2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

The Impact of Switching to a Rapid Release Cycle on the Integration Delay of Addressed Issues

An Empirical Study of the Mozilla Firefox Project

Daniel Alencar da Costa[1], Shane McIntosh[2], Uirá Kulesza[1], Ahmed E. Hassan[3]
[1]Federal University of Rio Grande do Norte, Natal, Brazil
danielcosta@ppgsc.ufrn.br,uira@dimap.ufrn.br
[2]McGill University, Montreal, Canada
shane.mcintosh@mcgill.ca
[3]Queen's University, Kingston, Canada
ahmed@cs.queensu.ca

ABSTRACT

The release frequency of software projects has increased in recent years. Adopters of so-called rapid release cycles claim that they can deliver addressed issues (*i.e.*, bugs, enhancements, and new features) to users more quickly. However, there is little empirical evidence to support these claims. In fact, in our prior work, we found that code integration phases may introduce delays in rapidly releasing software — 98% of addressed 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 addressed issues, we perform a comparative study of traditional and rapid release cycles. Through an empirical study of 72,114 issue reports from the Firefox system, we observe that, surprisingly, addressed issues take a median of 50 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 explain if an addressed issue will have its integration delayed or not. Our explanatory models achieve good discrimination (ROC areas of 0.81-0.83) and calibration scores (Brier scores of 0.05-0.16). Deeper analysis of our explanatory models indicates that traditional releases prioritize the integration of backlog issues, while rapid releases prioritize issues that were addressed during the current release cycle. Our results suggest that rapid release cycles may not be a silver bullet for the rapid delivery of addressed issues to users.

1. INTRODUCTION

To achieve sustained success, software projects must attract and retain the interest of users [30]. Since users will quickly lose interest in a stagnant software system, successful systems 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 have arisen to foster faster software delivery [5].[1] 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 [3]. 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 adopted shorter release cycles [1]. In this paper, we use the terms *rapid releases* to describe releases shipped using release cycles of weeks or days, and *traditional releases* to describe releases shipped using release cycles of months or years.

Prior research investigated the impact of adopting rapid releases [4, 18, 20, 28, 29]. For example, Khomh *et al.* [18] found that bugs related to crash reports tend to be fixed faster in rapid Firefox releases than traditional Firefox releases. Mäntylä *et al.* [20] 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, no previous research has empirically studied the impact that a shift from a traditional to a rapid release cycle has on the speed of integration of addressed issues. Such an investigation is important to empirically check if adopting a rapid release cycle really does lead to quicker delivery of addressed issues. In our previous work [7], we studied the delay that is introduced by the integration phase of a software project. We found that 98% of 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 addressed issues as quickly as its proponents purport.

Integration delay can be frustrating for users because they

MSR'16, May 14-15, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4186-8/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2901739.2901764

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

are mainly interested in the release of an addressed issue (so they can benefit from it) rather than if it is addressed. Anecdotally, a recent issue of the Firefox system, in which a user asks: *"So when does this stuff get added? Will it be applied to the next FF23 beta? A 22.01 release? Otherwise?"*.[2]

Hence, in this paper, we analyze 72,114 issue reports from the Firefox system (34,673 for traditional releases and 37,441 for rapid releases). These issue reports refer to bugs, enhancements, and new features [2]. We set out to comparatively study the integration delay of addressed issues in the traditional and rapid releases of the Firefox system. More specifically, we address the following research questions:

- ***RQ1: Are addressed issues integrated more quickly in rapid releases?*** Interestingly, we find that although issues are addressed more quickly in rapid releases, they tend to wait a longer time to be integrated and released to users.
- ***RQ2: Why can traditional releases integrate addressed issues more quickly?*** We find that minor-traditional releases (*i.e.*, shorter releases that occur after a major version of the software) are a key reason why addressed 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 in traditional releases — issues that are addressed early in the project backlog are less likely to be delayed. On the other hand, issues in rapid releases are queued up on a per release basis, in which issues that were addressed early in the release cycle of a given release are less likely to be delayed.

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 explain how we set up our empirical study. In Section 4, we present the results of our empirical study, while we discuss additional analyses in Section 5. Section 6 discloses the threats to the validity of our analyses. In Section 7, we discuss the related work. Finally, Section 8 draws conclusions.

2. BACKGROUND & DEFINITIONS

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 understood to being addressed.[3]

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).[4] Large 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 per day from 1999 to 2010.

When developers start working on issue reports, they use the *issue status* to track progress through an issue life cycle. In the issue life cycle, an issue is (1) reported (*new* status), (2) triaged to an appropriate developer (*assigned* status), and (3) addressed (*fixed* status). A more detailed description of the issue report life cycle of Firefox is provided in the Bugzilla documentation.[5]

In this paper, we study *addressed issues*, which are issues that are resolved with the *fixed* status and integrated into traditional and rapid releases of the Firefox system.

Firefox Release Cycles. In this paper, we study the popular Firefox web browser.[6] Firefox has approximately 18% of the worldwide market share of web browsers.[7] Firefox is a fitting subject for our study because it shifted from a traditional release cycle to a rapid release cycle.

The traditional release cycle of Firefox was applied to major releases (1.0 to 4.0). Such traditional major releases would take 12-18 months to be shipped.[8] Each major traditional release has subsequent minor releases containing bug fixes. Such minor releases may be released in parallel with other major traditional releases or even with major rapid releases. Indeed, the final minor traditional release (3.6.24) was released in tandem with major rapid release 8.

Firefox started to adopt a rapid release cycle in March 2011. The first official rapid release was shipped in June 2011. The rapid releasing Firefox ships a major release every 6 weeks. In the Firefox rapid release strategy, a release is shipped into the NIGHTLY channel every night. This NIGHTLY release incorporates the addressed issues that were integrated into the mozilla-central code repository.[9]

Releases of the NIGHTLY channel migrate to the AURORA and BETA channels to be stabilized. Once stabilized, an official release is broadcasted on the RELEASE channel. In the AURORA channel, the *quality assurance* team (QA) makes decisions of whether the code that was stabilized in AURORA can be pushed to the BETA channel.[10] Code that was further stabilized in the BETA channel is pushed to the RELEASE channel. The rapid release strategy is able to ship 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.

Moreover, the rapid release cycle of the Firefox system also includes minor releases that contain bug fixes and *Extended Support Releases* (ESR). ESRs are shipped to organizations/customers who are willing to have the latest Firefox features, but are not able to keep updating their Firefox system at the same pace that the rapid releases are shipped.[11]

3. EMPIRICAL STUDY SETUP

In this section, we provide our rationale for selecting the Firefox system for our empirical study and describe our approach to collect data from it.

[2]https://bugzilla.mozilla.org/show_bug.cgi?id=883554
[3]https://www.bugzilla.org/
[4]https://bugzilla.readthedocs.org/en/5.0/using/understanding.html

[5]https://bugzilla.readthedocs.org/en/5.0/using/editing.html#life-cycle-of-a-bug
[6]https://www.mozilla.org/en-US/firefox/new/
[7]https://clicky.com/marketshare/global/web-browsers/
[8]https://en.wikipedia.org/wiki/Firefox_release_history
[9]https://hg.mozilla.org/mozilla-central/
[10]http://mozilla.github.io/process-releases/draft/development_overview/
[11]https://www.mozilla.org/en-US/firefox/organizations/faq/

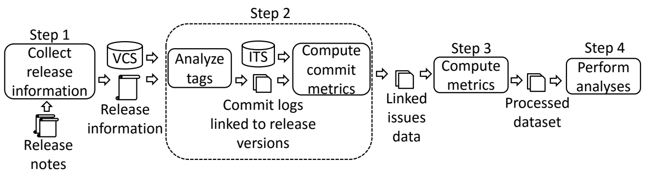


Figure 1: Overview of the process to construct the dataset that is used in our analyses.

Table 1: The studied traditional and rapid Firefox releases.

Strategy	Version range	Time period	#Major	#Minor
Trad.	1.0 - 4.0	Sep/2004 - Mar/2012	7	104
Rapid	5 - 27	Jun/2011 - Sep/2014	23	50

3.1 Firefox System

We choose to study the Firefox system 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 systems 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, to compare different systems 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 systems itself. Therefore, we highlight that the choice to investigate Firefox is not accidental, but based on the specific analysis constraints that such data satisfies.

3.2 Data Collection

Figure 1 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.

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 in 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 the respective tags. We then parse the commit logs to collect the issue IDs that are being addressed in the commits. We discard the following patterns of potential issue IDs that are false positives:

1. Potential IDs that have less than five digits, since the issue IDs of the range of the releases that we investigate have at least five digits (2,559 issues were discarded).
2. Commit logs that follow the pattern: "Bug <ID> - reftest" or "Bug <ID> - JavaScript Tests", which refer to tests and not bug fixes (269 issues were discarded).
3. Any potential ID that is the name of a file, *e.g.*, "159334.js" (607 issues were discarded).

We find that all of the remaining IDs match issue IDs that exist in the Firefox ITS.

Since the commit logs are linked to the 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 such a release is the first one that contains the complete fix for that 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 addressed issues may skip being integrated into mozilla-central (*i.e.*, NIGHTLY) tags, we manually collect the addressed issues that were integrated into those releases using the Firefox release notes (*i.e.*, 247 addressed issues).[15] We add the manually collected addressed issues from ESR releases within the rapid releases data, since they also represent data from a rapid release strategy.

Steps 3 and 4: 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 greater detail in Section 4) because we suspect that they share a relationship with integration delay.

4. RESULTS

In this section, we present the motivation, approach, results, and conclusions of our empirical analyses with respect to each of our research questions.

RQ1: Are addressed issues integrated more quickly in rapid releases?

Motivation: Recently, many software organizations have adopted rapid release cycles in order to deliver addressed issues to users more quickly. However, there is a lack of empirical evidence to indicate that rapid release cycles integrate addressed issues more quickly than traditional release cycles. In RQ1, we compare the integration delay of addressed issues in traditional and rapid releases.

Approach: Figure 2 shows the basic life cycle of an issue, which includes the triaging phase ($t1$), the fixing phase ($t2$), and the integration phase ($t3$). We consider the last RESOLVED-FIXED status as the moment that a particular issue was addressed (the fixed state in Figure 2). The *lifetime* of an issue is composed of all three phases (from *new* to *released*). 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.

We use beanplots [17] to compare the distributions of our data. The vertical curves of beanplots summarize and compare the distributions of different datasets (see Figure 3(a)).

[12]https://en.wikipedia.org/wiki/Firefox_release_history
[13]http://cvsbook.red-bean.com/cvsbook.html
[14]https://mercurial.selenic.com/
[15]https://www.mozilla.org/en-US/firefox/releases/

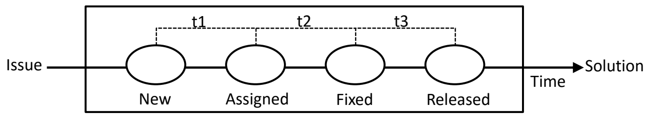


Figure 2: The basic life cycle of an issue.

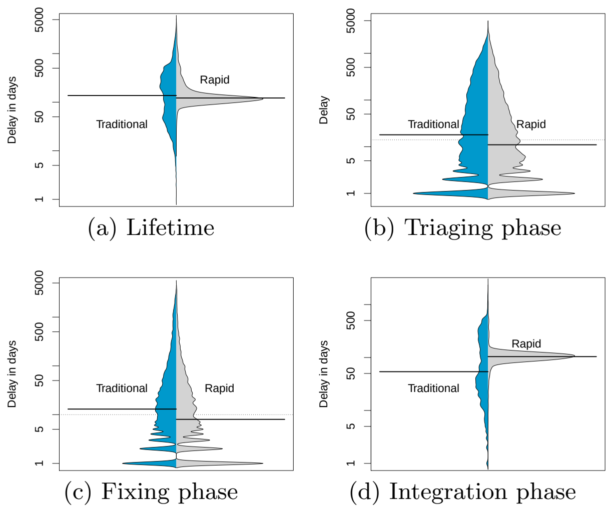


Figure 3: Time spans of the phases involved in the lifetime of an issue.

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 [31] and Cliff's delta effect-size measures [6]. 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 how often values in one distribution are larger than values in 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 they are not much different in terms of values. 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) [12, 19] 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.

Results: ***There is no significant difference between traditional and rapid releases regarding issue lifetime.*** Figure 3(a) shows the distributions of the lifetime of the issues in traditional and rapid releases. We observe a significant $p < 1.03^{-14}$ but a *negligible* ($delta = 0.03$) difference between the distributions. 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.

Addressed issues are triaged and fixed faster in rapid releases, but tend to wait for a longer time before being released. Figures 3(b) , 3(c), and 3(d) show the triaging, fixing, and integration time spans, respectively. We observe that addressed issues take 54 days on average (median) to be integrated into traditional releases, while taking 104 days (median) to be integrated into rapid releases ($p < 2.2^{-16}$ with a *small* effect-size of $delta = -0.25$).

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 ($p < 2.2^{-16}$), but there is only a *negligible* ($delta = 0.13$) difference between distributions. Our results complement previous research. Khomh *et al.* [18] 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 $p < 2.2^{-16}$ but a *negligible* ($delta = 0.11$) difference 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 Figure 2), we observe that an issue takes 11 days (median) to triage and address issues in rapid releases, while it takes 19 days (median) in traditional releases ($p < 2.2^{-16}$ with a *small* effect-size of $delta = 0.15$). Our results suggest that even though issues have shorter pre-integration phases time span in rapid releases, they remain "on the shelf" for a longer time on average.

Finally, we again observe that rapid releases are more consistent than traditional releases in terms of fixing and integration time spans. 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 tend to take a longer time to be integrated. Moreover, the integration delay is more consistent in rapid releases than in traditional ones.

RQ2: Why can traditional releases integrate addressed issues more quickly?

Motivation: In RQ1, we find that traditional releases tend to integrate addressed issues more quickly than rapid releases. This result raises the following question: how can a traditional release strategy, which has a longer release cycle, integrate addressed issues more quickly than a rapid release strategy?

Approach: We group traditional and rapid releases into major and minor releases and study their integration delay. Similar to RQ1, we use beanplots [17], Mann-Whitney-Wilcoxon tests [31], Cliff's delta [6], and MAD [12, 19] to analyze the data.

Results: ***Minor-traditional releases tend to have less integration delay than major/minor-rapid releases.*** Figure 4 shows the distributions of integration 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

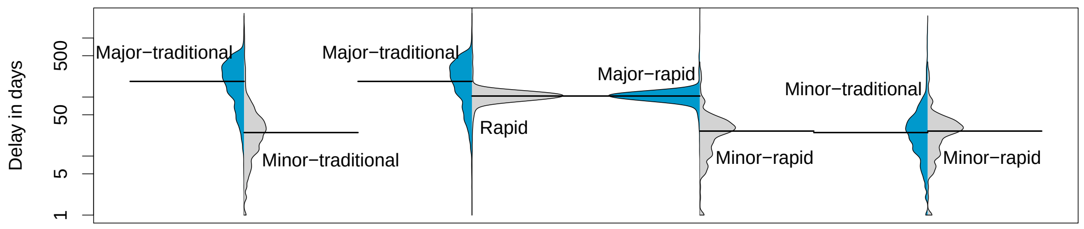
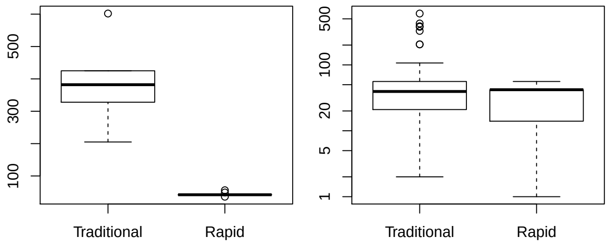


Figure 4: Distributions of integration delay of addressed issues grouped by minor and major releases.



(a) Only Major (b) Major and Minor

Figure 5: Release frequency (in days). The outliers in figure (b) represent the major-traditional releases.

of *major-traditional vs. minor-traditional*, we observe that minor-traditional releases are mainly associated with shorter integration delay. Furthermore, in the comparison *major-traditional vs. rapid*, rapid releases integrate addressed issues more quickly than major-traditional releases on average ($p < 2.2^{-16}$ 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 addressed issues more quickly than major-rapid releases (*major-rapid vs. minor-rapid*) with a $p < 2.2^{-16}$ and a *large* effect-size, *i.e.*, *delta* $= 0.92$.

Although we do not observe a statistically significant difference between distributions in the comparison of *minor-traditional vs. minor-rapid* ($p = 0.68$), it is interesting to note how minor-traditional releases tend to have shorter integration delay compared to minor-rapid releases (25 and 26, respectively).

Minor-traditional releases have the lowest integration delay. This is likely because they are more focused on a particular set of issues that, once addressed, should be released immediately. For example, the release history documentation of Firefox shows that minor releases are usually related to stability and security issues.[16]

When considering both minor and major releases, the time span between traditional and rapid releases 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. Figure 5(a) shows that, at first glance, one may speculate that rapid releases should deliver addressed issues more quickly because releases are produced more frequently. However, if we consider both major and minor releases — as shown in Figure 5(b) — 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).

Minor-traditional releases are one of the main reasons why the traditional release strategy can integrate addressed issues more quickly than the rapid release strategy. Furthermore, the length of the release cycles are roughly the same between traditional and rapid releases when both minor and major releases are considered.

RQ3: Did the change in the release strategy have an impact on the characteristics of delayed issues?

Motivation: 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 addressed 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 why traditional releases may integrate addressed issues more quickly. In RQ3, we investigate what are the characteristics of each release strategy that are associated with integration delay. This investigation is important to shed light on what may generate integration delay in each release strategy before choosing one of them.
Approach: We build explanatory models (*i.e.*, logistic regression models) for the traditional and rapid releases data using the metrics that are presented in Table 2. We model our response variable Y as $Y = 1$ for addressed issues that are delayed, *i.e.*, missed at least one release before integration [7] and $Y = 0$ otherwise. Hence, our models are intended to explain why a given addressed issue has its integration delayed (*i.e.*, $Y = 1$).

We follow the guidelines of Harrell Jr. [10] for building explanatory regression models. Figure 6 provides an overview of the process that we use to build our models. First, we estimate the budget (degrees of freedom) that we can spend on our models. Second, we check for metrics that are highly correlated using Spearman rank correlation tests (ρ) and we perform a redundancy check to remove the 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 achieved by the model. The values range between 0 (worst) and 1 (best). An area greater than 0.5 indicates that the explanatory model outperforms naïve models such as random guessing. 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.*, if I is actually delayed or not). Hence, the lower the Brier score, the more accurate the probabilities that are produced by our models.

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

Table 2: Metrics used in our explanatory models.

Dimension	Attributes	Value	Definition (d)|Rationale (r)
Reporter	Experience	Numeric	**d:** the number of previously integrated issues that were reported by the reporter of a particular addressed issue. **r:** The greater the experience of the reporter the higher the quality of his reports and the solution to his/her reports might be integrated more quickly [27].
	Reporter integration	Numeric	**d:** The median in days of the previously integrated addressed 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	Numeric	**d:** the number of previously integrated addressed issues that were addressed by the resolver of a particular addressed issue. We consider the assignee of the issue to be the resolver of the issue. **r:** The greater the experience of the resolver, the greater the likelihood that his/her code will be integrated faster [27].
	Resolver integration	Numeric	**d:** The median in days of the previously integrated addressed issues that were addressed by a particular resolver. **r:** If a particular resolver usually address issues that are integrated quickly, his/her future addressed issues might be integrated quickly as well.
Issue	Stack trace attached	Boolean	**d:** We verify if the issue report has an 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 addressed issue [26].
	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 addressed in the Eclipse project [24].
	Priority	Nominal	**d:** The priority level of the issue report. Issues with higher severity levels (*e.g.*, P1) might be integrated faster than other issues. **r:** Higher priority issues will likely be integrated before lower priority issues.
	Description size	Numeric	**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.
Project	Queue rank	Numeric	**d:** A rank number that represents the moment when an issue is addressed compared to other addressed issues in the backlog. For instance, in a backlog that contains 500 issues, the first addressed issue has rank 1, while the last addressed issue has rank 500 **r:** An issue with a high *queue rank* is an recently addressed issue. An addressed issue might be integrated faster/slower depending of its rank.
	Cycle queue rank	Numeric	**d:** A rank number that represents the moment when an issue is addressed compared to other addressed issues of the same release cycle. For example, in a release cycle that contains 300 addressed issues, the first addressed issue has a rank of 1, while the last has a rank of 300. **r:** An issue with a high *cycle queue rank* is an recently addressed issue compared to the others of the same release cycle. An issue addressed close to the upcoming release might be integrated faster.
	Queue position	Numeric	**d:** $\frac{\text{queue rank}}{\text{all addressed issues}}$. The *queue rank* is divided by all the issues that are addressed by the end of the next release. A *queue position* close to 1 indicates that the issue was addressed recently compared to others in the backlog. **r:** An addressed issue might be integrated faster/slower depending of its position.
	Cycle queue position	Numeric	**d:** $\frac{\text{cycle queue rank}}{\text{addressed issues of the current cycle}}$. The *cycle queue rank* is divided by all of the addressed issues of the release cycle. A *cycle queue position* close to 1 indicates that the issue was addressed recently in the release cycle. **r:** An issue addressed close to a upcoming release might be integrated faster.
Process	Number of Impacted Files	Numeric	**d:** The number of files 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 [16].
	Churn	Numeric	**d:** The sum of added lines plus the sum of deleted lines to address the issue. **r:** A higher churn suggests that a great amount of work was required to address the issue, and hence, verifying the impact of integrating the modifications may also be difficult [16, 23].
	Fix time	Numeric	**d:** Number of days between the date when the issue was triaged and the date that it was addressed [9]. **r:** If an issue is addressed quickly, it may have a better chance to be integrated faster.
	Number of activities	Numeric	**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 address the issue, which may impact the integration of the issue into a release [16].
	Number of comments	Numeric	**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 [9], which might impact the integration delay [16].
	Interval of comments	Numeric	**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 issue may be integrated faster.
	Number of tosses	Numeric	**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 [14, 16].

Next, we assess the stability of our models by computing the *optimism-reduced* ROC area and Brier score [8]. The optimism of each metric is computed by selecting a bootstrap *sample* to fit a model with the same degrees of freedom of

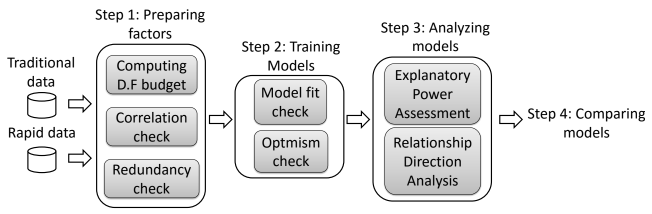


Figure 6: Overview of the process that we use to build our explanatory models.

the original model. The model built from 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 *optmism-reduced* scores by subtracting the average optimism from the initial ROC area and Brier score estimates [8].

We evaluate the impact that each metric has on the models that we fit. We use 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 the most impactful 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 each impactful metric while holding the other metrics constant at their median values using the `Predict` function in the `rms` package [10].

We also plot nomograms [10, 13] to evaluate the impact of the metrics in our models. Nomograms are user-friendly charts that visually represent explanatory models. For instance, Figure 8(a) shows the nomogram of the model that we fit for the rapid release data. The higher the number of points assigned to a 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.

Results: ***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 our statistical inferences that follow.

Traditional releases prioritize the integration of backlog issues, while rapid releases prioritize the integration of issues of the current release cycle. Table 3 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 addressed in the backlog of the project (see Table 2). Figure 7(a) shows the relationship that queue rank shares with integration delay. Our models reveal that the addressed issues in traditional releases have a higher likelihood of being delayed if they are addressed 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 addressed in a given release cycle. Figure 7(b) shows the relationship that cycle queue rank shares with integration delay. Our models reveal that the addressed issues in rapid releases have a higher likelihood of being delayed if they were addressed later than other addressed 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 7(c) shows the relationship that the *number of comments* shares with integration delay. We observe that the greater the number of comments of an addressed issue, the greater the likelihood of integration delay.

Table 3: 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		26	25
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	$\oslash$
	χ^2	1^{***}	
Resolver integration	D.F.	1	1
	χ^2	2^{***}	5^{***}
Fix time	D.F.	1	1
	χ^2	2^{***}	8^{***}
Severity	D.F.	6	6
	χ^2	1^{***}	1^{***}
Priority	D.F.	5	5
	χ^2	1^{***}	≈ 0
Size of description	D.F.	1	1
	χ^2	≈ 0	1^{***}
Stack trace attached	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
$*\ p < 0.05$; $**\ p < 0.01$; $***\ p < 0.001$

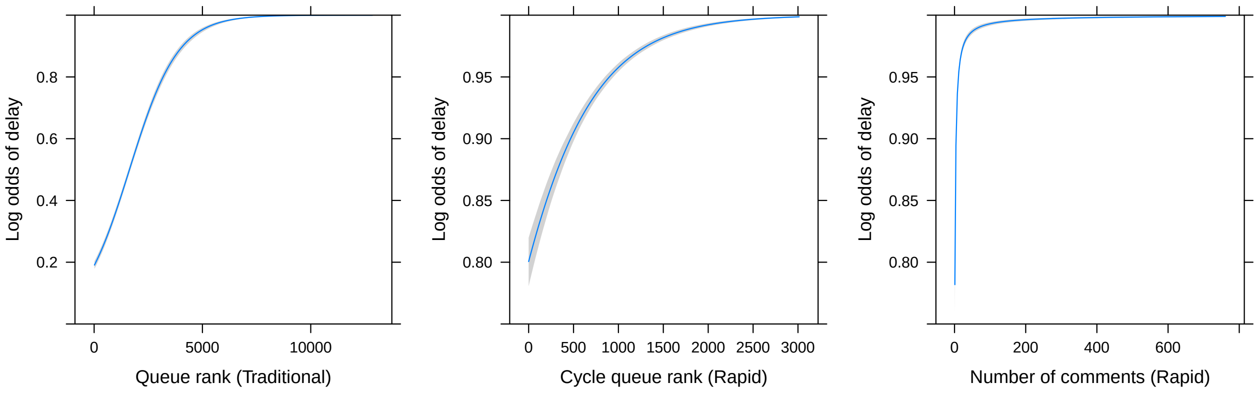
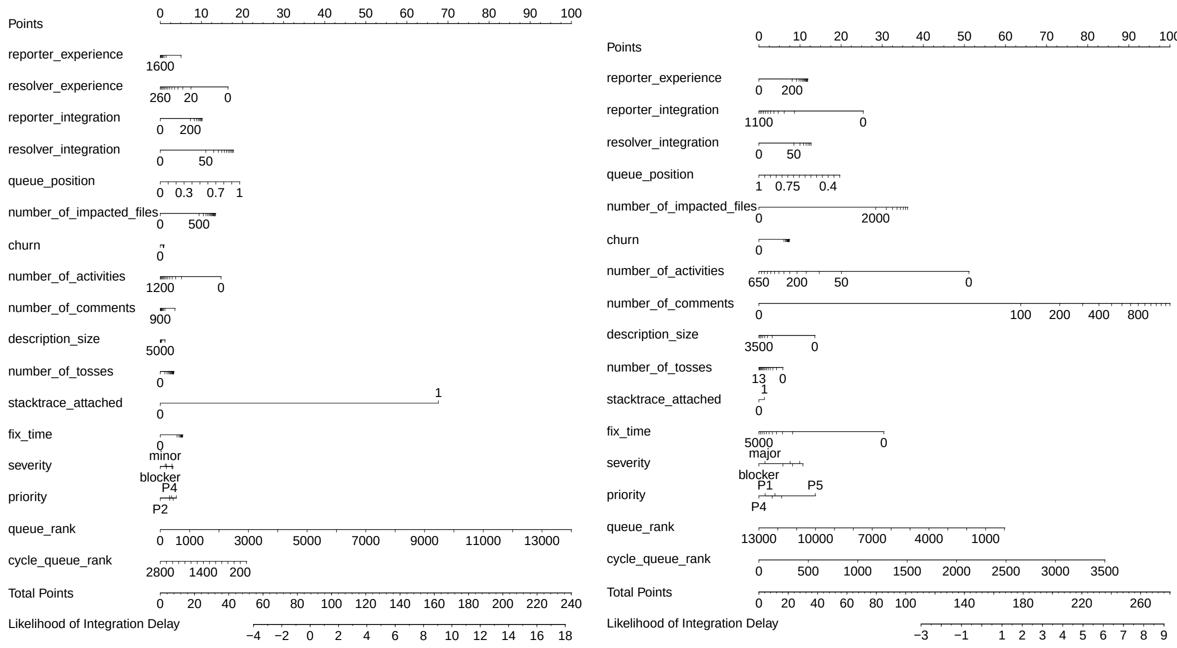


Figure 7: The relationship between metrics and integration delay. The blue line shows the values of our model fit, whereas the grey area shows the 95% confidence interval based on models fit to 1,000 bootstrap samples. The parenthesis indicate the release strategy that the metric is related to.



(a) Traditional cycle (b) Rapid cycle

Figure 8: Nomograms of our explanatory models.

This result corroborates the intuition that a lengthy discussion may be indicative of a complex issue, which may be more likely to be delayed.

Moreover, Figures 8(a) and 8(b) show the estimated effect of our metrics using nomograms [13]. 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 3). The large impact shown in our nomogram for *stack trace attached* is due to the skewness of our data, *i.e.*, only 5 instances within the traditional release data have the *stack trace attached* set to true. Thus, *stack trace attached* does not strongly contribute to the overall fit of our models.

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

Our models suggest that issues that are addressed early 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 which issues that are addressed early in the release cycle of the current release are less likely to be delayed.

5. DISCUSSION

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 addressed issues.

The integration delay of addressed issues is unlikely to be related to the size of an issue. One may suspect that the difference in integration delay between release strategies may instead 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 9 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 respectively obtain significant p values of 0.014 and $< 2.2^{-16}$, we observe *negligible* effect-sizes of $delta = -0.05$ and $delta = -0.07$ between the distributions.

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 addressed 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 10 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 addressed issues being integrated.

6. THREATS TO VALIDITY

We now describe the threats to the validity of our study.

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 system. Defects in these tools could have an influence on our results. However, we carefully tested our tools using manually-curated subsamples of the studied system, which produced consistent results.

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

The way that we link issue IDs to releases may not represent the total addressed issues per release. For example, although Firefox developers record issue IDs in commit logs, we do not know how many of the addressed issues were not recorded in the VCS. Techniques that improve the quality of the link between issue reports and commit logs could prove useful for future work.

In Section 5, 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 has noise [11] (*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 [11]. 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 that was involved in the fix of the issue.

External Validity. External threats are concerned with our ability to generalize our results. In our work, we study Firefox releases, since the Firefox system shifted from a traditional release cycle to a rapid release cycle. Although we control for variations using the same studied system in different time periods, we are not able to generalize our conclusions to other systems that adopt a traditional/rapid release cycle. Replication of this work using other systems is required in order to reach more general conclusions.

7. RELATED WORK

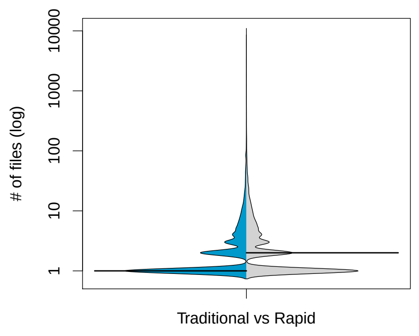
In this section, we situate our study with respect to prior work on the impact of adopting rapid release cycles and the process of integrating and delivering addressed issues.

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.* [20] 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.* [29] found that the number of reopened bugs increased by 7% when Firefox changed to a rapid release cycle. Souza *et al.* [28] 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 Firefox traditional 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.

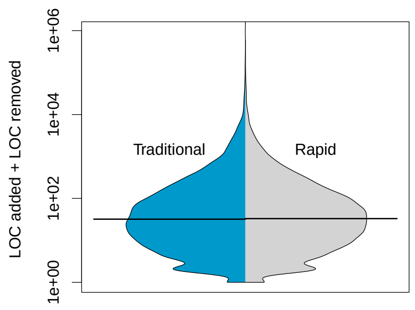
It is not clear yet if rapid releases lead to faster fixing of bugs. Baysal *et al.* [4] found that bugs are fixed faster in Firefox traditional releases when compared to fixes in the Chrome rapid releases. On the other hand, Khomh *et al.* [18] found that bugs associated to crash reports are fixed faster in Firefox rapid releases when compared to Firefox traditional releases. However, less bugs are fixed in rapid releases, proportionally. Our study corroborates that issues are addressed more quickly in rapid release cycles, but 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.* [4] found that users of the Chrome browser are more likely to adopt new versions of the system when compared to Firefox traditional releases. Khomh *et al.* [18] 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 addressed issues to users rather than user adoption of new releases.

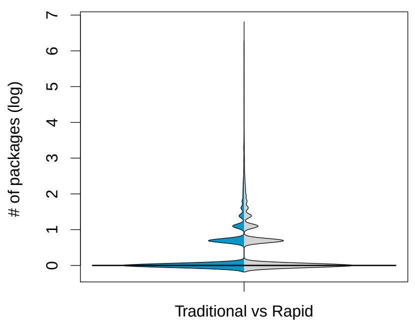
Delays and Software Issues. Prior research has studied delays related to the integration and delivery of addressed issues to end users. Jiang *et al.* [16] studied the integration process of the Linux kernel. They found that 33% of the code patches that were submitted to resolve issues are accepted into an official Linux release after 3 to 6 months. In our prior work [7], we investigate how many releases an addressed issue may be delayed before shipment. We found that 98% of addressed issues in the rapid releases of the Firefox system were delayed by at least one release. Unlike prior work [7], we investigate how the change of release strategy relates to



(a) Files



(b) LOC



(c) Packages

Figure 9: Size of the addressed issues in the traditional and rapid release data.

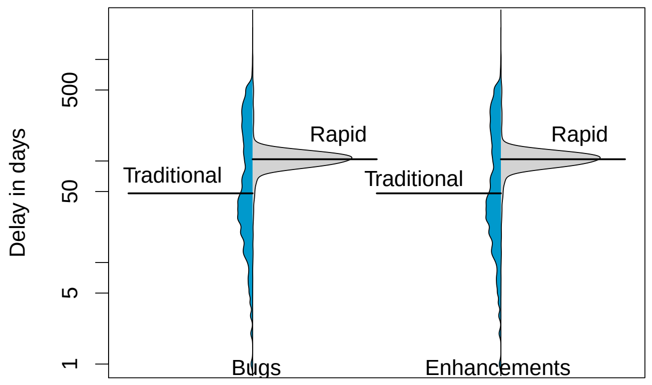


Figure 10: We group the addressed issues into "bugs" and "enhancements" by using the *severity* field. However, the difference in the integration delay between release strategies is unlikely to be related with the kind of the issue.

integration delay.

Morakot *et al.* [21, 22] study the risk of an issue to introduce delays in the delivery of software releases. 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 delay of a software release. Rahman and Rigby [25] found that the period to stabilize addressed issues can take from 45 to 93 days in the Linux kernel and from 56 to 149 days in Chrome. Jiang *et al.* [15] proposes the ISOMO model to measure the cost of integrating a new patch into a host project. Our work complements the aforementioned studies by investigating the impact that the adoption of a rapid release cycle may have upon the integration delay of addressed issues.

8. CONCLUSIONS

In this paper, we perform a comparison of the traditional and rapid releases of the Firefox system regarding integration delay. We analyze a total of 72,114 issue reports of 111 traditional releases and 73 rapid releases. We make the following observations:

- Although issues tend to be addressed more quickly in the rapid release cycle, addressed 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 addressed issues.
- We observe that the faster delivery of addressed 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 being shipped
- The triaging time of issues is not significantly different among the traditional and rapid releases.
- The total time spent from the issue report date to its integration into a release is not significantly different between traditional and rapid releases.
- In traditional releases, addressed issues are less likely to be delayed if they are addressed recently in the backlog. On the other hand, in rapid releases, addressed issues are less likely to be delayed if they are addressed recently in the current release cycle.

9. ACKNOWLEDGMENTS

This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the Brazilian Science without Borders program.

10. REFERENCES

[1] B. Adams and S. McIntosh. Modern Release Engineering in a Nutshell: Why Researchers should Care. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, page To appear, 2016.

[2] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y. Guéhéneuc. 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)*, page 23, 2008.

[3] R. Baskerville and J. Pries-Heje. Short cycle time systems development. In *Information Systems Journal*, volume 14, pages 237–264, 2004.

[4] O. Baysal, I. Davis, and M. W. Godfrey. A tale of two browsers. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, pages 238–241. ACM, 2011.

[5] K. Beck. Extreme programming explained: embrace change. Addison-Wesley Professional, 2000.

[6] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. In *Psychological Bulletin*, volume 114, page 494, 1993.

[7] D. A. da Costa, S. L. Abebe, S. McIntosh, U. Kulesza, and A. E. Hassan. An empirical study of delays in the integration of addressed issues. In *Proc. of the 30th Int'l Conf. on Software Maintenance and Evolution (ICSME)*, pages 281–290, 2014.

[8] B. Efron. How biased is the apparent error rate of a prediction rule? In *Journal of the American Statistical Association*, volume 81, pages 461–470. Taylor & Francis, 1986.

[9] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pages 52–56, New York, NY, USA, 2010. ACM.

[10] F. E. Harrell. Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis. Springer, 2001.

[11] I. Herraiz, D. M. German, J. M. Gonzalez-Barahona, and G. Robles. Towards a simplification of the bug report form in eclipse. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories (MSR)*, pages 145–148, 2008.

[12] D. C. Howell. Median absolute deviation. In *Encyclopedia of Statistics in Behavioral Science*. Wiley Online Library, 2005.

[13] A. Iasonos, D. Schrag, G. V. Raj, and K. S. Panageas. How to build and interpret a nomogram for cancer prognosis. In *Journal of Clinical Oncology*, volume 26, pages 1364–1370. American Society of Clinical Oncology, 2008.

[14] G. Jeong, S. Kim, and T. Zimmermann. 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)*, pages 111–120. ACM, 2009.

[15] Y. Jiang and B. Adams. How much does integrating this commit cost? - a position paper. *2nd International Workshop on Release Engineering (RELENG)*, 2014.

[16] Y. Jiang, B. Adams, and D. M. German. 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)*, pages 101–110, 2013.

[17] P. Kampstra et al. Beanplot: A boxplot alternative for visual comparison of distributions. In *Journal of Statistical Software*, volume 28, pages 1–9, 2008.

[18] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality? an empirical case study of mozilla firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 179–188. IEEE, 2012.

[19] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. In *Journal of Experimental Social Psychology*, volume 49, pages 764–766. Elsevier, 2013.

[20] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, and K. Petersen. On rapid releases and software testing: a case study and a semi-systematic literature review. In *Journal of Empirical Software Engineering*, pages 1–42. Springer, 2014.

[21] C. Morakot, D. Hoa Khanh, T. Truyen, and G. Aditya. Characterization and prediction of issue-related risks in software projects. In *12th International Conference on Mining Software Repositories (MSR)*, pages 280–291, 2015.

[22] C. Morakot, D. Hoa Khanh, T. Truyen, and G. Aditya. Predicting delays in software projects using networked classification. In *30th International Conference on Automated Software Engineering (ASE)*, 2015.

[23] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, pages 284–292. IEEE, 2005.

[24] L. D. Panjer. Predicting eclipse bug lifetimes. In *Proceedings of the 4th International Workshop on Mining Software Repositories (MSR)*, pages 29–, 2007.

[25] M. T. Rahman and P. C. Rigby. Release stabilization on linux and chrome. In *IEEE Software Journal*, number 2, pages 81–88. IEEE, 2015.

[26] A. Schroter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 118–121. IEEE, 2010.

[27] E. Shihab, A. Ihara, Y. Kamei, W. M. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K.-i. Matsumoto. Predicting re-opened bugs: A case study on the eclipse project. In *Proceedings of 17th Working Conference on Reverse Engineering (WCRE)*, pages 249–258. IEEE, 2010.

[28] R. Souza, C. Chavez, and R. Bittencourt. Rapid releases and patch backouts: A software analytics approach. In *IEEE Software Journal*, volume 32, pages 89–96. IEEE, 2015.

[29] R. Souza, C. Chavez, and R. A. Bittencourt. Do rapid releases affect bug reopening? a case study of firefox. In *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*, pages 31–40. IEEE, 2014.

[30] C. Subramaniam, R. Sen, and M. L. Nelson. Determinants of open source software project success: A longitudinal study. In *Journal of Decision Support Systems*, volume 46, pages 576–585. Elsevier, 2009.

[31] D. S. Wilks. Statistical methods in the atmospheric sciences. volume 100. Academic press, 2011.