed.

RQ3: Did the Change in the Release Strategy have an Impact on the Characteristics of Delayed Issues?

Observation 5—Our Models Achieve a Brier Score of 0.05-0.16 and ROC Areas of 0.81–0.83 The models that we fit to traditional releases achieve a Brier score of 0.16 and an ROC area of 0.83, while the models that we fit to the rapid release data achieve a Brier score of 0.05 and an ROC area of 0.81. Our models outperform naïve approaches such as random guessing and ZeroR—our ZeroR models achieve ROC areas of 0.5 and Brier scores of 0.06 and 0.45 for rapid and traditional releases, respectively. Moreover, the bootstrap-calculated optimism is less than 0.01 for both the ROC areas and Brier scores of our models. This result shows that our regression models are stable enough to perform the statistical inferences that follow.

Observation 6—Traditional releases prioritize the integration of backlog issues, while rapid releases prioritize the integration of issues of the current release cycle Table 7 shows the explanatory power (χ^2) of each metric that we use in our models. The *queue rank* metric is the most important metric in the models that we fit to the traditional release data. Queue rank measures the moment when an issue is fixed in the backlog of the project (see Table 4). Figure 14a shows the relationship that queue rank shares with

Table 7 Overview of the regression model fits. The χ^2 of each metric is shown as the proportion in relation to the total χ^2 of the model

		Traditional releases	Rapid releases
# of instances		34,673	37,441
Wald χ^2		4,964	2,705
Budgeted Degrees of Freedom		1033	149
Degrees of Freedom Spent		27	26
Reporter experience	D.F.	1	1
	χ^2	2^{***}	2^{***}
Reporter integration	D.F.	1	1
	χ^2	5^{***}	4^{***}
Resolver Experience	D.F.	1	⊘
	χ^2	1^{***}	
Resolver integration	D.F.	1	1
	χ^2	2^{***}	5^{***}
Fix time	D.F.	1	1
	χ^2	≈ 0	≈ 0
Number of files	D.F.	1	1
	χ^2	1^{***}	1^{***}
Number of comments	D.F.	1	1

Table 7 (continued)

		Traditional releases	Rapid releases
# of instances		34,673	37,441
Wald χ^2		4,964	2,705
Budgeted Degrees of Freedom		1033	149
Degrees of Freedom Spent		27	26
	χ^2	$\approx 0^{*}$	31^{***}
Number of tossing	D.F.	1	1
	χ^2	$\approx 0^{***}$	≈ 0
Number of activities	D.F.	1	1
	χ^2	1^{***}	3^{***}
Interval of comments	D.F.	$\oslash$	$\oslash$
	χ^2		
Code churn	D.F.	1	1
	χ^2	≈ 0	≈ 0
Queue position	D.F.	1	1
	χ^2	17^{***}	2^{***}
Queue rank	D.F.	1	1
	χ^2	56^{***}	14^{***}
Cycle queue rank	D.F.	1	1
	χ^2	≈ 0	≈ 0
Number of files	D.F.	1	1
	χ^2	1^{***}	1^{***}
Number of comments	D.F.	1	1
	χ^2	$\approx 0^{*}$	31^{***}
Number of tossing	D.F.	1	1
	χ^2	$\approx 0^{***}$	≈ 0
Number of activities	D.F.	1	1
	χ^2	1^{***}	3^{***}
Interval of comments	D.F.	$\oslash$	$\oslash$
	χ^2		
Code churn	D.F.	1	1
	χ^2	≈ 0	≈ 0
Queue position	D.F.	1	1
	χ^2	17^{***}	2^{***}
Queue rank	D.F.	1	1
	χ^2	56^{***}	14^{***}
Cycle queue rank	D.F.	1	1
	χ^2	10^{***}	28^{***}
Cycle queue position	D.F.	$\oplus$	$\oslash$
	χ^2		

$\oslash$ discarded during correlation analysis
$\oplus$ discarded during redundancy analysis
$* \; p < 0.05$; $** \; p < 0.01$; $*** \; p < 0.001$

integration delay (i.e., *release delay*). Our models reveal that the fixed issues in traditional releases have a higher likelihood of being delayed if they are fixed later when compared to other issues in the backlog of the project.

On the other hand, *cycle queue rank* is the second-most important metric in the models that we fit to the rapid release data. Cycle queue rank is the moment when an issue is fixed in a given release cycle. Figure 14b shows the relationship that cycle queue rank shares with integration delay. Our models reveal that the fixed issues in rapid releases have a higher likelihood of being delayed if they were fixed later than other fixed issues in the *current release cycle*. Interestingly, we observe that the most important metric in our rapid release models is the *number of comments*. Figure 14c shows the relationship that the *number of comments* shares with integration delay. We observe that the greater the number of comments of a fixed issue, the greater the likelihood of integration delay. This result corroborates the intuition that a lengthy discussion might be indicative of a complex issue, which may be more likely to be delayed.

Moreover, Figs. 15 and 16 show the estimated effect of our metrics using nomograms (Iasonos et al. 2008). Indeed, our nomograms reiterate the large impact of *number of comments* (100 points) and *cycle queue rank* (84 points) in rapid releases, and the large impact of *queue rank* (100 points) in traditional releases. We also observe that *stack trace attached* has a large impact on traditional releases (68 points) despite not being a significant contributor to the fit of our models (cf. Table 7). The large impact shown in our nomogram for *stack trace attached* is due to the skewness of our data—only 5 instances within the traditional release data have the *stack trace attached* set to true. Thus, *stack trace attached* cannot significantly contribute to the overall fit of our models.

Another *key* difference between traditional and rapid releases is how fixed issues are prioritized for integration. Traditional releases are analogous to a queue in which the earlier an issue is fixed, the lower its likelihood of delay. On the other hand, rapid releases are analogous to a stack of cycles, in which the earlier an issue is fixed in the current cycle, the lower its likelihood of delay.

Issues that are fixed earlier in the project backlog are less likely to be delayed in traditional releases. On the other hand, issues in rapid releases are queued up on a per release basis. In rapid releases, issues that are fixed earlier in the current release cycle are less likely to be delayed.

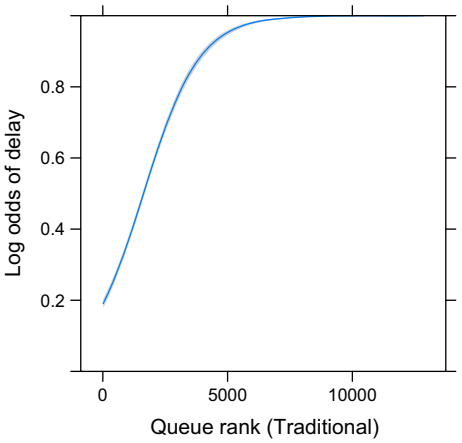
5 Qualitative Study Results

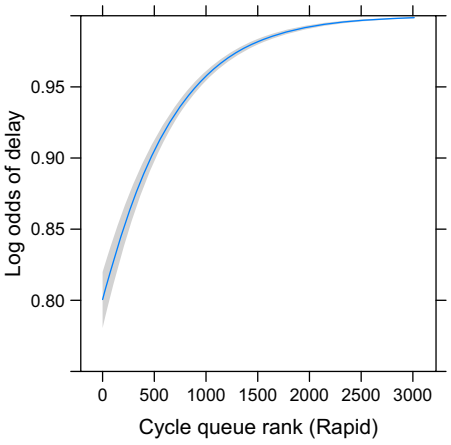
In this section, we present the results of our qualitative study (RQ4-RQ6).

RQ4: What are developers' perceptions of why integration delay happens?

Our findings about developers' perceptions of the causes of integration delay is divided into the following themes: (i) *development activities*, (ii) *decision making*, (iii) *risk*, (iv) *frustration*, and (v) *team collaboration*. After discussing each theme below, we present a quantitative analysis of the factors that can impact integration delay using the responses to *question #13* (see Section 3.2.2).

Theme 1—Development activities The number of tests that should be executed was a recurrent theme among participants. For instance, several participants stated that *additional*





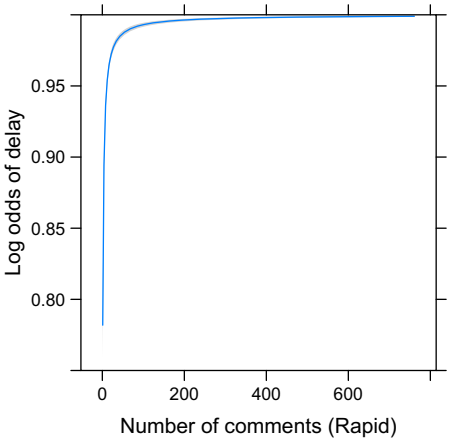


Fig. 14 The relationship between metrics and the *release delay*. The blue line shows the values of our model fit, whereas the grey area shows the 95% confidence interval based on models fit to 1,000 bootstrap samples. The parentheses indicate the release strategy that the metric is related to

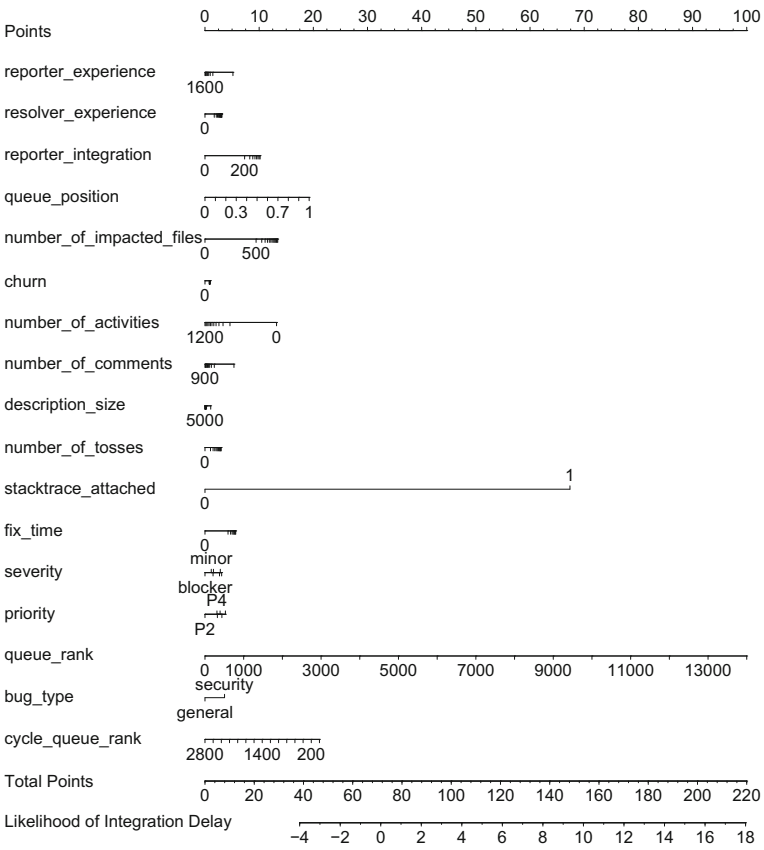


Fig. 15 Nomogram of our explanatory models for the traditional release cycle

testing(12) should be executed in order to avoid integration delay. *F17* states that the lack of *"actual user testing beyond what QA can provide"* can lead to integration delay. Additionally, according to *F15*, *"the most common reason is that testing was incomplete"* and according to *F19*, integration delay may happen because *"testing has been too narrow"*. Finally, *E32* voices concerns about integration testing: *"No integration tests has been done."* Such observations bring us back to a core software engineering problem of when is testing sufficient? (Beller et al. 2015; AlGhamdi et al. 2016).

Other recurrent themes that emerged during our qualitative analysis are *workload*(7) and *code review*.(7) For example, *E30* states that *"As the delayed completed issues stack up, they are harder to integrate (the codebase is constantly changing, merge issues might emerge)."* Interestingly, our statistical models in our prior work (da Costa et al. 2014) indicate *workload*(7) as a metric that shares a strong relationship with integration delay. As for *code review*,(7) the *"Unavailability of the lead/reviewer/[Project Management Committee] (PMC)"* is a reason of integration delay that is pointed out by *E26*, while *F08* argues that a *"prompt code reviews [may] help"* to avoid integration delays (McIntosh et al. 2016).

Theme 2—Decision making Decision making refers to the activities that are not directly related to software construction, but can influence the speed at which software is shipped.

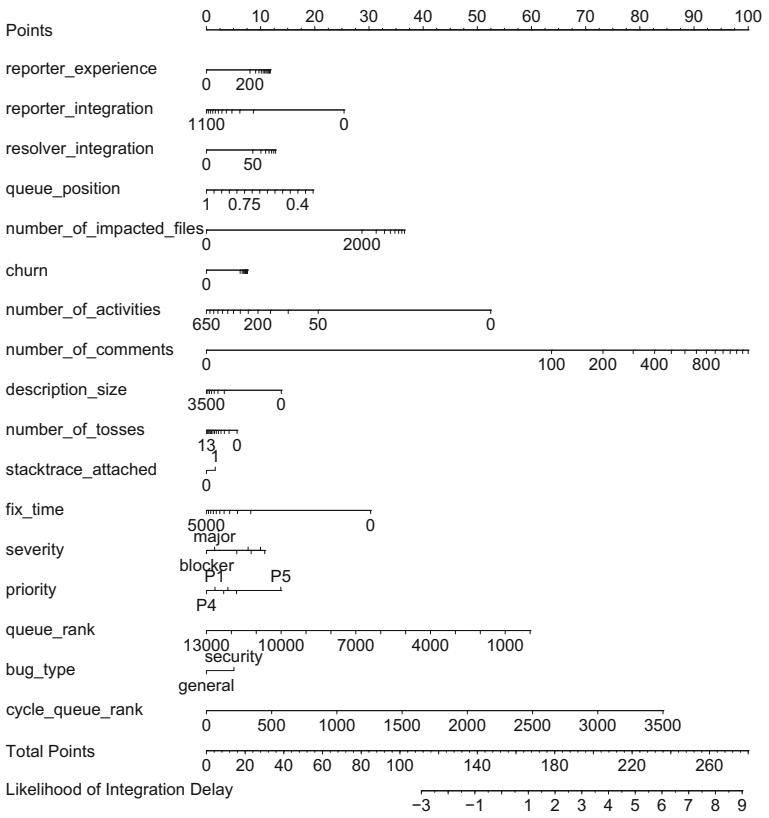


Fig. 16 Nomogram of our explanatory models for the rapid release cycle

For example, how early a codebase should be "frozen"? Which issues should be prioritized? The *timing*(9) and *prioritization*(9) are the recurrent themes in our survey responses. For instance, two of the participants stated that issues can be delayed because they are fixed *"too late in the release cycle"* (*E28*) or because they were fixed in a *"long release cycle."* Also, *F12*'s opinion about how to avoid integration delay is to *"test [fixed issues] early using real users (e.g., on the pre-release channels)."* Regarding *prioritization*,(9) *E28* argues that team members should *"try to complete most important things early in the release cycle"* to avoid integration delay. Additionally, *F07* points out how re-prioritization of issues is important: *"[...] prioritizing and re-prioritizing tasks to be sure you are building things on time [...]."*

Theme 3—Risk The risk that is associated with shipping fixed issues may generate integration delay according to our participants. Among the risky fixed issues, the ones that have *compatibility*(12) concerns are the most recurrent in this theme. For example, when asked about reasons that may lead to integration delay, *F12* calls attention to issues that *"break third-party websites"* and that can generate *"incompatibility with third-party software that users install."* Another risk that is associated with integration delay is *stability*.(9) For instance, *F03* states that *"when there are regressions noticed during Aurora/Beta cycles,"* a fixed issue will likely skip the upcoming official release.

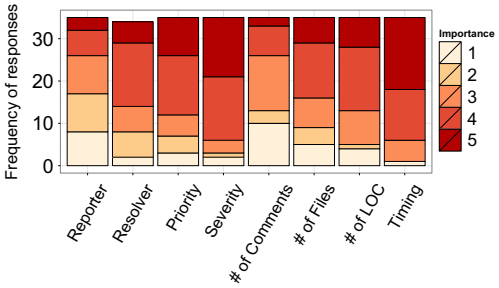


Fig. 17 Frequency of ranks per factor

Theme 4—Frustration Integration delay may generate frustration to both users and developers of the software. The majority of users' frustration comes from their *expectation*(20) about the fixed issues. *F07* makes an interesting analogy to explain user frustration: *"as a user, it's like when you are waiting your suitcase in the airport to come out on the belt. You know it has to be there, but you keep waiting." F14* also provides another analogy: *"it's like a gift for Christmas, but the day of Christmas is postponed."* On the other hand, developers may get frustrated for different reasons than users' reasons. The greatest frustration source for developers is the feeling of *useless/unreleased work*.(9) According to *F09*, when a fixed issue is delayed, a developer *"feels like [their] work is meaningless." F04* complements *F09* by stating that *"it is frustrating to work on something and not see it shipped."*

Theme 5—Collaboration with other teams Integration delay may also occur due to the overhead that is introduced when *collaboration*(10) is needed between teams. For example, when asked to recall a delayed fixed issue, *F23* answers that *"sometimes, issues that require cross-team cooperation may be delayed when the issue is differently prioritized by each team."* The *marketing*(5) team is mentioned recurrently when integration delay occurs due to other teams' collaboration. For instance, according to *F21*, integration delay *"generally happens when marketing wants to make a splash." F08* also corroborates *F21* by stating that *"product management [may] change their mind about the desirability of a feature, or would like to time the release of the feature with certain external events for marketing reasons."*

Observation 7—The time at which an issue is fixed during a release cycle and the issue severity are the factors that receive the highest ratings of importance In *question #13* of our survey, we ask participants to rate the degree to which a factor is related to integration delay. The factors that we list are: the reporter, the resolver, the priority, the

Table 8 Rating of factors related to integration delay. The highest ratings are in bold

Factor	Average rating (mean)
Time at which an issue is fixed during a release cycle (timing)	**4.257**
Severity	**4.086**
Priority	3.629
Number of LOC	3.571
Resolver	3.441
Number of files	3.314
Number of comments	2.657
Reporter	2.629

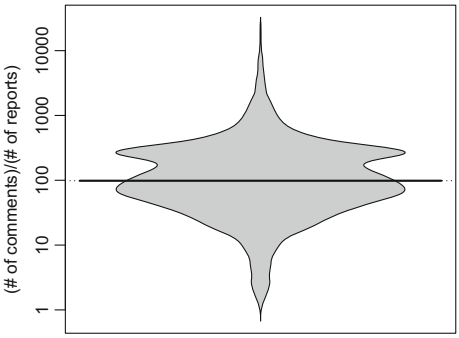


Fig. 18 Distribution of number of comments normalized by the number of reported issues

severity, the number of comments, the number of modified files, the number of modified LOC, and the time at which an issue was fixed during a release cycle. The responses to *question #13* are based on a 5-points-Likert scale, i.e., participants rate factors using ranks from 1 (strongly disagree) to 5 (strongly agree).

In Fig. 17, we show the frequency of each rank per factor, while we show the average rating of each factor in Table 8. We observe that the factors that receive the highest ranks are *severity* and *timing*. This result is in agreement with our regression models that are presented in RQ3, in which *cycle queue rank* is one of the most influential variables (see Observation 6). Indeed, during the interview, *F06* further explains that if an issue that is risky is fixed in the end of a release cycle, such an issue is likely to be delayed to the next cycle, so that it can receive additional testing.

On the other hand, the factors with the lowest ranks are *reporter*, and *# of comments*. We also asked our interviewees about these lower ratings. One of our interviewees explained that the reporter of an issue might influence integration delay only in cases in which the reporter is also a Firefox employee. In these cases, the reporter will fix the issue her/himself, which can speed up the shipping process.[24] As for the *# of comments*, another interviewee clarified that there are several passionate people on bugs that can inflate the number of comments even if the issue is easy to ship. For each reporter, we normalize the number of his/her comments by the number of his/her reported issues. We plot the distribution of the normalized number of comments in Fig. 18. The median number of comments per reported issue is 98. Indeed, we observe reporters with a great number of comments (e.g., 500 to 10,000 comments) per reported issue. This result suggest that the perception of our interviewee is likely to be true.

A Kruskal Wallis test indicates that the difference in ratings between metrics are statistically significant ($p = 0.01507$). Table 9 shows the Bonferroni-Holm corrected $p\text{-}values$ of the Dunn tests. We observe that the *timing* factor has significant larger response values than all the other factors except the *severity*, *priority*, and *LOC* factors ($p < 0.05$).

We also use Spearman's ρ to correlate the rating of the factors with (i) general experience (*question #1*) and (ii) project experience (*question #2*). The only statistically significant correlation that we observe is between the *timing* factor and general experience. We achieve a negative correlation of -0.36 ($p = 0.03235$). This result suggests that less experienced participants tend to report that the time at which an issue is fixed during a release cycle plays a more important role in integration delay. One of our interviewees explains this observation by stating that *"when an issue is fixed early in the release cycle, it should have more time to*

[24] We did not observe a statistically significant difference in integration delays between issues that are fixed by the reporters themselves and issues that are fixed by a different team member.

Table 9 Comparisons between factors. The first row for a factor shows the Bonferroni-Holm statistic, while the second row shows the $p-value$. We use the $*$ and $**$ symbols to denote $p-values$ that are < 0.05 and < 0.0001, respectively

Factor x Factor ($\frac{\text{Holm stat.}}{\text{p-value}}$)	Reporter	Resolver	Priority	Severity
Reporter	—	−2.453397	−3.260129	−4.881387
	—	0.1203	0.0134*	0.000014**
Resolver	−2.453397	—	-0.783021	-2.392487
	0.1203	—	1	0.1171
Priority	−3.260129	−0.783021	—	−1.621257
	0.0134*	1	—	0.4723
Severity	-4.881387	-2.392487	-1.621257	—
	0.000014**	0.1171	0.4723	—
# of Comments	−0.038291	2.415384	3.221838	4.843095
	0.4847	0.1179	0.0140*	0.000016**
# of Files	−2.198691	0.270696	1.061437	2.682695
	0.1813	0.7866	1	0.0657
# of LOC	−2.978304	−0.503246	0.281824	1.903082
	0.0304*	1	1	0.2851
Timing	−5.425890	−2.933031	−2.165761	−0.544503
	0.0000008**	0.0319*	0.1820	1

Factor x Factor ($\frac{\text{Holm stat.}}{\text{p-value}}$)	# of Comments	# of Files	# of LOC	Timing
Reporter	−0.038291	−2.198691	−2.978304	−5.425890
	0.4847	0.1813	0.0304*	0.00000081**
Resolver	2.415384	0.270696	−0.503246	−2.933031
	0.1179	0.7866	1	0.0319*
Priority	3.221838	1.061437	0.281824	−2.165761
	0.0140*	1	1	0.1820
Severity	4.843095	2.682695	1.903082	−0.544503
	0.000016**	0.0657	0.2851	1
# of Comments	—	−2.160400	−2.940013	−5.387599
	—	0.1691	0.0328*	0.00000096**
# of Files	−2.160400	—	−0.779612	−3.227198
	0.1691	—	1	0.0144*
# of LOC	−2.940013	−0.779612	—	−2.447586
	0.0328*	1	—	0.1151
Timing	−5.387599	−3.227198	−2.447586	—
	0.00000096**	0.0144*	0.1151	—

be tested before integration," which can be helpful for fixes from less experienced resolvers. Finally, we also correlate the responses to *question #6* with general and project experience. However, no significant correlations were found.

The integration of fixed issues are delayed due to reasons that are associated with the development activities, decision making, team collaboration, or risk. Moreover, integration delay lead to user/developer frustration.

RQ5: What are developers' perceptions of shifting to a rapid release cycle?

In this RQ, we study the perceptions of developers about the impact of shifting to a rapid release cycle. Our findings about these perceptions are organized along the following themes: *management*, *delivery*, and *development*. We describe each theme below.

Theme 6—Management The shift to a rapid release cycle has a considerable impact on release cycle management.

The most recurrent theme in this respect is *flexibility*(4) to plan the scope of the releases that should be shipped. *F01*'s opinion is that rapid releases *"provide a bit more flexibility, since if an important issue pushed back a less important change and it misses the release cycle, it's not a huge deal with rapid releases."* *F01*'s observation is supported by our observation that rapid Firefox releases tend to deliver fixed issues more consistently (see Observation 2).

Another perceived advantage of rapid release cycles are the *risk mitigation*(3) and *better prioritization*.(3) With respect to *risk mitigation*,(3) *F07* argues that in rapid release cycles, the team is *"able to identify issues sooner. It is easier to identify issues when you have only deployed 3 new commits than 100."* As for *better prioritization*,(3) *F19* explains that rapid release cycles *"probably decreases unnecessary delays of the releases because deadline is closer and developers have to react faster for the pressuring issues. Non-critical issues gets also pushed back and don't receive useless attention nor create delays."* Still on the *better prioritization*(3) matter, *F17* adds that rapid releases *"provide a time box in which [the team] must forecast the top priority work to complete within that time frame."*

Theme 7— Delivery The most recurrent perceived advantage of rapid release cycles is the *"faster delivery"*(33) of new functionalities. When asked about the motivation to use rapid release cycles, *F05* mentions *"increasing speed of getting new features to users,"* while *F06* mentions a similar statement: *"getting new features to users sooner."*. In fact, the *"faster delivery"*(33) motivation is mentioned by participants who have experience with both release strategies and participants who do not. 48% ($\frac{10}{21}$) of the participants who have experience with both release strategies mentioned *"faster delivery"*(33) as a motivation to adopt rapid releases, while 56% ($\frac{9}{16}$) of participants who worked with only one release strategy, mentioned *"faster delivery"*(33). Interestingly, not all participants that mentioned the time to deliver new functionalities report that rapid releases always reduce such time. For *F22*, rapid releases *"reduce the time to deliver issues to end users in some cases, and lengthen them in others."* More specifically, *F24* says that *"Low priority issues (new features) take less time to be delivered, whereas high priority ones (important bugs) take more time."*

Another recurrent perception about rapid releases is the *faster user feedback*(17) due to the constant delivery of new functionalities. For instance, *E29* provides an example that *"you don't find yourself fixing a bug that you introduced two years ago which the field only discovered on the release."*

Theme 8—Development activities We do not observe a specific theme that is recurrent with respect to development activities. Instead, we observe a broad range of themes that are cited by the participants. Among such themes, we observe *quality*,(3) *more functionalities*,(2) *better motivation*,(2) and *better prototyping*.(2) Quality should be a measure of success of using rapid release cycles. According to *E26*, *"quality of delivered code should remain the*

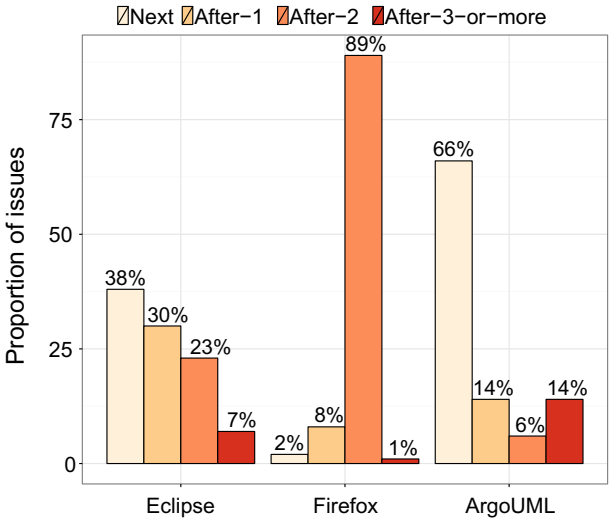


Fig. 19 Proportion of fixed issues that have their integration delayed by a given number of releases. For example, 89% of the fixed issues skip two Firefox stable releases before being shipped to users. This chart was conceived in our prior work (da Costa et al. 2014)

same or improve" after switching to rapid releases. Another way to measure the success of a rapid release cycle is the *number of functionalities*(2) that are completed. *E34* states the following: *"I would see if more issues were completely fixed"* as a measure of success.

Moreover, rapid releases may also impact team members' motivation. For instance, *F06*'s opinion about why to switch to rapid release cycles is *"the need to motivate the community via more frequent collaboration."* Finally, rapid release cycles may also improve prototyping activities. For instance, *E27* argues that, by adopting rapid releases, a development team can *"fix bugs quickly [and] prototype features, having results in few months."*

The allure of delivering fixed issues more quickly is the most recurrent motivator to switch to a rapid release cycle. In addition, the allure of improving management flexibility and the quality of fixed issues are other perceived advantages of switching to rapid release cycles.

RQ6: To what extent do developers agree with our quantitative findings about integration delay?

In this research question, we investigate how our participants feel about the data that we collect during our prior quantitative studies (i.e., the Study I of this paper and our prior work (da Costa et al. 2014)). This research question is divided into two subsections: (i) integration delay in general and (ii) the impact of rapid release cycles on integration delay.

Theme 9—Integration delay in general In this analysis, we present the data that we collect in our prior work (da Costa et al. 2014) and investigate if this data resonates with participants' experience. We provide the methodology of our data-related questions to participants through a web page that is mentioned in our surveys (see Appendix D).

Figure 19 shows the chart that we presented to participants. For example, 89% of the fixed issues skip two Firefox stable releases before being shipped to users. The most recurrent themes among the responses of participants to explain this data are: *team workload*(5) and *dependency*.(2) Among the responses that are related to *team workload*,(5) *E27* explains that *"committers are too busy,"* while *E26* argues that there might be *"delay[s] in review[s] when the issues [are] completed,"* which can generate integration delay. Regarding *dependency*,(2) *E32*'s opinion is that integration delay may happen due to *"the strong connection to other Eclipse projects which makes integration more costly (time consuming)."* Furthermore, two of our interviewees (*F11* and *F23*) provide us with examples of why fixed issues may be delayed due to dependency problems. For example, *F23* explains during the interview that integration delay can happen when there are *"dependencies between projects and one of them gets done, but the other implementation takes a longer while."* Another example, provided by *F23* is when *"you release a bug fix but then you realize: Hey! These users are not able to use these websites anymore because web servers implement the spec in a wrong way or do some really weird things that are not expected."*

Additionally, we ask participants from the Eclipse and ArgoUML projects about their opinion of why the data from the Firefox project behaves differently from theirs, i.e., a larger number of releases being skipped by fixed issues. The most recurrent responses explain that this difference may be due to the *rapid release cycle*(4) that is adopted by the Firefox project. For example, *E30*'s opinion is that *"on a rapid release cycle (e.g. 6 weeks for firefox), a two-release delay means 12 weeks, less than 3 months, which is still less than no delay for a fix submitted early in a project with a 6-month release cycle."*

Theme 10—Impact of switching to a rapid release cycle We present the data that is shown in Fig. 10b to the participants of the Firefox project. Figure 10b compares the integration delay between traditional and rapid release cycles. We then ask if this result resonates with the participants' experience. More details about how we show this data to participants can be found in Appendix E. From the 14 responses that we received for this question, 5 participants explicitly disagree with our analysis, while 6 participants explicitly agree with it.

We interviewed two participant that disagree with the results (*F06* and *F09*). After providing extra explanation about our methodology and asking them to elaborate on their responses, we could better understand their reasons. *F06* clarifies: *"I'm not surprised that there are things in that bucket"* (the short delays due to minor traditional releases), instead *"I'm surprised that there are many of them."* In addition, *F09* declared *"I misunderstood [your] question, but now it [(the data)] makes sense."* With respect to the remaining participants that disagree with our results, they inform us that the data does not resonate with their experience. For instance, *F21* provides the following opinion *"this does not resonate with my experience. I find the traditional model is much much slower than rapid release to get fixes in users hands."* From the set of participants that agree with our results, two of them explain that the behaviour that is presented by the traditional release data is due to the *integration rush*(2) that happens prior to shipping. *F15*'s opinion is that *"since missing a release cycle isn't a big deal, more features are kept from being released until they're properly polished instead of being rushed at the end of a long release cycle."* *F22* also provides us with a reasonable explanation when stating that our result *"makes perfect sense as issues will, unless fast-tracked or held back, be released a set quantum of time after they are completed. This is dominated by the timing of the release schedule, not by the timing of the discovery or fix."*

The dependency of fixed issues on other projects and team workload are major perceived reasons to explain our findings about integration delays. In addition, because (i) an integration rush is no longer needed in rapid releases and (ii) additional time can be spent on polishing fixed issues, rapid releases might have a longer integration time.

6 Analysis of Potential Confounding Factors

In this section, we discuss if the difference of integration delay between release strategies could be due to confounding factors, such as the type and the size of the fixed issues.

Observation 8—The integration delay of fixed issues is not likely to be associated with the size of an issue One may suspect that the difference in integration delay between release strategies may be due to the *size of an issue*. We use the *number of files*, *LOC*, and *number of packages* that were involved in the fix of an issue to measure the *size of an issue*. Figure 20 shows the distributions of the metrics that measure the *size of an issue*. We observe that the difference between distributions of *LOC* is statistically insignificant ($p = 0.86$). As for the *number of files* and the *number of packages*, although we observe significant differences (p values of 0.014 and $< 2.2e^{-16}$, respectively), effect-sizes are *negligible* ($delta = -0.05$ and $delta = -0.07$, respectively).

Observation 9—The difference between traditional and rapid releases is unlikely to be related to the differences between enhancements and bug fixes We also investigate if the observed difference in the integration delay between traditional and rapid releases is related to the kind of fixed issues. For example, rapid releases could be delivering more enhancements, which likely require additional integration time in order to ensure that the new content is of sufficient quality. Figure 21 shows the distributions of delays among release strategies grouped by bug fixes and enhancements. We observe no clear distinction between integration delay and the kind of fixed issues being integrated.

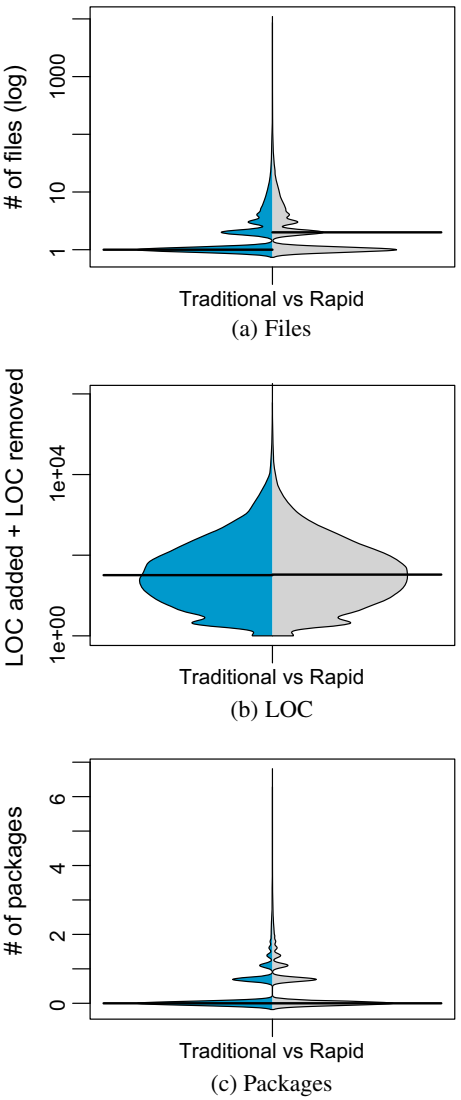
7 Discussion

In this section, we discuss suggestions for practitioners and researchers as well as lessons learned that are based on the results of our empirical studies.

7.1 Practical Suggestions

Small transition The choice of adopting a rapid release cycle is often motivated by the allure of accelerating the delivery of fixed issues. Such a choice needs to be carefully rethought. We observe in our empirical study that although issues are fixed faster, they tend to wait longer to be integrated in the Firefox rapid releases (see Observation 2). One suggestion for software organizations is to begin the transition of release cycles in specific teams or specific products if possible. The result of such a small transition could be compared with the current development process to test the impact of a more rapid release cycle on the delivery of fixed issues.

Consistency of delivering fixed issues Our empirical study suggests that rapid releases can improve the consistency of the time to deliver fixed issues (see Observation 2). A more



(a) Files

(b) LOC

(c) Packages

Fig. 20 Size of the fixed issues in the traditional and rapid release data

consistent delivery of fixed issues can be an advantage for the software organization, since end users would have a better understanding as to when issues will be fixed and integrated.

Minor releases We observe that a large contributor to the faster delivery of fixed issues in the Firefox traditional releases is due to minor releases (see Observation 3). One suggestion is that more effort should be invested in accommodating minor releases to issues that are urgent without compromising the quality of the other releases that are being shipped.

8 Threats to Validity & Limitations

In this section we discuss the threats to the validity of our quantitative study and the limitations of our qualitative study.

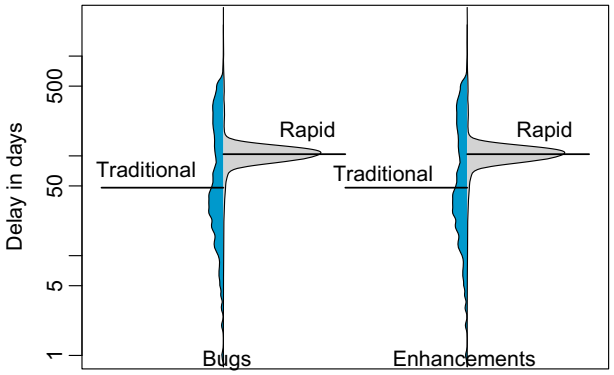


Fig. 21 We group the fixed issues into "bugs" and "enhancements" by using the *severity* field. However, the difference in the integration delay between release strategies is unlikely to be related with the kind of the issue

8.1 Threats to Validity

Construct Validity Construct threats to validity are concerned with the degree to which our analyses are measuring what we are claiming to analyze. Tools were developed to extract and analyze the integration data in the studied projects. Defects in these tools could have an influence on our results. However, we carefully tested our tools using manually-curated subsamples of the studied projects, which produced consistent results.

Moreover, the way that we link issue IDs to releases may not represent the total fixed issues per release. For example, Firefox developers might not record the issue ID in commit logs when fixing issues. Nevertheless, we achieve good linkage rates between commit logs and fixed issues in the studied time period. We link 77% of commit logs for traditional releases, while we link 97% of the commit logs for rapid releases.

Internal Validity Internal threats to validity are concerned with the ability to draw conclusions from the relation between the studied independent and dependent variables.

In Section 6, we compare the integration delay between rapid and traditional releases by grouping the issues as bug fixes or enhancements. We use the *severity* field of the issue reports to perform this grouping. We are aware that the severity field is noisy (Herraiz et al. 2008; Tian et al. 2015) (i.e., many values represent the same level of importance). Still, the *enhancement* severity is one of the significantly different values of severity according to previous research (Herraiz et al. 2008). We also use the number of files, packages, and the LOC to approximate the *size* of an issue. Although these are widely used metrics to measure the size of a change, we are aware that this might not represent the true complexity of the fix of an issue.

External Validity External threats are concerned with our ability to generalize our results. In the quantitative part of our work, we study Firefox releases, since the Firefox project shifted from a traditional release cycle to a rapid release cycle. Although we control for variations using the same studied project in different time periods, we are not able to generalize our conclusions to other projects that adopt a traditional/rapid release cycle. In order to mitigate the external threat, we also perform a qualitative study of the Firefox, ArgoUML, and Eclipse projects. By adding two other projects in our qualitative analysis, and analyzing new sources of data (our participants), we are able to gain insights from other subjects and better understand why integration delay occurs. Still, we cannot claim that our results are

generalizable to other software projects that are not studied in this work. Hence, replication of this work using other projects is required in order to reach more general conclusions.

8.2 Limitations of our Qualitative Study

The main limitation of our qualitative study is its 5% ($\frac{37 \text{ responses}}{780 \text{ e-mails}}$) response rate. We recognize that the general population of our studied projects might have different characteristics and opinions than the ones that we present. Nevertheless, the purpose of our qualitative study is not to achieve generalizability (as in quantitative studies) nor theoretical saturation, but rather to enrich the explanations of our quantitative results, which does not necessarily require a large sample size (Sandelowski 1995). Future research is necessary to better understand why some companies decide not to choose rapid releases. We hope that our study will help decision makers who are considering or dismissing rapid releasing as an efficient strategy to deliver releases.

In addition, while we aimed to achieve a high response rate by personalizing the survey, we were not as successful as we hoped. We still believe that personalization is important. However, it might be worthwhile to present the personalized part of the survey earlier to the participants (e.g., presenting it as the title of the invitation). For example, focusing on particular recent issues might produce a higher response rate instead of asking more general questions.

Moreover, a lengthy survey is not desirable as it is associated with lower response rates (Sheehan 2001). Although we strived to produce a short survey, our response rate is lower than usual (Smith et al. 2013). We believe that this low response rate might be attributed to the very targeted nature of survey–our surveys targets a very specific population (i.e., mostly senior developers who were part of specific issues) in contrast to many prior surveys who have a wider population of varying experience (Smith et al. 2013).

Finally, although the coding process is performed by two authors independently and reviewed by a third author, we cannot claim that we reach all the perspectives that are possible from our questions.

9 Related Work

In this section, we situate our study with respect to prior work on the impact of adopting rapid release cycles and the process of integrating and delivering fixed issues.

Traditional vs. Rapid Releases Shifting from traditional releases to rapid releases has been shown to have an impact on software quality and quality assurance activities. Mäntylä et al. (2014) found that rapid releases have more tests executed per day but with less coverage. The authors also found that the number of testers decreased in rapid releases, which increased the test workload. Souza et al. (2014) found that the number of reopened bugs increased by 7% when Firefox changed to a rapid release cycle. Souza et al. (2015) found that backout of commits increased when rapid releases were adopted. However, they note that such results may be due to changes in the development process rather than the rapid release cycle—the backout culture was not widely adopted during the traditional Firefox releases. We also investigate the shift from traditional releases to rapid releases in this paper. However, we analyze integration delay rather than quality and quality assurance activities.

It is not clear yet if rapid releases lead to a faster rate of bugs fixes. Baysal et al. (2011) found that bugs are fixed faster in Firefox traditional releases when compared to fixes in the Chrome rapid releases. On the other hand, Khomh et al. (2012) found that bugs that are associated with crash reports are fixed faster in rapid Firefox releases when compared to

Firefox traditional releases. However, fewer bugs are fixed in rapid releases, proportionally. Our study corroborates that issues are fixed more quickly in rapid release cycles, but tend to wait longer to be delivered to the end users.

Rapid releases may cause users to adopt new versions of the software earlier. Baysal et al. (2011) found that users of the Chrome browser are more likely to adopt new versions of the system when compared to traditional Firefox releases. Khomh et al. (2012) also found that the new versions of Firefox that were developed using rapid releases were adopted more quickly than the versions under traditional releases. In this paper, we investigate the impact that a shift from traditional to rapid releases has on delivering fixed issues to users rather than user adoption of new releases.

Delays and Software Issues Prior research has studied delays that are related to the integration and delivery of fixed issues to end users. Jiang et al. (2013) studied the integration process of the Linux kernel. They found that 33% of the code patches that were submitted to resolve issues are accepted into an official Linux release after 3 to 6 months. In our prior work (da Costa et al. 2014), we investigate how many releases a fixed issue may be delayed before shipment. We found that 98% of fixed issues in the rapid releases of the Firefox project were delayed by at least one release. Unlike prior work (da Costa et al. 2014), this paper investigates how the change of release strategy relates to integration delay.

Morakot et al. (2015a, b) study the risk of delaying the resolution of issues. The authors found that metrics such as the percentage of delayed issues that a developer is involved with, discussion time, and number of issue reopenings are strongly related to the risk of postponing the resolution of issues. Rahman and Rigby (2015) found that the period to stabilize fixed issues can take from 45 to 93 days in the Linux kernel and from 56 to 149 days in Chrome. Jiang and Adams (2014) propose the ISOMO model to measure the cost of integrating a new patch into a host project. Our work complements the aforementioned studies by quantitatively and qualitatively investigating the impact that the adoption of a rapid release cycle may have upon the integration delay of fixed issues.

Finally, the problem of prioritizing which fixed issues should be integrated in the next release is well known in the requirements engineering field (Paetsch et al. 2003; Regnell and Brinkkemper 2005; Karlsson and Ryan 1997; Greer and Ruhe 2004). Fixed issues should be prioritized based on whether they meet the needs of a specific customer or of an open market with many customers (Regnell and Brinkkemper 2005). To prioritize fixed issues according to their relative value and cost, Karlsson and Ryan (1997) proposed a *Cost-Value* approach. The proposed approach is composed of five steps that involve both customers and software engineers to estimate the importance of issues and the effort to fix them. Our work sheds more light on the prioritization problem by quantitatively and qualitatively analyzing the reasons why fixed issues are delayed to future releases.

10 Conclusions

In this paper, we perform two studies of integration delays and the impact that rapid release cycles have on such delays. In our quantitative study, we analyze a total of 72,114 issue reports of 111 traditional releases and 73 rapid releases of the Firefox project. In our qualitative study, we survey 37 participants from the Firefox, ArgoUML, and Eclipse projects. We make the following observations:

– Although issues tend to be fixed more quickly in the rapid release cycle, fixed issues tend to be integrated into consumer-visible releases more quickly in the traditional

release cycle. However, a rapid release cycle may improve the consistency of the delivery rate of fixed issues (see Observation 2).
- We observe that the faster delivery of fixed issues in the traditional releases is partly due to minor-traditional releases. One suggestion for practitioners is that more effort should be invested in accommodating minor releases to issues that are urgent without compromising the quality of the other releases that are being shipped (see Observation 3).
- The triaging time of issues is not significantly different among the traditional and rapid releases (see Observation 2).
- The total time spent from the issue report date to its integration into a release is not significantly different between traditional and rapid releases (see Observation 1).
- In traditional releases, fixed issues are less likely to be delayed if they are fixed early in the backlog. On the other hand, in rapid releases, fixed issues are less likely to be delayed if they are fixed early in the current release cycle (see Observation 6).
- The perceived reasons for integration delay of fixed issues are primarily related to activities such as development, decision making, team collaboration, and risk management (see Themes 1, 2, 3, and 5).
- The dependency of issues on other projects and team workload are the main perceived reasons to explain our data about integration delay in general (see Theme 1).
- The allure of delivering fixed issues more quickly to users is the most recurrent motivator for switching to a rapid release cycle (see Theme 7). In addition, the allure of improving management flexibility and quality of fixed issues are other advantages that are perceived by our participants (see Themes 6 and 8).
- Integration rush and the increased time that is spent on polishing fixed issues (during rapid releases) emerge as one of the main explanations as to why traditional releases may achieve shorter integration delay values (see Theme 10).

Although the bulk of our analyses comes mainly from the Firefox project given the very unique nature of this project (and the availability of the data), we believe that the impact of our findings and work goes well beyond the Firefox project. Today the Firefox project is often used in support of proposals for moving to a rapid release cycle throughout many development organizations worldwide. Yet few studies extensively explored the benefits and challenges of rapid release cycles. In this regard, our quantitative and qualitative observations may serve any organization that is interested in adopting a rapid release cycle. For instance, even though the allure of delivering fixed issues more quickly is the most recurrent motivator to adopt rapid releases (Theme 7), we observe that this often is not achieved (Observation 2). In summary, our study provides real observations and offers a wider context of the dis/advantages of adopting a rapid release strategy.

Moreover, our qualitative study opens new directions for future (quantitative) studies. For example, one could investigate whether a large proportion of the integration delay is happening due to code review delays (see Theme 9). Other future research can investigate the trade-off between software quality and integration delays, since one possible reason for the slower integration of fixed issues in rapid releases is the increased time that is spent on validating and verifying such fixes (see Theme 10).

Acknowledgments This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the National Institute of Science and Technology for Software Engineering (INES), CNPq grant 465614/2014-0. We also thank all the participants from the Firefox, Eclipse, and ArgoUML projects for giving their time to respond our surveys and participate in our interviews.

Appendix A: Firefox Survey

Understanding the Delivery Delay of Completed Issues

This survey is part of a broad research project about the delay to deliver new software issues to end users. By issues we broadly refer to bugs, new features and enhancements. Our research team is from Australia, Brazil, and Canada.

Our goal is to study how much time is necessary to deliver issues that are completed (i.e., implemented and tested) to the end users. For example, whether a completed issue is delivered as soon as it is completed or if it is often delayed for several releases.

The survey will ask you a few general and specific questions (i.e., we will show you data that is derived from Firefox).

1. By answering this survey, you are eligible to win a $100 gift card from Amazon. If you'd like to participate in the draw, please leave your email address below.

2. For how long have you been developing software?
   Mark only one oval.
   - ◯ 0 years
   - ◯ 1 year
   - ◯ 2 years
   - ◯ 3 years
   - ◯ 4 years
   - ◯ 5 years
   - ◯ 6 years
   - ◯ 7 years
   - ◯ 8 years
   - ◯ 9 years
   - ◯ 10 or more years

3. For how long have you worked in the Firefox project?
   Mark only one oval.
   - ◯ 0 years
   - ◯ 1 year
   - ◯ 2 years
   - ◯ 3 years
   - ◯ 4 years
   - ◯ 5 or more years

4. How would you describe your roles in the software development of the Firefox project? (e.g., developer, tester, release manager etc.)

5. In your opinion, what motivates a development team to shift from a traditional release cycle (e.g., a release every 9 to 18 months) to a rapid release cycle (e.g., a release every 6 weeks)?

6. In this survey, we consider that an issue is completed when it is implemented and tested, i.e., it is ready to be integrated. Do you remember an issue that the development team completed work on, but was not delivered to end users through the next possible release? Can you tell us what caused the delivery delay of this issue in your opinion?

7. In your experience, how common are the cases in which completed issues (issues that are implemented and tested) are omitted from the next possible release?
   Mark only one oval.

   - All completed issues are included in the next possible release.
   - More than 90% of all completed issues are included in the next possible release.
   - More than three quarters of all completed issues are included in the next possible release.
   - More than a half of all completed issues are included in the next possible release.
   - About a half of all completed issues are included in the next possible release.
   - Fewer than a half of all completed issues are included in the next possible release.
   - Fewer than a quarter of all completed issues are included in the next possible release.
   - Fewer than 10% of all completed issues are included in the next possible release.
   - No completed issues are included in the next possible release.

8. Who decides when a completed issue is integrated into an official release in your team?

9. In your opinion, when is the delivery of a completed issue to the end user considered to be delayed in your project?

Delivery delay in your project

In our research, we consider that a completed issue suffers from a delivery delay if such an issue is not included in the next release immediately after the issue is completed, i.e., such a completed issue is postponed to be delivered in future releases. We use the term delivery delay to refer to completed issues that are not delivered in the next immediate release.

10. In your opinion, is it frustrating to users when a completed issue skips one or more releases? Why?

11. Is it frustrating for the team members when a completed issue skips one or more releases? Why?

Reasons related to delivery delay

We are developing a tool to detect if a completed issue will suffer from delivery delay. In this part of the survey, we want to know your opinion about reasons that we are investigating in our research to identify completed issues that are likely to suffer from delivery delay.

12. Assuming that an issue is completed today (implementation and testing are completed), what reasons can you think of for the issue not to be delivered to end users in the next release?

Reasons related to delivery delay

13. What can team members do to avoid the delivery delay of completed issues?

14. To what extent do you agree that the characteristics listed in the table below are related to the delivery delay of a completed issue?
Mark only one oval per row.

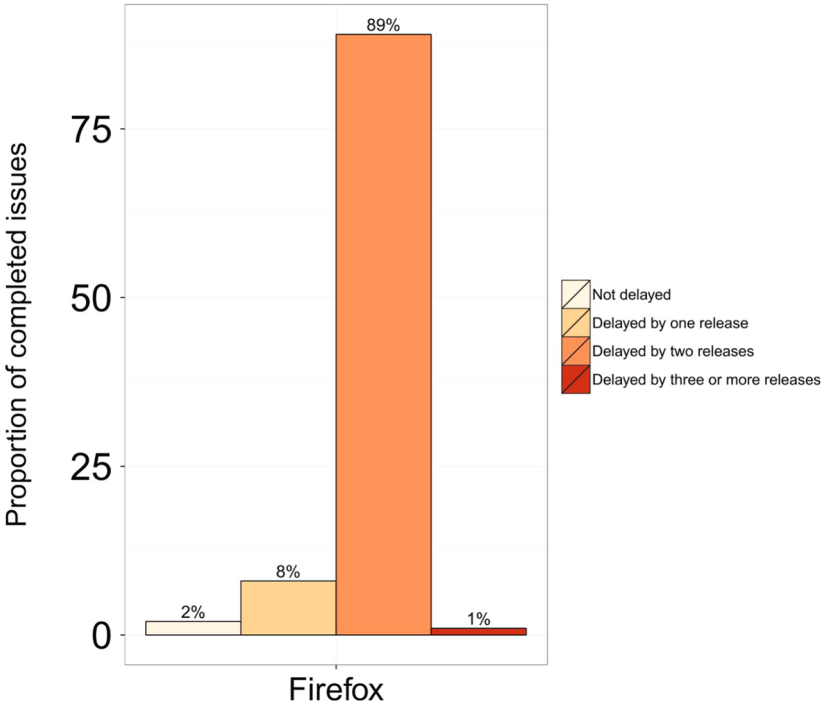
	Strongly agree	Agree	Neither agree nor disagree	Disagree	Strongly disagree
The reporter of a completed issue					
The resolver of a completed issue					
The priority value of a completed issue					
The severity value of a completed issue					
Number of comments recorded in a completed issue					
Number of files modified to complete an issue					
Number of lines of code to complete an issue					
The time at which an issue is completed during a release cycle					

Feedback on the results about delivery delay

In this section, we are interested in your feedback about the results that we found in our research using the publicly available data of your project.

15. We found that 89% of the Firefox completed issues are delayed by two releases (see Figure 1) from releases 10 to 27. In your opinion, why is this the case for the Firefox project? More details about the methodology of this finding in http://goo.gl/VC3CoK

Figure 1. Proportion of completed issues that suffered from delivery delay.



16. We find that: (1) the more time spent on issues in the release cycle, the shorter the delivery delay and (2) the higher the number of completed-but-not-yet-integrated issues the larger the delivery delay. Do these results resonate with your experience? Why do you think so?

Shifting to a rapid release cycle

In this part of the survey, we are interested in getting your feedback about the impact of shifting to a rapid release cycle on the delivery delay of completed issues. We also present results that are obtained in our research.

17. Have you worked in both traditional and rapid release cycles of the Firefox project?
    Mark only one oval.

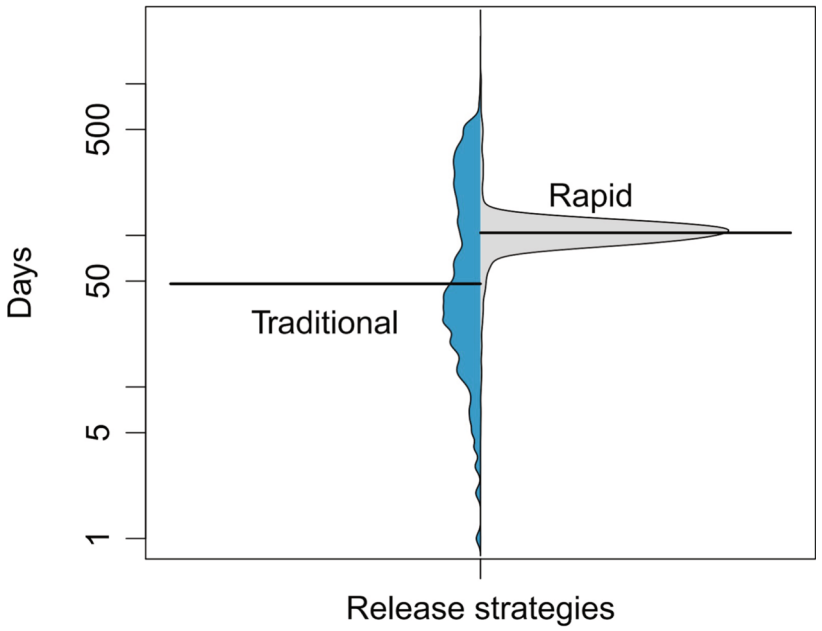
    ◯ Yes
    ◯ No

18. In your opinion, how much impact does a rapid release cycle have on the time to deliver completed issues for end users?

19. Did your project evaluate the shift to rapid release cycles? If so, how?

20. In our research, we compared the time in days that traditional and rapid releases (both minor and majors) take to deliver completed issues to users. We obtained the results that are provided in Figure 3. The Figure shows a beanplot for each release strategy. The vertical curves of beanplots compare the distributions in traditional and rapid releases. The higher the frequency of data within a particular value, the thicker the bean is plotted at that particular value on the y axis. Finally, the black horizontal line represents the median value of each distribution. We observe that the median number of days to deliver is significantly higher with the rapid release cycle, but there is much less variation. Does this result resonate with your experience? Why do you think so? More details about the methodology of this finding in http://goo.gl/me9aOw

Figure 3. Number of days (log-scale) to deliver completed issues in traditional and rapid release cycles.



Ending our questionnaire

21. If you'd like to participate for the $100 Amazon gift card draw and haven't provided your e-mail yet, please leave it below.

22. Would you like to be informed about our findings?
Mark only one oval.

- Yes
- No

23. Would you be willing to be contacted for a quick online follow-up interview (at a time convenient for you)?
Mark only one oval.

- Yes
- No

24. Do you have further comments for us?

Appendix B: ArgoUML Survey

Understanding The Delivery Delay of Completed Issues

This survey is part of a broad research project about the delay to deliver new software issues to end users. By issues we broadly refer to bugs, new features and enhancements. Our research team is from Australia, Brazil, and Canada.

Our goal is to study how much time is necessary to deliver issues that are completed (i.e., implemented and tested) to the end users. For example, whether a completed issue is delivered as soon as it is completed or if it is often delayed for several releases.

The survey will ask you a few general and specific questions (i.e., we will show you data that is derived from ArgoUML).

1. By answering this survey, you are eligible to win a $100 gift card from Amazon. If you'd like to participate in the draw, please leave your email address below.

2. For how long have you been developing software?
   Mark only one oval.
   - ◯ 0 years
   - ◯ 1 year
   - ◯ 2 years
   - ◯ 3 years
   - ◯ 4 years
   - ◯ 5 years
   - ◯ 6 years
   - ◯ 7 years
   - ◯ 8 years
   - ◯ 9 years
   - ◯ 10 or more years

3. For how long have you worked in the ArgoUML project?
   Mark only one oval.
   - ◯ 0 years
   - ◯ 1 year
   - ◯ 2 years
   - ◯ 3 years
   - ◯ 4 years
   - ◯ 5 or more years

4. How would you describe your roles in the software development of the ArgoUML project? (e.g., developer, tester, release manager etc.)

5. In your opinion, what motivates a development team to shift from a traditional release cycle (e.g., a release every 9 to 18 months) to a rapid release cycle (e.g., a release every 6 weeks)?

6. In this survey, we consider that an issue is completed when it is implemented and tested, i.e., it is ready to be integrated. Do you remember an issue that the development team completed work on, but was not delivered to end users through the next possible release? Can you tell us what caused the delivery delay of this issue in your opinion?

7. In your experience, how common are the cases in which completed issues (issues that are implemented and tested) are omitted from the next possible release?
   Mark only one oval.

   - ◯ All completed issues are included in the next possible release.
   - ◯ More than 90% of all completed issues are included in the next possible release.
   - ◯ More than three quarters of all completed issues are included in the next possible release.
   - ◯ More than a half of all completed issues are included in the next possible release.
   - ◯ About a half of all completed issues are included in the next possible release.
   - ◯ Fewer than a half of all completed issues are included in the next possible release.
   - ◯ Fewer than a quarter of all completed issues are included in the next possible release.
   - ◯ Fewer than 10% of all completed issues are included in the next possible release.
   - ◯ No completed issues are included in the next possible release.

8. Who decides when a completed issue is integrated into an official release in your team?

9. In your opinion, when is the delivery of a completed issue to the end user considered to be delayed in your project?

Delivery delay in your project

In our research, we consider that a completed issue suffers from a delivery delay if such an issue is not included in the next release immediately after the issue is completed, i.e., such a completed issue is postponed to be delivered in future releases. We use the term delivery delay to refer to completed issues that are not delivered in the next immediate release.

10. In your opinion, is it frustrating to users when a completed issue skips one or more releases? Why?

11. Is it frustrating for the team members when a completed issue skips one or more releases? Why?

Reasons related to delivery delay

We are developing a tool to detect if a completed issue will suffer from delivery delay. In this part of the survey, we want to know your opinion about reasons that we are investigating in our research to identify completed issues that are likely to suffer from delivery delay.

12. Assuming that an issue is completed today (implementation and testing are completed), what reasons can you think of for the issue not to be delivered to end users in the next release?

Reasons related to delivery delay

13. What can team members do to avoid the delivery delay of completed issues?

14. To what extent do you agree that the characteristics listed in the table below are related to the delivery delay of a completed issue?
Mark only one oval per row.

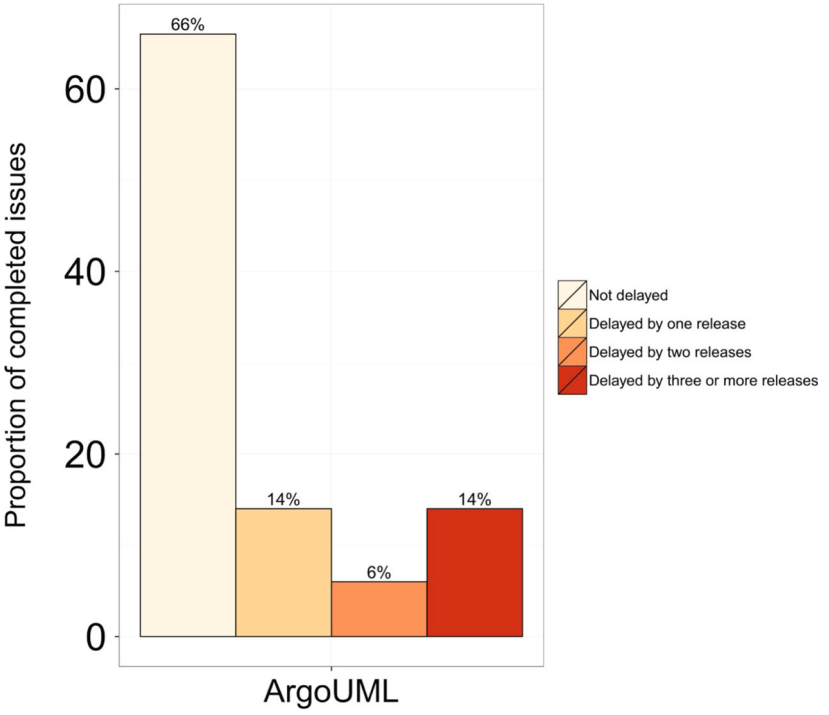
	Strongly agree	Agree	Neither agree nor disagree	Disagree	Strongly disagree
The reporter of a completed issue					
The resolver of a completed issue					
The priority value of a completed issue					
The severity value of a completed issue					
Number of comments recorded in a completed issue					
Number of files modified to complete an issue					
Number of lines of code to complete an issue					
The time at which an issue is completed during a release cycle					

Feedback on the results about delivery delay

In this section, we are interested in your feedback about the results that we found in our research using the publicly available data of your project.

15. We found that 34% of the ArgoUML completed issues are delayed by at least one release (See Figure 1). In your opinion, why is this the case for the ArgoUML project? More details about the methodology of this finding in http://goo.gl/VC3CoK

Figure 1. Proportion of completed issues that suffered from delivery delay.



16. We find that: (1) the more time spent on issues in the release cycle, the shorter the delivery delay and (2) the higher the number of completed-but-not-yet-integrated issues the larger the delivery delay. Do these results resonate with your experience? Why do you think so?

Shifting to a rapid release cycle

In this part of the survey, we are interested in getting your feedback about the impact of shifting to a rapid release cycle on the delivery delay of completed issues.

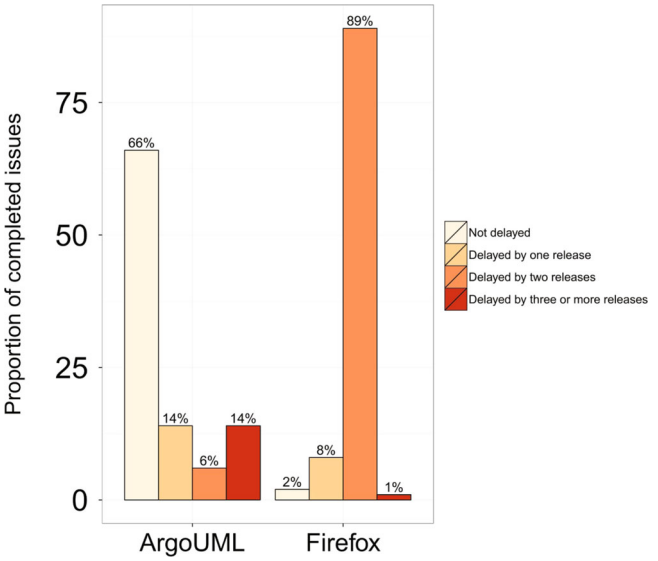
17. Do you have experience working on a rapid release cycle in any other project?
    Mark only one oval.

    Yes
    No

18. In your opinion, what would be the impact of shifting to a rapid release cycle (e.g., a release every 6 weeks rather than a release every 9 to 18 months) on the delay to deliver completed issues, in your project?

19. Figure 2 shows the Firefox project in which 90% of completed issues are delayed by at least two releases. Firefox adopts a rapid release cycle. How do you feel about the difference between your project and the Firefox project?

Figure 2. Proportion of completed issues that suffered delivery delay in the ArgoUML and Firefox projects.



20. If your project had shifted from a traditional to a rapid release cycle, how would you evaluate if this shift benefited your project?

Ending our questionnaire

21. If you'd like to participate in the $100 Amazon gift card draw and haven't provided your e-mail yet, please leave it below.

22. Would you like to be informed about our findings?
Mark only one oval.

- Yes
- No

23. Would you be willing to be contacted for a quick online follow-up interview (at a time convenient for you)?
    Mark only one oval.

    - ◯ Yes
    - ◯ No

24. Do you have further comments for us?

Appendix C: Eclipse Survey

Understanding The Delivery Delay of Completed Issues

This survey is part of a broad research project about the delay to deliver new software issues to end users. By issues we broadly refer to bugs, new features and enhancements. Our research team is from Australia, Brazil, and Canada.

Our goal is to study how much time is necessary to deliver issues that are completed (i.e., implemented and tested) to the end users. For example, whether a completed issue is delivered as soon as it is completed or if it is often delayed for several releases.

The survey will ask you a few general and specific questions (i.e., we will show you data that is derived from Eclipse JDT).

1. By answering this survey, you are eligible to win a $100 gift card from Amazon. If you'd like to participate in the draw, please leave your email address below.

2. For how long have you been developing software?
   Mark only one oval.
   - ◯ 0 years
   - ◯ 1 year
   - ◯ 2 years
   - ◯ 3 years
   - ◯ 4 years
   - ◯ 5 years
   - ◯ 6 years
   - ◯ 7 years
   - ◯ 8 years
   - ◯ 9 years
   - ◯ 10 or more years

3. For how long have you worked in the Eclipse (JDT) project?
   Mark only one oval.
   - ◯ 0 years
   - ◯ 1 year
   - ◯ 2 years
   - ◯ 3 years
   - ◯ 4 years
   - ◯ 5 or more years

4. How would you describe your roles in the software development of the Eclipse JDT project? (e.g., developer, tester, release manager etc.)

5. In your opinion, what motivates a development team to shift from a traditional release cycle (e.g., a release every 9 to 18 months) to a rapid release cycle (e.g., a release every 6 weeks)?

6. In this survey, we consider that an issue is completed when it is implemented and tested, i.e., it is ready to be integrated. Do you remember an issue that the development team completed work on, but was not delivered to end users through the next possible release? Can you tell us what caused the delivery delay of this issue in your opinion?

7. In your experience, how common are the cases in which completed issues (issues that are implemented and tested) are omitted from the next possible release?
   Mark only one oval.

   - ◯ All completed issues are included in the next possible release.
   - ◯ More than 90% of all completed issues are included in the next possible release.
   - ◯ More than three quarters of all completed issues are included in the next possible release.
   - ◯ More than a half of all completed issues are included in the next possible release.
   - ◯ About a half of all completed issues are included in the next possible release.
   - ◯ Fewer than a half of all completed issues are included in the next possible release.
   - ◯ Fewer than a quarter of all completed issues are included in the next possible release.
   - ◯ Fewer than 10% of all completed issues are included in the next possible release.
   - ◯ No completed issues are included in the next possible release.

8. Who decides when a completed issue is integrated into an official release in your team?

9. In your opinion, when is the delivery of a completed issue to the end user considered to be delayed in your project?

Delivery delay in your project

In our research, we consider that a completed issue suffers from a delivery delay if such an issue is not included in the next release immediately after the issue is completed, i.e., such a completed issue is postponed to be delivered in future releases. We use the term delivery delay to refer to completed issues that are not delivered in the next immediate release.

10. In your opinion, is it frustrating to users when a completed issue skips one or more releases? Why?

11. Is it frustrating for the team members when a completed issue skips one or more releases? Why?

Reasons related to delivery delay

We are developing a tool to detect if a completed issue will suffer from delivery delay. In this part of the survey, we want to know your opinion about reasons that we are investigating in our research to identify completed issues that are likely to suffer from delivery delay.

12. Assuming that an issue is completed today (implementation and testing are completed), what reasons can you think of for the issue not to be delivered to end users in the next release?

Reasons related to delivery delay

13. What can team members do to avoid the delivery delay of completed issues?

14. To what extent do you agree that the characteristics listed in the table below are related to the delivery delay of a completed issue?
Mark only one oval per row.

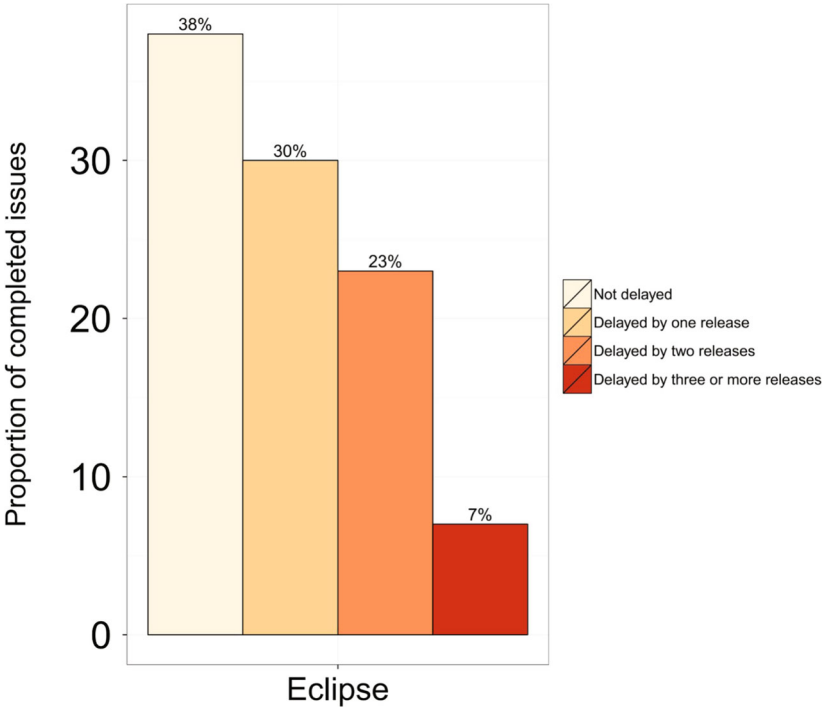
	Strongly agree	Agree	Neither agree nor disagree	Disagree	Strongly disagree
The reporter of a completed issue					
The resolver of a completed issue					
The priority value of a completed issue					
The severity value of a completed issue					
Number of comments recorded in a completed issue					
Number of files modified to complete an issue					
Number of lines of code to complete an issue					
The time at which an issue is completed during a release cycle					

Feedback on the results about delivery delay

In this section, we are interested in your feedback about the results that we found in our research using the publicly available data of your project.

15. We found that 60% of the Eclipse JDT completed issues are delayed by at least one release (See Figure 1). In your opinion, why is this the case for the Eclipse JDT project? More details about the methodology of this finding in http://goo.gl/VC3CoK

Figure 1. Proportion of completed issues that suffered from delivery delay.



16. We find that: (1) the more time spent on issues in the release cycle, the shorter the delivery delay and (2) the higher the number of completed-but-not-yet-integrated issues the larger the delivery delay. Do these results resonate with your experience? Why do you think so?

Shifting to a rapid release cycle

In this part of the survey, we are interested in getting your feedback about the impact of shifting to a rapid release cycle on the delivery delay of completed issues.

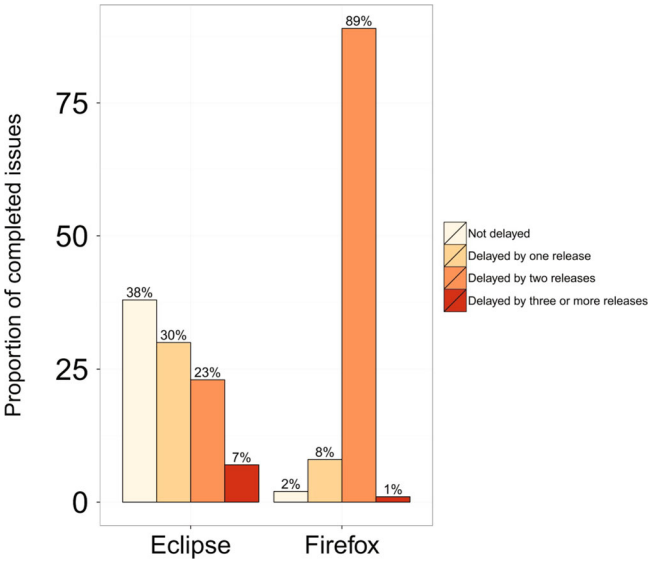
17. Do you have experience working on a rapid release cycle in any other project?
    Mark only one oval.

    Yes
    No

18. In your opinion, what would be the impact of shifting to a rapid release cycle (e.g., a release every 6 weeks rather than a release every 9 to 18 months) on the delay to deliver completed issues, in your project?

19. Figure 2 shows the Firefox project in which 90% of completed issues are delayed by at least two releases. Firefox adopts a rapid release cycle. How do you feel about the difference between your project and the Firefox project? More details about the methodology of this finding in http://goo.gl/VC3CoK

Figure 2. Proportion of completed issues that suffered delivery delay in the Eclipse and Firefox projects.



20. If your project had shifted from a traditional to a rapid release cycle, how would you evaluate if this shift benefited your project?

Ending our questionnaire

21. If you'd like to participate in the $100 Amazon gift card draw and haven't provided your e-mail yet, please leave it below.

22. Would you like to be informed about our findings?
Mark only one oval.

- Yes
- No

23. Would you be willing to be contacted for a quick online follow-up interview (at a time convenient for you)?
Mark only one oval.

- Yes
- No

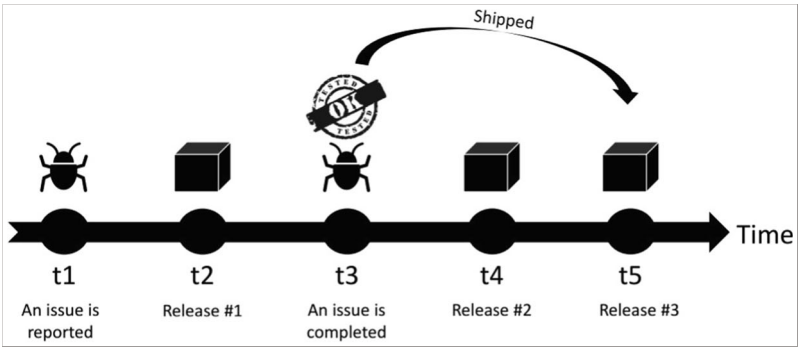
24. Do you have further comments for us?

Appendix D: Methodology Web Page I

How do we compute the delivery delay of completed issues?

In this page, we explain how we measure the data that is shown in page 5 of our survey. You can find the concepts that are necessary to understand the data collection process below.

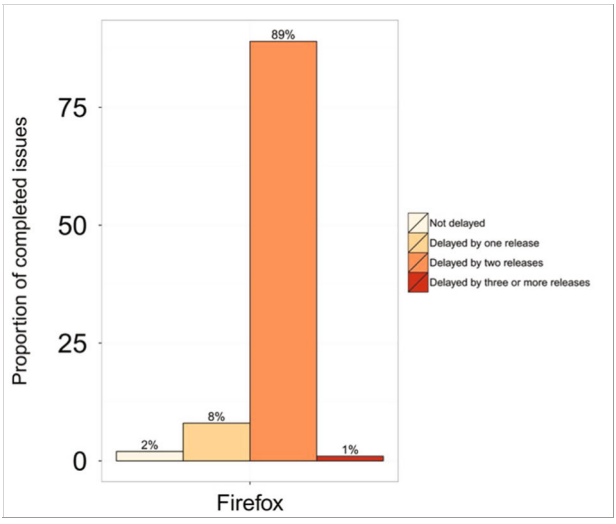
Delivery delay measures how long it takes for a system functionality (i.e., an issue) to be delivered to the end user from the time at which the issue was completed (i.e., implemented and tested).



Delivery delay in terms of releases is the number of official releases that are missed before the issue is officially shipped after it is completed. The figure above illustrates an issue that is completed at time t3 . Such an issue misses release number 2 at time t4 . Finally, the completed issue is shipped in release number 3 at time t5 . In this example, the delivery delay of the completed issue is 1 official release .

By "official release" we mean a release that is intended to be used by the entire user base of the project. For example, in a pipelining release strategy (e.g., as in the Firefox project), in which a release is stabilized through several channels, an official release is the final product of the process, i.e., the release that is to be published to every user from the release channel.

In the figure below, we show the delivery delay in terms of releases for the completed issues in the Firefox project.

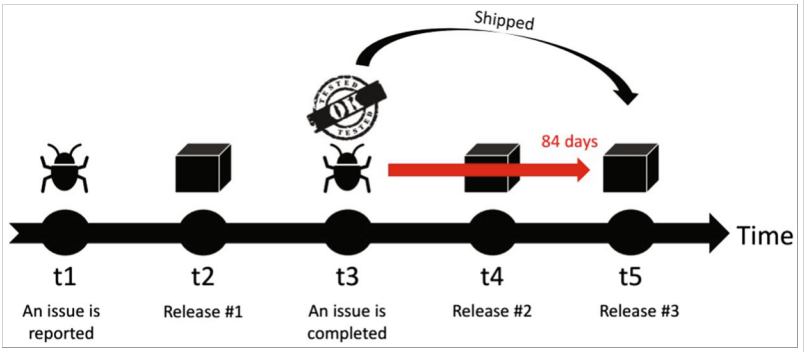


The figure shows that 89% of the Firefox completed issues miss 2 official releases before being shipped to end users.

Appendix E: Methodology Web Page II

How do we compare rapid vs traditional releases?

In this page, we explain how we measure the data that is shown in page 6 of our survey. You can find the concepts that are necessary to understand the data collection process below.

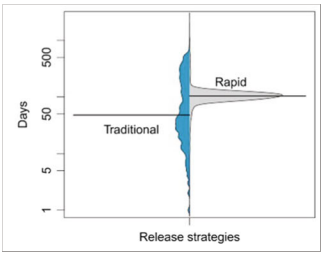


Delivery delay measures how long it takes for a system functionality (i.e., an issue) to be delivered to the end user from the time at which the issue was completed (i.e., implemented and tested).

Delivery delay in terms of days is the number of days for an issue to be officially shipped after it is completed. The figure above illustrates an issue that is completed at time t3 .
This issue takes 84 days to be shipped at t5 .

By "official release" we mean a release that is intended to be used by the entire user base of the project. For example, in a pipelining release strategy (e.g., as in the Firefox project), in which a release is stabilized through several channels, an official release is the final product of the process, i.e., the release that is to be published to every user from the release channel.

In the figure below, we show the delivery delay in terms of days for the completed issues in the Firefox project. We collected data from traditional releases (major and minor releases from version 1.0 to 4.0) and from rapid releases (releases in the release channel from version 10 to 27). The Figure shows a beanplot for each release strategy. The vertical curves of beanplots compare the distributions in traditional and rapid releases. The higher the frequency of data within a particular value, the thicker the bean is plotted at that particular value on the y axis. Finally, the black horizontal line represents the median value of each distribution. We observe that the median number of days to deliver is significantly higher with the rapid release cycle, but there is much less variation.



Appendix F: Invitation Letter

Dear <Participant>,

We are a group of researchers based out of universities in Brazil, Australia, and Canada. We are performing an empirical study to understand why some software functionalities are unexpectedly delayed before reaching end users. With this study, we intend to help software development teams to better deliver software with little delay or schedule slippage.

In our opinion, you are a perfect fit to our research, since we observed that you have participated in the development of the <X> project. Would you mind sharing your thoughts with us by filling out this survey about the <X> project? The survey has 19 questions (all of them are optional) and will take less than 15 minutes to complete.

<Survey URL>

To compensate you for your time, we will have a draw to give away $100 Amazon gift certificates to 5% of all participants that answer all questions.

Please, do not hesitate to contact me if you have any questions. More information about this study is available at <Link of the paper>

Thank you,
Daniel Alencar da Costa.
PhD student at the Federal University of Rio Grande do Norte, Brazil.
http://danielcalencar.github.io

Appendix G: Interview Script

Thank you for participating in our survey about the delay to deliver completed issues!

(Part 1)

Can you tell us about a specific completed issue in one of your projects that was delayed to be delivered?

Prompts:
Lack of code review?
Workload of integrators?
Changing requirements?
Marketing issues?
Possible side effects?
The need of extra testing? (or any other test strategy)
Integration effort?

Was this a typical situation?

What other reasons can you think of that can lead the delivery of a completed issue to be delayed?

In our study, the reporter of an issue and the discussion that occurred to complete an issue received the lowest ranks as possible reasons that lead to delivery delay. Do you agree? Why?

On the other hand, the time at which an issue is completed in the release cycle and the severity of an issue received the highest ranks as reasons that lead to delivery delay. Do you agree? Why?

(Part 2)

Can you tell us about your experience with the shift from a traditional release cycle to a rapid release cycle?

Prompts:
Regarding with tests
Regarding with feedback
Amount of functionalities that are delivered
Overall quality

Can you tell us your impressions about the time to deliver completed issues after the shift?

How would you evaluate the success of a shift to a rapid release cycle?

References

Adams B, McIntosh S (2016) Modern release engineering in a nutshell: why researchers should care. In: Proceedings of the 23rd international conference on software analysis, evolution, and reengineering (SANER), pp 78–90

AlGhamdi HM, Syer MD, Shang W, Hassan AE (2016) An automated approach for recommending when to stop performance tests. In: Proceedings of the international conference on software maintenance and evolution. IEEE, Piscataway, pp 279–289

Antoniol G, Ayari K, Penta MD, Khomh F, Guéhéneuc Y (2008) Is it a bug or an enhancement?: a text-based approach to classify change requests. In: Proceedings of the 2008 conference of the centre for advanced studies on collaborative research (CASCON), pp 23–37

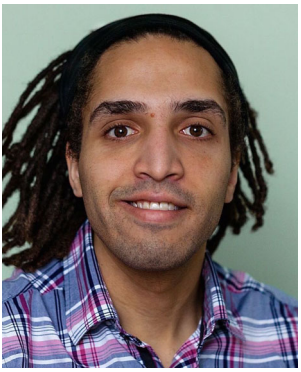
Baskerville R, Pries-Heje J (2004) Short cycle time systems development. Inf Syst J 14:237–264

Baysal O, Davis I, Godfrey MW (2011) A tale of two browsers. In: Proceedings of the 8th working conference on mining software repositories (MSR). ACM, New York, pp 238–241

Beck K (2000) Extreme programming explained: embrace change. Addison-Wesley Professional, Reading
Beller M, Gousios G, Zaidman A (2015) How (much) do developers test? In: 2015 IEEE/ACM 37th IEEE international conference on software engineering, vol 2. IEEE, Piscataway, pp 559–562
Boehm BW (1988) A spiral model of software development and enhancement. Computer 21(5):61–72
Charmaz K (2014) Constructing grounded theory. SAGE, Newbury Park
Cliff N (1993) Dominance statistics: ordinal analyses to answer ordinal questions. Psychol Bull 114:494–509
da Costa DA, Abebe SL, McIntosh S, Kulesza U, Hassan AE (2014) An empirical study of delays in the integration of addressed issues. In: Proceedings of the 30th international conference on software maintenance and evolution (ICSME), pp 281–290
da Costa DA, McIntosh S, Kulesza U, Hassan AE (2016) 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: Proceedings of the 13th international workshop on mining software repositories. ACM, New York, pp 374–385
Dunn OJ (1964) Multiple comparisons using rank sums. Technometrics 6(3):241–252
Efron B (1986) How biased is the apparent error rate of a prediction rule? In: Journal of the american statistical association, Taylor & Francis, Milton Park, vol 81, pp 461–470
Fisher RA (1925) Statistical methods for research workers. Genesis Publishing Pvt Ltd, New Delhi
Giger E, Pinzger M, Gall H (2010) Predicting the fix time of bugs. In: Proceedings of the 2nd international workshop on recommendation systems for software engineering (RSSE). ACM, New York, pp 52–56
Greer D, Ruhe G (2004) Software release planning: an evolutionary and iterative approach. Inf Softw Technol 46:243–253
Harrell FE (2001) Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis. Springer, New York
Herraiz I, German DM, Gonzalez-Barahona JM, Robles G (2008) Towards a simplification of the bug report form in eclipse. In: Proceedings of the 2008 international working conference on mining software repositories (MSR), pp 145–148
Holm S (1979) A simple sequentially rejective multiple test procedure. Scand J Stat 6:65–70
Howell DC (2005) Median absolute deviation. In: Encyclopedia of statistics in behavioral science. Wiley Online Library, Hoboken
Iasonos A, Schrag D, Raj GV, Panageas KS (2008) How to build and interpret a nomogram for cancer prognosis. In: Journal of clinical oncology, american society of clinical oncology, vol 26, pp 1364–1370
Jeong G, Kim S, Zimmermann T (2009) Improving bug triage with bug tossing graphs. In: Proceedings of the 7th joint meeting of the european software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE). ACM, New York, pp 111–120
Jiang Y, Adams B (2014) How much does integrating this commit cost? - a position paper. In: 2nd International Workshop on Release Engineering (RELENG)
Jiang Y, Adams B, German DM (2013) Will my patch make it? and how fast?: case study on the linux kernel. In: Proceedings of the 10th working conference on mining software repositories (MSR), pp 101–110
Kampstra P et al. (2008) Beanplot: a boxplot alternative for visual comparison of distributions. J Stat Softw 28:1–9
Karlsson J, Ryan K (1997) A cost-value approach for prioritizing requirements. IEEE Softw 14:67–74
Khomh F, Dhaliwal T, Zou Y, Adams B (2012) 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 (MSR). IEEE, Piscataway, pp 179–188
Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621
Leys C, Ley C, Klein O, Bernard P, Licata L (2013) Detecting outliers: do not use standard deviation around the mean, use absolute deviation around the median. J Exp Soc Psychol 49:764-766
Mäntylä MV, Adams B, Khomh F, Engström E, Petersen K (2014) On rapid releases and software testing: a case study and a semi-systematic literature review. In: Journal of Empirical Software Engineering. Springer, Berlin, pp 1–42
McIntosh S, Kamei Y, Adams B, Hassan AE (2016) An empirical study of the impact of modern code review practices on software quality. Empir Softw Eng 21(5):2146–2189
Morakot C, Hoa Khanh D, Truyen T, Aditya G (2015a) Characterization and prediction of issue-related risks in software projects. In: 12th international conference on mining software repositories (MSR), pp 280–291
Morakot C, Hoa Khanh D, Truyen T, Aditya G (2015b) Predicting delays in software projects using networked classification. In: 30th international conference on automated software engineering (ASE)
Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on software engineering (ICSE). IEEE, Piscataway, pp 284–292

Paetsch F, Eberlein A, Maurer F (2003) Requirements engineering and agile software development. In: Twelfth IEEE international workshops on enabling technologies: infrastructure for collaborative enterprises, pp 308–313
Panjer LD (2007) Predicting eclipse bug lifetimes. In: Proceedings of the 4th international workshop on mining software repositories (MSR), p 29
Rahman MT, Rigby PC (2015) Release stabilization on linux and chrome. In: IEEE software journal. vol 2. IEEE, Piscataway, pp 81–88
Regnell B, Brinkkemper S (2005) Market-driven requirements engineering for software products. In: Engineering and managing software requirements, pp 287–308
Sandelowski M (1995) Sample size in qualitative research. Res Nurs Health 18(2):179–183
Schroter A, Bettenburg N, Premraj R (2010) Do stack traces help developers fix bugs? In: 2010 7th IEEE working conference on mining software repositories (MSR). IEEE, Piscataway, pp 118–121
Schwaber K (1997) Scrum development process. In: Business object design and implementation. Springer, Berlin, pp 117–134
Sheehan KB (2001) E-mail survey response rates: a review. J Comput-Mediat Commun 6(2) https://doi.org/10.1111/j.1083-6101.2001.tb00117.x
Shihab E, Ihara A, Kamei Y, Ibrahim WM, Ohira M, Adams B, Hassan AE, Matsumoto K (2010) Predicting re-opened bugs: a case study on the eclipse project. In: Proceedings of 17th working conference on reverse engineering (WCRE). IEEE, Piscataway, pp 249–258
Shimagaki J, Kamei Y, McIntosh S, Pursehouse D, Ubayashi N (2016) Why are commits being reverted?: a comparative study of industrial and open source projects. In: 2016 IEEE international conference on software maintenance and evolution (ICSME). IEEE, Piscataway, pp 301–311
Smith E, Loftin R, Murphy-Hill E, Bird C, Zimmermann T (2013) Improving developer participation rates in surveys. In: 6th international workshop on cooperative and human aspects of software engineering (CHASE), pp 89–92
Souza R, Chavez C, Bittencourt RA (2014) Do rapid releases affect bug reopening? a case study of firefox. In: Proceedings of the brazilian symposium on software engineering (SBES). IEEE, Piscataway, pp 31–40
Souza R, Chavez C, Bittencourt R (2015) Rapid releases and patch backouts: a software analytics approach. In: IEEE software journal, vol 32. IEEE, Piscataway, pp 89–96
Spearman C (1904) The proof and measurement of association between two things. Am J Psychol 15(1):72–101
Subramaniam C, Sen R, Nelson ML (2009) Determinants of open source software project success: a longitudinal study. In: Journal of decision support systems, vol 46. Elsevier, Amsterdam, pp 576–585
Tian Y, Ali N, Lo D, Hassan AE (2015) On the unreliability of bug severity data. Empir Softw Eng 21:1–26
Wilks DS (2011) Statistical methods in the atmospheric sciences, vol 100. Academic Press, Cambridge
Zaman S, Adams B, Hassan AE (2011) Security versus performance bugs: a case study on firefox. In: Proceedings of the 8th working conference on mining software repositories. ACM, New York, pp 93–102

Daniel Alencar da Costa is a Post-doc fellow at the Queen's University, Canada. He completed his PhD and his Masters in Computer Science at the Federal University of Rio Grande do Norte (UFRN), Brazil, in 2017 and 2013, respectively. He received his Bachelor's degree in Computer Science from the University Centre of Pará (CESUPA), Brazil. His research goal is to advance the body of knowledge of Software Engineering methodologies through empirical studies using statistical and machine learning approaches as well as consulting and documenting the experience of Software Engineering practitioners.



Shane McIntosh received the bachelor's degree from the University of Guelph and the MSc and PhD degrees from Queen's University. He is an assistant professor in the Department of Electrical and Computer Engineering, McGill University, where he leads the Software Repository Excavation and Build Engineering Labs (Software REBELs). During his PhD, he held an NSERC Vanier Scholarship and received the Governor General's Academic Gold Medal. In his research, he uses empirical software engineering techniques to study software build systems, release engineering, and software quality. He actively collaborates with academics in Canada, the Netherlands, Singapore, Brazil, and Japan, as well as industrial practitioners in Germany and the USA. More about him and his work is available online at http://rebels.ece.mcgill.ca/.

Christoph Treude is a Senior Lecturer in the School of Computer Science at the University of Adelaide, Australia. He completed his PhD in Computer Science at the University of Victoria, Canada, in 2012 and received his Diplom degree in Computer Science / Management Information Systems from the University of Siegen, Germany, in 2007. The goal of his research is to advance collaborative software engineering through empirical studies and the innovation of processes and tools that explicitly take the wide variety of artifacts available in a software repository into account.

Uirá Kulesza received the bachelor's degree in computer science from Federal University of Campina Grande, the MSc degree in computer science from the University of São Paulo, and the PhD degree in computer science from Pontifical Catholic University of Rio de Janeiro (PUC-Rio). He is an associate professor in the Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte (UFRN), Brazil. His main research interests include software evolution, software architecture, and software analytics. He has published at several top-tier software engineering venues, such as the International Conference on Software Engineering (ICSE), the International Symposium on the Foundations of Software Engineering (FSE), and the European Conference on Object-Oriented Programming (ECOOP). More about him and his work is available online at http://www.dimap.ufrn.br/~uira.

Ahmed E. Hassan received the PhD degree in computer science from the University of Waterloo. He is the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair in the School of Computing, Queens University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves on the editorial boards of the IEEE Transactions on Software Engineering, the Springer Journal of Empirical Software Engineering, the Springer Journal of Computing, and PeerJ Computer Science. Contact ahmed@cs.queensu.ca. More information at: http://sail.cs.queensu.ca/.