Daniel Alencar da Costa

Understanding Software Delivery Delay

Brazil

2015, v-1.9.5

Daniel Alencar da Costa

Understanding Software Delivery Delay

A thesis submitted to the Department of Informatics and Applied Mathematics in conformity with the requirements for the Degree of Doctor of Philosophy

Federal University of Rio Grande do Norte – UFRN

Department of Informatics and Applied Mathematics (DIMAp)

Programa de Pós-Graduação em Sistemas e Computação

Supervisor: Uirá Kulesza

Co-supervisor: Ahmed E. Hassan

Brazil

2015, v-1.9.5

List of Figures

List of Tables

Contents

1 Introduction

Attracting and retaining the interest of users are key factors for a software system to achieve sustained success (SUBRAMANIAM; SEN; NELSON, 2009; DELONE; MCLEAN, 2003). In this context, software development teams that do not address issues that are reported by users, cause the software system to remain stagnant and lose credibility. We broadly use the term issue to either describe a *new feature*, a *bug*, or an *enhancement* that should be addressed in a software system (ANTONIOL et al., 2008).

Within a globalized world, in which technology has fostered geographically distributed software development (HERBSLEB; MOCKUS, 2003), software development teams use *Issue Tracking Systems* (ITS, e.g., Bugzilla) to coordinate tasks between the software development team.[1]

Users can use ITSs to report issues within software systems. To do so, these users must file a report that contain information about the issue (*e.g.*, the description and severity of the issue).

The basic lifecycle of an issue is the following: once reported, an issue has to be *triaged*, *i.e.*, the team member with the right expertise is assigned to the issue (ANVIK; HIEW; MURPHY, 2006). After being triaged, an issue is then *addressed* by its assignee, *i.e.*, a solution is provided for the issue and such solution is tested. Finally, the addressed issue is integrated and delivered to the end user through an official release of the software. Issues may also be *reopened* if the solution provided to the issue is found to be incorrect. In this case, a new solution has to be provided and tested.

1.1 Problem Statement

Once an issue is *addressed*, *i.e.*, a solution is provided and tested, such issue may still suffer a delay to be delivered to end users. Users care most about when addressed issues will be available in the software system (so they can get benefited from those addressed issues). We use the term *delivery delay* to refer to the delay that addressed issues suffer prior being delivered to end users.

Delivery delay can be frustrating to users. For example, in a recent issue report of the Firefox system, 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]

[1] https://www.bugzilla.org/
[2] https://bugzilla.mozilla.org/show_bug.cgi?id=883554

On the other hand, in the open source software community, developers may also be motivated to contribute because they want to see a particular feature available in the software system (JIANG; ADAMS; GERMAN, 2013). In such a case, delivery delay can also be frustrating to these contributors.

In this context, this thesis is an effort to reduce the lack of empirical studies that aim to understand why and how addressed issues suffer delivery delay before being available to users, so that such a delay can be reduced.

1.2 Current Research Limitations

Previous research has investigated the time needed to triage and to address issues (ANVIK; HIEW; MURPHY, 2005; ANBALAGAN; VOUK, 2009; GIGER; PINZGER; GALL, 2010; KIM; WHITEHEAD JR., 2006; MARKS; ZOU; HASSAN, 2011; WEIB et al., 2007; ZHANG; GONG; VERSTEEG, 2013). Such research provides valuable insight on which issues should be prioritized given the estimated time that they will take to be addressed. However, differently from what one could speculate, addressed issues still need time to be delivered to users after they are addressed.

Prior research has also be invested in the integration phase of software development. Jiang *et al.* (JIANG; ADAMS; GERMAN, 2013) studied which patches submitted to the Linux Kernel project are likely to be integrated in the main system. On the other hand, Choetkiertikul *et al.* (MORAKOT et al., 2015a; MORAKOT et al., 2015b) studied the risk of issues to postpone the shipment of new releases. However, the investigation of: (i) what leads addressed issues to suffer delivery delay even when releases are shipped and (ii) the impact of the development process in such delay remain as open challenges.

1.3 Thesis Proposal

In this thesis, we study the following general research question:

Once issues are addressed, why do they still suffer delivery delay?

Figure 1 provides an overview of the scope of this thesis. The scope shows the studies that we perform towards our general research question and the (potential) outcomes of these studies. Our studies are grouped into the *delivery delay* theme and the *stage delivery* sub-theme. The performed studies and the themes are detailed below.

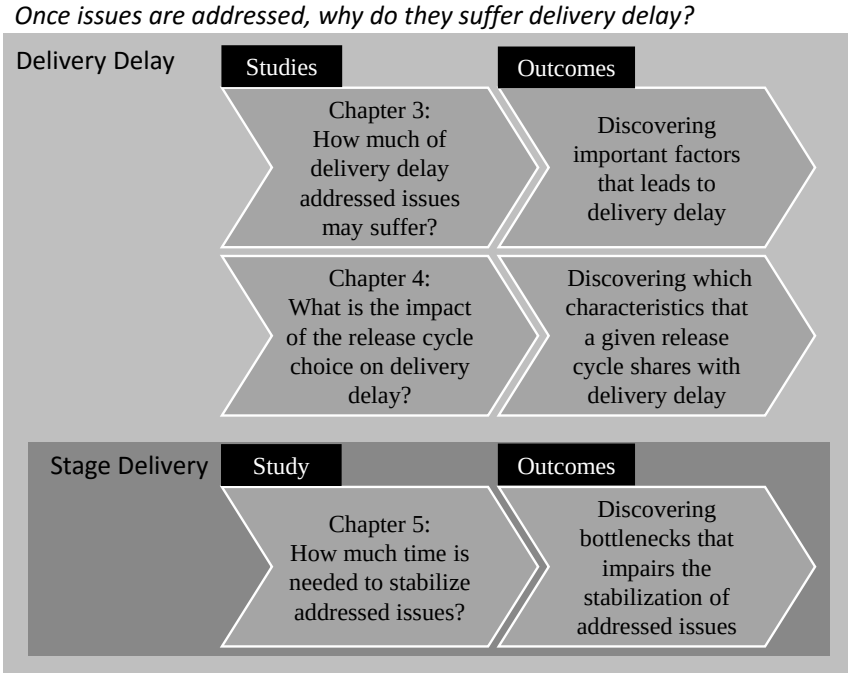


Figure 1 – An overview of the scope of the thesis.

1.3.1 Delivery Delay

Once issues are *ready*, *i.e.*, they are addressed and tested, they still suffer delays prior being delivered. Such delay is denominated as *delivery delay* in this thesis.

Chapter 3: *How much of delivery delay addressed issues may suffer?*

33% of the code patches that are submitted to resolve issues of the Linux kernel take 3 to 6 months to be accepted into an official release (JIANG; ADAMS; GERMAN, 2013). Such observation hints that the integration phase may introduce non-trivial delay before delivering addressed issues. We perform an empirical study to understand how much delivery delay addressed issues suffer and what factors may be related to such delay. We perform our analyses using data of the ArgoUML, Eclipse, and Firefox systems. In total, we analyze the delivery delay of $20,995$ addressed issues.

Chapter 4: *What is the impact of the release cycle choice on delivery delay?*

After an issue is addressed, a release must be shipped, so the users can be benefited from it. The process of shipping releases varies according to the release cycles adopted by the development team. Recently, many organizations have shifted to shorter release cycles (*i.e.*, rapid releases) with the allure of delivering software issues more quickly to end users. For instance, Firefox, Chrome, and Facebook have adopted shorter release cycles. We empirically study if shorter release cycles, in fact, deliver addressed issues more quickly to end users. We set out to study $71,114$ issue reports of the Firefox system.

1.3.2 Stage Delivery

Many organizations have been adopting the so called *pipelining releases* strategy.[3] This strategy consists on maintaining several branches (*channels*) of a given release, in which each branch represent a level of stability. For example, Google Chrome has the *dev*, *beta*, and *release* channels, where the level of stability increases from *dev* to *release*. Naturally, the greater the stability of the channel, the greater the user base that experience and test the release that is deployed in the channel.[4]

The strategy of releasing newly addressed issues to only a part of the user base has also been used by large companies such as Microsoft. For example, Windows 10 was released in "waves", in which only some early adopters would get the update by the date of the release (July 29th of 2015).[5] Such strategy enables companies to stage the delivery of addressed issues with less risk of losing credibility with end users. In this thesis, we refer to the strategy of increasing the user base of the delivery as the stability increases as *stage delivery*.

Chapter 5: *How much time is needed to stabilize addressed issues?*
Once a addressed issues are ready, they still suffer delays to be delivered. In the meantime, those addressed issues are staged through several release channels before reaching the end users. The development team takes decisions about which addressed issues should (not) make to an upcoming official release. It is important to understand how long an addressed issue takes to be stabilized and which factors play important roles to decide if such issues should be included into a more stable channel. This knowledge may be useful to improve the processes of delivering addressed issues to end users. Hence, in this chapter, we empirically study how addressed issues are stabilized in the Mozilla Firefox and Google Chrome systems.

1.4 Thesis Contributions

This thesis demonstrates that:

- 34 to 98% of addressed issues are delayed by at least one release before being delivered to users. Workload of integrators and the timing when an issue is addressed within a release cycle are the most important metrics to explain the (abnormal) integration delay of addressed issues (Chapter 3).

[3] https://www.chromium.org/developers/tech-talk-videos/release-process
[4] http://blog.mozilla.org/hacks/files/2012/05/firefox-releases.jpg
[5] http://www.theverge.com/2015/7/2/8883325/windows-10-will-roll-out-in-waves

- The priority and severity fields of addressed issues have little impact to explain integration delay. Moreover, the component of an issue do not share a relationship with integration delay (Chapter 3).
- Although rapid releases address issues faster, traditional releases integrate addressed issues more quickly on average (median). Additionally, we observe that there is not significant difference between traditional and rapid releases in the speed to address-and-deliver addressed issues, *i.e.*, from the issue creation until shipment (Chapter 4).
- Minor-traditional releases are one of the reasons why a traditional release cycle can integrate addressed issues faster than rapid releases (Chapter 4).
- Metrics related to release cycle timing can accurately explain integration delay of addressed issues in traditional and rapid releases. Nevertheless, issues in traditional releases are queued up – issues addressed earlier in the project backlog are less likely to be delayed. On the other hand, issues in rapid releases are queued on a per release basis, in which issues addressed early the current release cycle are less likely to be delayed (Chapter 4).

1.5 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 describes the background material to the reader. Chapter 3 presents our empirical study to understand the integration delay. Chapter 4 presents our analyses with respect to integration delay within traditional and rapid releases. Chapters 5 and **??** present the following research that we plan to incorporate into this thesis. Finally, Chapter 6 draws partial conclusions.

2 Background

In this chapter, we describe the definitions and key concepts to understand the empirical studies that are performed in this thesis.

2.1 Issue Reports

One of the main factors that drives software evolution are the issues that are filed by users, developers, and quality assurance personnel. Below we describe what issues are and the major steps involved in addressing and integrating them.

We use the term *issue* to broadly refer to bug reports, enhancements, and new feature requests (ANTONIOL et al., 2008). Issues can be filed by users, developers, or quality assurance personnel. To track development progress, software teams use an Issue Tracking System (ITS) such as Bugzilla or JIRA.[1,2] Such ITSs allows for describing and monitoring the status of the issue reports.

Each issue in an ITS has a unique identifier, a brief description of the nature of the issue, and a variety of other meta-data. Large software projects receive plenty of issue reports everyday. For example, our data shows that a median of 124 issues were opened per day in the Firefox project (from 1999 to 2010). The number of filed issues is usually greater than the size of the development team. After an issue has been filed, project managers and team leaders *triage* them, *i.e.*, assign them to developers, denoting the urgency of the issue using priority and severity fields (ANVIK; HIEW; MURPHY, 2006).

After being triaged, issues are then *addressed* (or *fixed* in case of bugs), *i.e.*, solutions to the described issues are provided by developers. Generally speaking, an issue may be in an open or closed status. An issue is marked as open when a solution has not yet been found. We consider UNCONFIRMED, CONFIRMED, and IN_PROGRESS as open statuses. An issue is considered closed when a solution has been found. Usually, a *resolution* is provided with a closed issue. For instance, if a developer made code changes to address an issue, the status and resolution combination should be RESOLVED-FIXED. However, if the developer was not able to reproduce the bug, then the status and resolution may be RESOLVED-WORKSFORME.[3] The issue lifecycle is thoroughly documented on the Bugzilla website.[4]

[1] https://www.bugzilla.org
[2] https://www.atlassian.com/software/jira
[3] https://bugzilla.mozilla.org/page.cgi?id=fields.html
[4] https://bugzilla.readthedocs.org/en/5.0/using/editing.html#life-cycle-of-a-bug

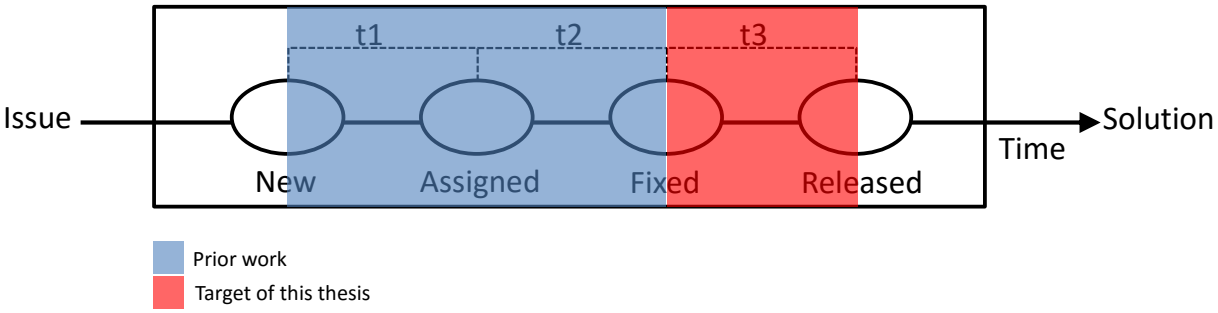


Figure 2 – The basic life-cycle of an issue.

Finally, addressed issues must be integrated into an official release in order to make them available to users. The releases that contain such addressed issues could be made available every few weeks or months, depending on the project release policy. Releasing every few weeks is typically referred to as a *rapid release* cycle, while releasing monthly or yearly is typically referred to as a *traditional release* cycle (MANTYLA et al., 2013). In the next sections, we explain in more detail the phases of the issue lifecycle that are important to understand the studies that are performed in this thesis. Figure 2 shows the phases of the lifecycle of issues. The time $t1$ is referred as *triaging* time, *i.e.*, the time required to address the right team member for the issue. We use the term *addressing* or *fixing* time to refer to the time $t2$ (*i.e.*, the time to fix and test the issue). Finally, the time $t3$ is referred as the *integration* time, *i.e.*, the time required to integrate and release the solution to end users.

Prior work has studied the triaging and fixting time of issues (blue color in Figure 2). The focus of this thesis is to study the delivery delay (red color in Figure 2), which is the delay to deliver addressed issues once they are ready.

2.2 Triaging Issues

To *triage* issues is the process of deciding which issues have to be addressed and assigning the appropriate developer to them (ANVIK; HIEW; MURPHY, 2006). This decision depends of several factors, such as the impact of the issue on the software, or how much effort is required to address the issue. Projects receive a high number of issue reports, which is usually larger than the developer team. Hence, effective triaging of issue reports is an important means of keeping up with user demands.

Hooimeijer and Weimer (HOOIMEIJER; WEIMER, 2007) built a model to classify whether or not an issue report will be “cheap” or “expensive” to triage by measuring the quality of the report. Based on their findings, the authors state that the effort required to maintain a software system could be reduced by filtering out reports that

are “expensive” to triage. Saha *et al.* (SAHA; KHURSHID; PERRY, 2014) studied long lived issues, *i.e.*, issues that were not addressed for more than one year. They found that the time to assign a developer and address such issues is approximately two years. Our research (Chapters 3 and 4) complements these prior studies by investigating the time to integrate issues once they are addressed rather than the time to assign a developer to handle the issue.

2.3 Addressing Issues

Once an issue is properly triaged, the assigned developer starts to address it. To estimate the time required to address issues, some approaches used the similarity of an issue report to existing issue reports (WEIB et al., 2007; ZHANG; GONG; VERSTEEG, 2013), while others built prediction models using different machine learning techniques (PANJER, 2007; ANBALAGAN; VOUK, 2009; GIGER; PINZGER; GALL, 2010; MARKS; ZOU; HASSAN, 2011).

Kim and Whitehead (KIM; WHITEHEAD JR., 2006) computed the time taken to address issues in ArgoUML and PostgreSQL. They found that the median issue-fix time is about 200 days. Guo *et al.* (GUO et al., 2010) used logistic regression model to predict the probability that a new issue will be fixed. The authors trained the model on Windows Vista issues and achieved a precision of 0.68 and recall of 0.64 when predicting Windows 7 issue reports. These approaches focus on estimating the time required to address an issue. In our studies (Chapters 3 and 4), however, we investigate in which release an addressed issue will be integrated.

Recent empirical studies assess the relationship between the attributes used to build models for estimating bug fix time. Bhattacharya and Neamtiu (BHATTACHARYA; NEAMTIU, 2011) performed univariate and multivariate regression analyses to capture the significance of four features in issue reports. Their results indicate that more independent variables are required to build better prediction models.

Herraiz *et al.* (HERRAIZ et al., 2008) studied the mean time to close issues reported in Eclipse, and how the severity and priority levels of the issues affect this time. In their study, the authors used one way analysis of variance to group the different priority and severity levels used in Eclipse. Based on their results, the authors suggest to reduce the severity and priority options to three levels.

Zhang *et al.* (ZHANG et al., 2012) investigated the delays incurred by developers in the issue addressing process. To do such analyses, they extract the beginning and ending time of an issue from interaction logs. The authors investigated the impact of three dimensions related to issues: issue reports, source code involved in the issue, and code changes that are required to address the issue. They found that metrics such as

severity, operating system, description of the issue, and comments are likely to impact the delays in starting to address the issue and changing the status to RESOLVED.

As Zhang *et al.* (ZHANG et al., 2012), we use attributes related to issue reports to build explanatory models (Chapters 3 and 4). However, our aim is to understand which attributes play an important role in the delay of integrating addressed issues. In addition, we investigate why severity and priority levels are not relevant to distinguish issue reports that are addressed and integrated in a release prior to others.

2.4 Integrating Issues

After issues are addressed, they need to be integrated into an official release to be available to end users. Usually, such official releases are also accompained by release notes, which are documents that specify what was added in the new release.[5] Prior research has studied the integration and delivery of addressed issues to end users. For example, Jiang *et al.* (JIANG; ADAMS; GERMAN, 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. Choetkiertikul *et al.* (MORAKOT et al., 2015a; MORAKOT et al., 2015b) studied the risk of issues introducing delays to deliver new releases of a software project.

2.5 Firefox Release Cycles

Release cycle is the time period required by the development team to develop and deliver a new release to end users. In Chapter 4, we study the impact of adopting a rapid release cycle on the integration delay of addressed issues. 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, later, with major rapid releases. Indeed, minor traditional releases were shipped until major rapid release version 8 (in that case, minor version 3.6.24).

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

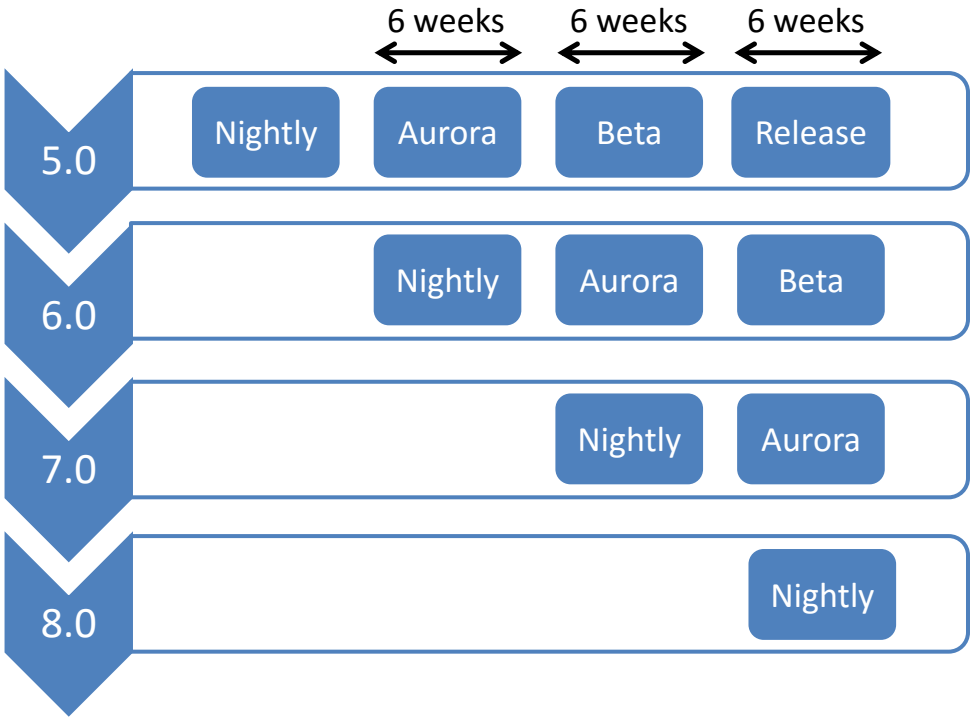


Figure 3 – Releases built into NIGHTLY channel are migrated to be stabilized into AURORA and BETA channels until it gets ready to be released into RELEASE channel.

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's 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 built into 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 enables the development of consecutive releases that are migrated from one channel to another on a regular basis (see Figure 3).

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

[9] https://hg.mozilla.org/mozilla-central/
[10] http://mozilla.github.io/process-releases/draft/development_overview/

are shipped.[11]

2.6 Chapter Summary

In this chapter, we describe the key concepts of *issue reports* and *release cycles*. An issue report may refer to either a bug-fix, a enhancement, or a new feature. The basic life cycle on an issue report consists on *triaging*, *addressing*, and *integration*.

Release cycles can be grouped into to categories: traditional releases and rapid releases. Traditional releases are releases that are shipped from a larger release cycle (*e.g.*, months or years), whereas rapid releases are releases that are shipped from a shorter release cycle (*e.g.*, days or weeks).

In the studies that we perform in this thesis, we investigate the delays that are related to the delivery of software issues to end users. We also investigate if such delays are related to the release cycle strategies that are adopted by the development team. In the following sections, we describe in details the empirical studies that are performed in this thesis.

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

3 Understanding integration delay

Earlier versions of the work in this chapter appear in the proceedings of the International Conference on Software Maintenance and Evolution (ICSME) and in the Springer Journal of Empirical Software Engineering (EMSE) (COSTA et al., 2014)

Key question: *How much of delivery delay addressed issues may suffer?*

3.1 Introduction

Prior studies have explored several approaches to help developers to estimate the time needed to address issues (new features, enhancements, and bug fixes) (ANVIK; HIEW; MURPHY, 2005; ANBALAGAN; VOUK, 2009; GIGER; PINZGER; GALL, 2010; KIM; WHITEHEAD JR., 2006; MARKS; ZOU; HASSAN, 2011; WEIB et al., 2007; ZHANG; GONG; VERSTEEG, 2013). Such studies are useful for project managers who need to allocate development resources effectively in order to deliver new releases on time without exceeding budgets.

On the other hand, users and contributors care most about when an official release of a software system will include an addressed issue. Although an issue may have been addressed, it may not be integrated into an official release for some time. Jiang *et al.* (JIANG; ADAMS; GERMAN, 2013) find that after a change has taken 1-3 months to complete the code review process, it takes an additional 1-3 months for that change to be integrated into the Linux kernel. We refer to the time between when an issue is addressed and when it is integrated into an official release as *integration delay*.

Although one can often speculate, it is not always clear why an addressed issue would not be integrated into an upcoming release. When the reasons for these integration delays are unclear, users and contributors may become frustrated.

To investigate why the integration of some addressed issues is delayed, we perform an empirical study of 20,995 issues collected from the ArgoUML, Eclipse, and Firefox systems. We investigate (1) how much delay addressed issues typically have before integration, and (2) which issues suffer from longer integration delays than

most others. To that end, this chapter addresses five research questions structured along these two themes as described below.

3.1.1 Release Integration Delay

- ***RQ1: Are issues often delayed after being addressed?*** The integration of 34% to 60% of addressed issues within traditional releasing cycles (ArgoUML and Eclipse) are delayed by at least one release. Furthermore, 98% of the addressed issues are delayed by at least one release in the rapidly released Firefox system.
- ***RQ2: Can we accurately explain how many releases an addressed issue will be delayed?*** Our models can accurately explain the integration delay of addressed issues achieving ROC areas above 0.78.
- ***RQ3: What are the most influential attributes for estimating release integration delay?*** We find that the workload of the integrators plays an influential role in estimating the release integration delay of an addressed issue. On the other hand, we find that priority and severity denoted in issue reports have little influence on release integration delay.

3.1.2 Abnormal integration delay

- ***RQ4: Can we accurately explain which addressed issues will take longer to be integrated than most others?*** In this regard, our models outperform random guessing, achieving ROC areas above 0.87.
- ***RQ5: What are the most influential attributes for estimating abnormally delayed issues?*** Similar to RQ3, we find that the moment when an issue is addressed during the release cycle and the workload of integrators are also the most influential attributes for identifying the issues that will suffer from abnormal integration delay.
- ***RQ6: Does the integration delay of addressed issues relates to the components that they are being modified?*** We find that there is no considerable difference between the number of days that the integration of an issue is delayed and the components of the studied systems.

Our results suggest that the integration backlog shares a strong link with both release and abnormal integration delay. Therefore, in addition to studying the triaging and addressing stages of the issue lifecycle, the integration stage should also be the target of research, tools, and techniques in order to reduce the time-to-delivery of addressed issues.

Table 1 – An overview of the studied systems.

System	Time frame	Releases	# of releases	# addr. issues	Median time between releases (weeks)
Eclipse (JDT)	03/11/2003 - 12/02/2007	2.1.1 - 3.2.2	11	3,344	16
Firefox	05/06/2012 - 04/02/2014	13 - 27	15	3,121	6
ArgoUML	18/08/2003 - 15/12/2011	0.14 - 0.34	17	14,530	26

3.1.3 Chapter Organization

The remainder of the chapter is organized as follows. Section 3.2 presents our empirical study by describing the studied systems, the data collection procedures, and the dimensions that we investigate in our study. Sections 3.3 and 3.4 present the results with respect to our release and abnormal integration delay dimensions. Section 3.5 discusses the role of the "code freeze" stage in integration delay. Section 3.6 discusses the threats to the validity of our conclusions. In Section 3.7, we survey the related work. Finally, Section 3.8 draws conclusions.

3.2 Study Design

In this section, we describe the studied systems, explain how the data is collected, and present an overview of the two studied themes of integration delay, *i.e.*, *release integration delay* and *abnormal integration delay*.

3.2.1 Studied Systems

In order to study integration delay, we study three subject systems: Firefox, ArgoUML and Eclipse.[1,2,3] ArgoUML is a UML modeling tool that includes support for all standard UML 1.4 diagrams. Eclipse is a popular open-source IDE, of which we study the JDT core subsystem. Firefox is a popular web browser.

Table 1 shows the studied: (i) timeframe, (ii) quantity and range of releases, and (iii) quantity of issue reports. We focus our study on the releases for which we could recover a list of issue IDs from the release notes. We collected a total of 20,995 issue reports from the three studied systems. Each issue report corresponds to an issue that was addressed and could be mapped directly to a release.

3.2.2 Database Construction

Figure 4 provides an overview of our database construction approach — how we collect and organize the data to perform our empirical study. We create a relational database describing the integration of addressed issues in the studied systems. To do

[1] <http://argouml.tigris.org/>
[2] <https://www.eclipse.org/>
[3] <http://goo.gl/Qyizd2>

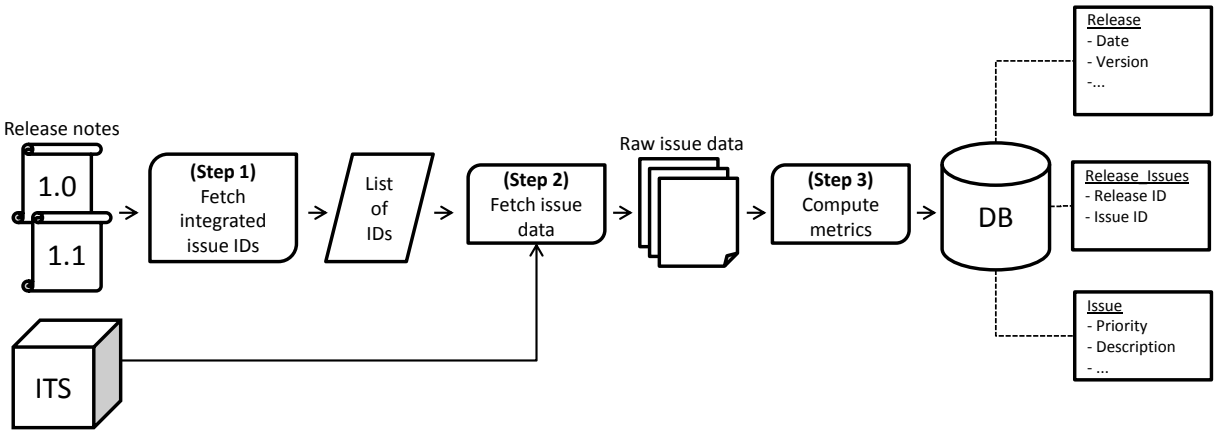


Figure 4 – An overview of our approach to construct a database for studying integration delay.

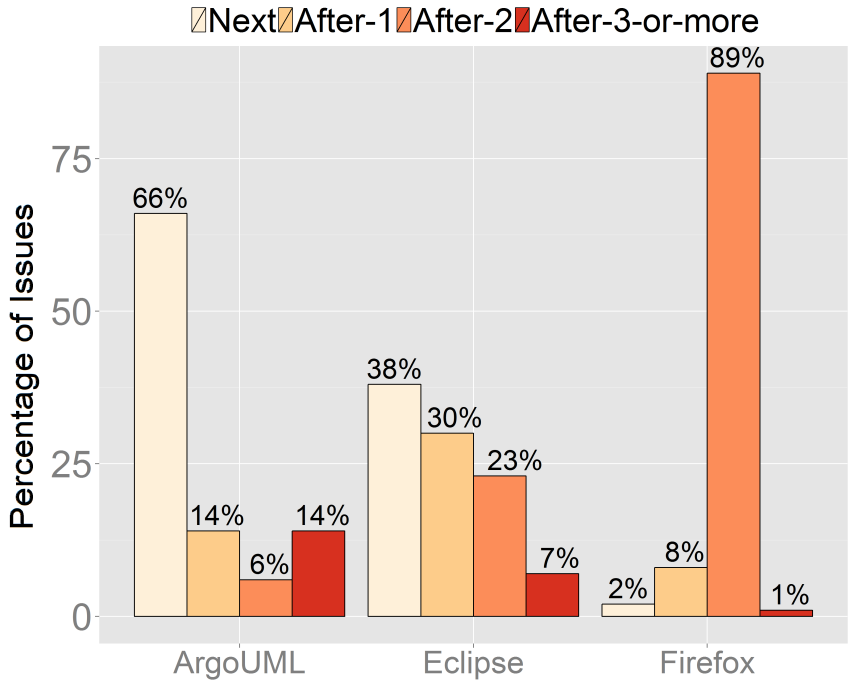


Figure 5 – Distribution of issues for each project

so, we collect data from two sources. We briefly describe each source, and each step involved in the database construction process.

3.2.2.1 Step 1: Fetch integrated issue IDs

In Step 1 we consult the release notes of each studied system to identify the release that an addressed issue was integrated into. A release note is a document that describes the content of a release. For instance, a release note might provide information about the improvements included in a release (with respect to prior releases), the new features, the fixed issues, and the known problems. Eclipse, ArgoUML, and Firefox

publish their release notes on their respective websites.[4]

Unfortunately, release notes may not mention all of the addressed issues that have been integrated into a release. This limitation hinders the investigation of issues that were addressed but have not been integrated because we cannot claim that an issue not listed in a release note was not integrated. However, the issues that are listed in a release note have certainly been integrated. Thus, we choose to use release notes despite their incompleteness to identify integrated issues in order to reduce the noise in our dataset, i.e., the release where we claim that an issue has been integrated is almost certainly correct.

The output of Step 1 is a list containing the integrated addressed issue IDs. To retrieve the list of addressed issues that have been integrated into Eclipse and Firefox, we wrote a script to extract the listed issue IDs from the release notes and insert them into our database. The retrieved issue IDs are then used to collect issue reports from the corresponding ITSs. In our database, we also store the dates and versions of the releases.

3.2.2.2 Step 2: Fetch issue data

Once we collect the IDs of the addressed issues that were integrated in each release note (Step 1), in Step 2 we fetch the data regarding each issue in the ITS system using the previous collected IDs. Not all release notes from ArgoUML list the issues that were addressed in that official release, and when they do, only a few issues are listed (*e.g.*, 1-4).[5] Hence, we rely on the ITS in order to map addressed issues to releases. We used the milestone field indicated in the issue reports to approximate the release that an issue was integrated into. Development milestones are counted towards the next official releases. For instance, the development milestones 0.33.7 is counted towards the official release 0.34. The output of Step 2 is the raw issue data collected from the ITS using the IDs fetched in Step 1.[6]

Finally, for all of our analysis, we choose the last change to the RESOLVED-FIXED status of an issue as the moment when the issue was addressed. For instance, in case an issue changes from RESOLVED-FIXED to REOPENED, and changes to the RESOLVED-FIXED status again, we consider that the issue was addressed at the last change to the RESOLVED-FIXED status. Also, we use the RESOLVED-FIXED status rather than the VERIFIED-FIXED status because we found that all of the issues that are mapped to releases went through RESOLVED-FIXED before being integrated, while only a small percentage went through VERIFIED-FIXED. For instance, only 17% of

[4] <http://goo.gl/x8htzm>
[5] <http://goo.gl/nbdXp7>
[6] <http://goo.gl/IoK56R>

addressed issues in Firefox went through the VERIFIED-FIXED status. We focus on issues that were resolved as FIXED because they involve changes to the source and/or test code that must be integrated into a release to become visible to the public.

3.2.2.3 Step 3: Compute metrics

After collecting information from each addressed issue, we compute all of the information that may be related to the integration delay themes that we investigate in our study. We investigate two themes of integration delay: (i) the *release integration delay* theme, and (ii) the *abnormal integration delay* theme. The following subsections describe each theme:

Release Integration Delay

We compute the release integration delay of each addressed issue, which we group into four classes: *next*, *after-1*, *after-2*, and *after-3-or-more*. The *next* class contains addressed issues that are integrated immediately. The *after-1*, *after-2*, and *after-3-or-more* classes contain addressed issues whose integration is delayed by one, two, or three or more releases, respectively. We use this data to address RQ1-RQ3.

Figure 5 shows the distributions of the addressed issues among the classes for each studied system. ArgoUML has the highest percentage of addressed issues that fall into the *next* class (71%), whereas *next* accounts for only 2% and 38% of addressed issues in Firefox and Eclipse, respectively.

We use exploratory models to study the relationship between the characteristics of addressed issues (*e.g.*, severity and priority) and the release integration delay. Our models are used to understand which characteristics are important for explaining the release integration delay of addressed issues.

Abnormal Integration Delay

In this theme, we study issues that have been abnormally delayed by integration for a given studied system. Again, we use exploratory models to study the relationship between the characteristics of addressed issues and the abnormal integration delay. To do so, we tag each studied addressed issue as being either *abnormally delayed* or *typically delayed*. We use the median integration delay measured in days to divide the addressed issues into these two classes. If a given addressed issue has a delay greater than the median, we classify it as *abnormally delayed*. Addressed issues with integration delay less than or equal to the median are classified as *typically delayed*. We use this data to address RQ4 and RQ5.

3.3 Release Integration Delay

In this section, we present the results of our study of *release integration delay*. This study is comprised of investigations of the integration delay measured in number of releases, which address RQ1-RQ3. We present our results with respect to each research question below.

RQ1: Are issues often delayed after being addressed?

RQ1: Motivation

Users and contributors care most about when an addressed issue will be integrated into an official release rather than when it is initially addressed. In this regard, we observe that some addressed issues are integrated in the next release, while others are delayed.

It is not clear why some addressed issues take more time to be integrated than others. In RQ1, we investigate the delay between when an issue is addressed and when it is integrated. In order to better understand integration delay in each studied system, we also investigate if the delays differ between rapid and traditional release cycles. The analysis of RQ1 is our first step toward understanding why integration delays differs among addressed issues.

RQ1: Approach

We compute the *integration delay* of an addressed issue as shown in Figure 6. We first collect the time when the resolution status of each issue was changed to *RESOLVED-FIXED* from the ITS. To determine the moment of integration, we analyze the release notes of each project. Finally, we count the number of releases that occurred between the time when an issue status changed to the *RESOLVED-FIXED* and the release that it was integrated into.

RQ1: Results

Addressed issues are usually delayed in the Firefox studied system. Figure 7 shows the difference between the studied systems regarding the time interval between their releases. The median time in days for Firefox (42 days) is approximately $\frac{1}{4}$ of that of ArgoUML (180 days), and $\frac{1}{3}$ of that of Eclipse (112 days). Unlike for Eclipse and Firefox, the distribution for ArgoUML is skewed. In addition, Figure 5 shows that the vast majority of addressed issues for Firefox are integrated *after-2* releases, whereas for Eclipse and ArgoUML, the majority are integrated in the *next* release.

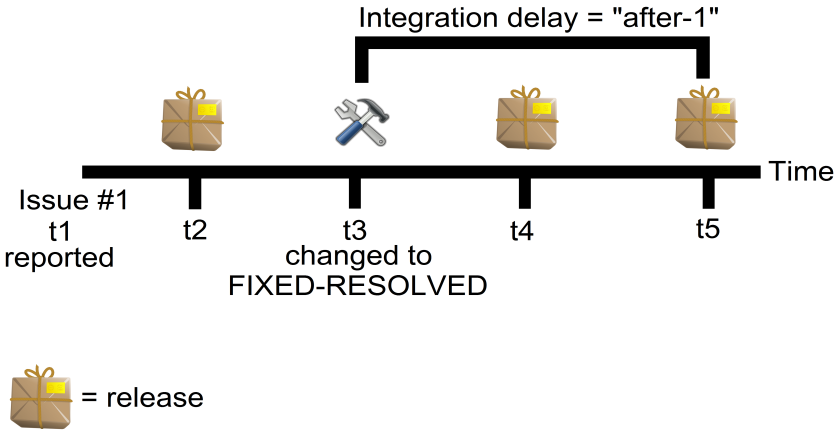


Figure 6 – Integration delay is computed by counting the releases that occur between when an issue status changes to RESOLVED-FIXED and the the date of the release note that lists that issue.

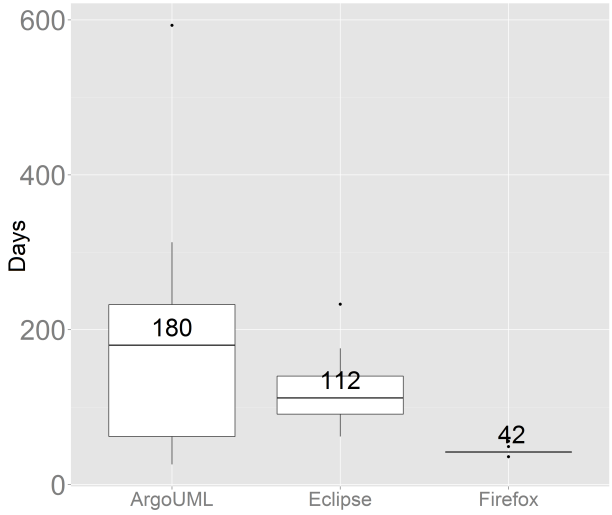


Figure 7 – Delays in days between releases of ArgoUML, Eclipse, and Firefox. The number shown over each boxplot is the median interval

The reason for the difference may be the release policies followed in each project. For example, Figure 7, shows that Firefox releases consistently every 42 days (six weeks), whereas the times between ArgoUML releases vary from 50 to 220 days. Indeed, the ArgoUML releasing guidelines state that the ArgoUML team should release at least one stable release every 8 months (≙ 240 days).[7] The consistency of Firefox releases may lead to more delayed issues, since they rigidly adhere to a six-week release schedule despite accumulating issues that could not be integrated.

34% to 60% of addressed issues in the traditional release cycle systems were delayed by one or more releases. Figure 5 shows that 98% of the addressed issues in Firefox are delayed by one or more releases. We conjecture that Firefox is more likely to have delayed issues due to its rapid releasing cycle. However, 98% is still a large

[7] <http://argouml.tigris.org/wiki/Strategic_Planning>

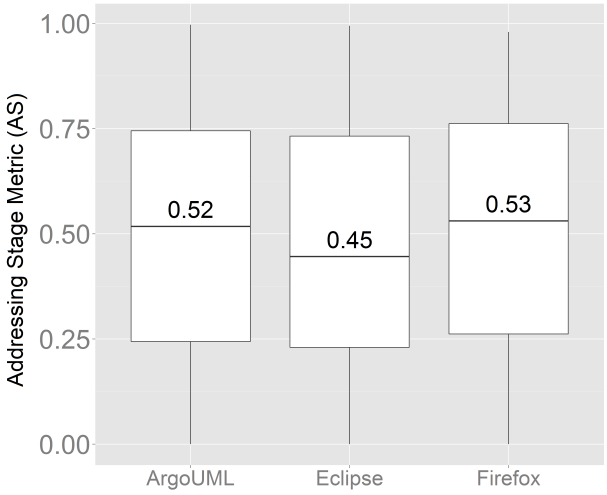


Figure 8 – Distribution of days between when an issue was addressed and the next missed release divided by the release window time.

percentage. Furthermore, even for the systems that adopt a more traditional release cycle, 34% (ArgoUML) to 60% (Eclipse) of the addressed issues are delayed by at least one release. This result indicates that even though an issue is addressed, integration could be delayed by one or more releases.

Many delayed issues were addressed well before releases from which they were omitted. Addressed issues could be delayed from integration because they were addressed late in the release cycle, *e.g.*, one day or one week before the upcoming release date. In order to compare the rapid and traditional release cycles regarding whether delayed issues are addressed late in the release schedule, we computed the *Addressing Stage* metric (*AS*) for each issue.

The *AS* metric is calculated using the following equation: $\frac{days\ to\ next\ release}{release\ window}$, where *days to next release* is the number of days that an issue is addressed before the next release (*e.g.*, the time between *t3* to *t4* in Figure 6), and the *release window* is the time in days between the next upcoming release and the previous release (*e.g.*, *t2* to *t4*). An *AS* value close to 0 means that an issue was addressed too close to the next release, whereas a value close to 1 means that an issue was addressed at the beginning of a release cycle.

Figure 8 shows the distribution of the *AS* metric for each project. The smallest *AS* median is observed for Eclipse, which is 0.45. For ArgoUML and Firefox, the median is 0.52 and 0.53, respectively. The *AS* medians are roughly in the middle of the release. Moreover, the boxes extend to cover between 0.25 and 0.75. The result suggests that, in the studied projects, delayed issues are usually addressed $\frac{1}{4}$ to $\frac{3}{4}$ of the way through a release. Hence, it is unlikely that most addressed issues miss the next release solely because they were addressed too close to an upcoming release date.

The integration of 34% to 60% of the addressed issues in the traditionally releasing systems and 98% in the rapidly releasing system were delayed by one or more releases. Furthermore, we find that many delayed issues were addressed well before releases from which they were omitted from.

RQ2: Can we accurately explain how many releases an addressed issue will be delayed?

RQ2: Motivation

Several studies have proposed approaches to investigate the time required to address an issue (ANBALAGAN; VOUK, 2009; GIGER; PINZGER; GALL, 2010; KIM; WHITEHEAD JR., 2006; MARKS; ZOU; HASSAN, 2011; WEIB et al., 2007; ZHANG; GONG; VERSTEEG, 2013). These studies could help to estimate when an issue will be addressed. However, we find that an addressed issue may be delayed before being delivered to users. Even though most issues are addressed well before the next release date, many of them are not integrated until a future release. For users and contributors, however, knowing the release in which an addressed issue will be integrated is of great interest. In RQ2, we investigate if we can accurately explain how many releases an addressed issue will be delayed. Our explanatory models are important to understand which variables may impact in the integration delay of addressed issues. Moreover, our models could estimate for users and contributors when an addressed issue will likely be integrated.

RQ2: Approach

In order to study when an addressed issue will be integrated, we collected information from both ITSs and VCSs of the studied systems. We build models using metrics from the following families: *reporter*, *issue*, *project*, and *process*.

- **Reporter** refers to the reputation of an issue reporter. Issues reported by a reporter who is known to report important issues may receive more attention from the integration team.
- **Issue** refers to reported issues. Project teams use this information to triage, address, and integrate issues. For example, integrators may not be able to properly assess the importance and impact of poorly described issues, which may, in turn, lead to integration delays.
- **Project** refers to the status of the project when a specific issue is addressed. If the project team has a heavy integration workload, *i.e.*, many addressed issues waiting to be integrated, the integration of newly addressed issues may be delayed.

Table 2 – Reporter and Issue metrics used to explain the integration delay of an addressed issue

Family	Attributes	Value	Definition (d)|Rationale (r)
Reporter	Experience	Numeric	**d:** Experience in filing reports for the project. It is measured by the number of previously reported issues of a reporter. **r:** An issue reported by an experienced reporter might be integrated quickly.
	Delay of previously addressed issues	Numeric	**d:** Measured by the median of the integration delays of previous issues that were reported. **r:** If previously addressed issues were integrated quickly for a reporter, future issues reported by the same reporter may also be integrated quickly.
Issue	Component	Nominal	**d:** The component specified in the issue report. **r:** Issues related to a given component (*e.g.*, authentication) might be more important, and thus, might be integrated prior to issues in less important components.
	Platform	Nominal	**d:** The platform specified in the issue report. **r:** Issues regarding one platform (*e.g.*, MS Windows) might be integrated prior to issues in less important platforms.
	Severity	Nominal	**d:** The severity of the issue. **r:** Issues with higher severity levels (*e.g.*, blocking) might be integrated faster than other issues. Panjer observed that the severity of an issue has a large effect on its lifetime for Eclipse project (PANJER, 2007).
	Priority	Nominal	**d:** The priority of the issue. **r:** Higher priority issues will likely be integrated before lower priority issues.
	Stack trace attached	Boolean	**d:** We verify if the issue report has an stack trace attached in its description. **r:** A stack trace attached in the issue reported may provide useful information regard the cause of the issue, which may quicken the integration of the addressed issue (SCHROTER; BETTENBURG; PREMRAJ, 2010).
	Description Size	Numeric	**d:** Description of the issue measured by the number of the words. **r:** Issues that are well-described might be more easy to integrate than issues that are difficult to understand.

- **Process** refers to the process of addressing an issue. An addressed issue that involved a complex process (*e.g.*, long comment threads, large code changes) could be difficult to understand and integrate.

Tables 2 and 3 describe the information that we collect in each family. Henceforth, we refer to the collected information as attributes. For each attribute, Tables 2 and 3 presents the type and the rationale behind its use in our models.

Table 3 – Project and Process metrics used to explain the integration delay of an addressed issue

Family	Attributes	Value	Definition (d)|Rationale (r)
Project	Integration Workload	Numeric	**d:** The number of issues in the RESOLVED-FIXED state at a given time.
			r: Having a large number of addressed issues at a given time might create a high workload on integrators, and may affect the number of addressed issues that are integrated.
	Queue position	Numeric	**d:** $\frac{\text{rank of the issue}}{\text{all addressed issues}}$, where the rank is the position in time when an issue was addressed in relation to others in the current release cycle. The rank is divided by all the issues addressed by the end of the release cyle.
			r: An issue that is near the front of the queue is more likely to be integrated quickly.
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 (JIANG; ADAMS; GERMAN, 2013).
	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 necessary to address the issue, which can impact the integration of the issue into a release. (JIANG; ADAMS; GERMAN, 2013).
	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 (GIGER; PINZGER; GALL, 2010), which might impact the integration delay. (JIANG; ADAMS; GERMAN, 2013).
	Number of Tosses	Numeric	**d:** The number of times the assignee has changed.
			r: The number of changes in the issue assignee might indicate a complex issue to address or a difficulty in understanding the issue, which can impact the integration delay. One of the reasons for changing the assigned developer is because additional expertise may be required to address the issue (JIANG; ADAMS; GERMAN, 2013; JEONG; KIM; ZIMMERMANN, 2009).
	Comment Interval	Numeric	**d:** The sum of all of the time intervals between comments (measured in hours) divided by the total number of comments.
			r: A short comment time interval indicates that an active discussion took place, which suggests that the issue is important. (JIANG; ADAMS; GERMAN, 2013).
	Churn	Numeric	**d:** The sum of the added lines and removed lines in the code repository.
			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 (NAGAPPAN; BALL, 2005; JIANG; ADAMS; GERMAN, 2013).
	Issue addressing time	Numeric	**d:** Measured by number of days between when the opening date of the issue and the moment when the issue changed to RESOLVED-FIXED (GIGER; PINZGER; GALL, 2010).
			r: If issues are addressed quickly, there is also a good change that such addressed issues might be integrated quickly.

Explanatory model. We train our models using the *random forest* technique (BREIMAN, 2001), which is known to have a good overall accuracy and to be robust to outliers as well as noisy data. Model robustness is important for our study because the data in the ITSs are filed with subjective criteria and tend to be noisy (HERRAIZ et al., 2008). In our study, we use the *random forest* implementation provided by the *bigrf* R package[8]. To build and test our explanatory models, we use a 10-fold cross-validation and 100 trees in each forest.

Evaluation metrics. We use *precision*, *recall*, *F-measure*, and *ROC area* to evaluate our models. We describe each metric below.

Precision (P) measures the correctness of our models in estimating the release delay of an addressed issue. An estimation is considered correct if the estimated integration delay is the same as the actual integration delay it had. Precision is computed as the proportion of correctly estimated integration delays for each class (*e.g.*, next, after-1).

Recall (R) measures the completeness of a model. A model is considered complete if all of the addressed issues that were integrated in a given release r are estimated to appear in r. Recall is computed as the proportion of issues that actually appear in a release r that were correctly estimated as such.

F-measure (F) is the harmonic mean of precision and recall, *i.e.*, $(\frac{2\times P\times R}{P+R})$. F-measure combines the inversely related precision and recall values into a single descriptive statistic.

ROC area is used to evaluate the degree of discrimination achieved by the model. The ROC area is the area below the curve plotting the true positive rate against false positive rate. The value of ROC area ranges between 0 (worst) and 1 (best). An area greater than 0.5 indicates that the explanatory model outperforms a random predictor. We computed the ROC area for a given class c (*e.g.*, *next*) on a binary basis. In other words, the probabilities of the instances were analyzed as pertaining to a given class c or not for each class. Therefore, each class has its own ROC area value.

RQ2: Results

Our explanatory models achieve a weighted average precision between 0.25 to 0.86 and a recall between 0.72 to 0.92. Figure 9 shows the precision, recall, F-measure, and ROC area of our explanatory models. The boxplots represent the distributions of the ten folds that we perform for each class.

The best precision values that we obtain for Eclipse, Firefox, and ArgoUML are related to the *next* (median of 0.81), *after-2* (median of 0.99), and *next* (median of 0.98),

[8] Bigrf package http://goo.gl/fyyd33

respectively. However, for classes with low number of instances the precisions decrease considerably. For instance, the median precision obtained in the Firefox data for the remaining classes (*next*,*after-1*, and *after-3-or-more*) are very low (median of 0.05).

On the other hand, the obtained recall values tend to be high in classes where the number of instances are the minority. For example, the highest medians for recall in ArgoUML are for *after-2* and *after-3-or-more*, whereas in Firefox the highest medians are for *next* and *after-3-or-more*.

Moreover, we obtained ROC areas of 0.78 to 0.93 on average (median), which indicate that our model estimations are better than random guessing (ROC area of 0.5). Summarizing the results, we obtained an weighted average of the medians for precision between 0.25 to 0.86 and for recall between 0.72 to 0.92. Although there is room for improvement, our models provide a sound starting point for explaining the release that an addressed issue will be integrated into.

Our models achieve better F-measure values than Zero-R. We compared our models to Zero-R models as a baseline. For all test instances, Zero-R selects the class that contains the majority of the instances. Hence, the recall for the class containing the majority of instances is 1.0. We compared the F-measure of our models to the F-measure of Zero-R models. We choose to compare to the F-measure values because precision and recall are very skewed for Zero-R.

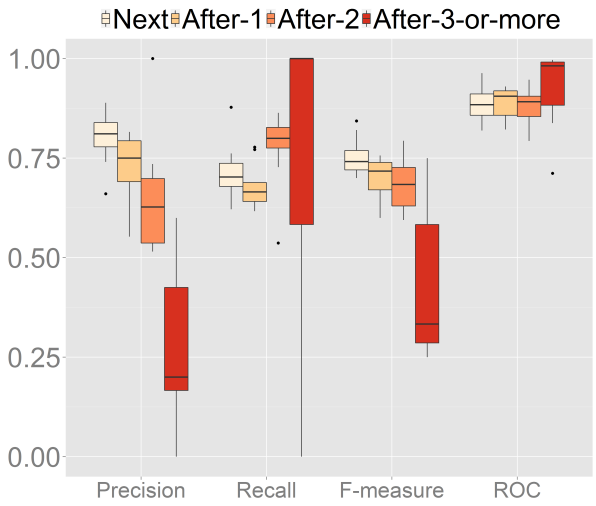
For Firefox, Zero-R has an F-measure of 0.95 for the class *after-2*, which was equal to our model. For Eclipse, Zero-R always selects *next* and achieves an F-measure of 0.58 while our model achieves 0.70. Finally, for ArgoUML, Zero-R selects always *next* with an F-measure of 0.84, whereas our model achieves 0.85. These results show that our models yield better F-measure values than naive techniques like Zero-R or random guessing (ROC = 0.5) in the majority of cases.

We are able to accurately explain how many releases an addressed issue is likely to be delayed. Our models outperform naive techniques such as Zero-R and random guessing, achieving an ROC area of above 0.78.

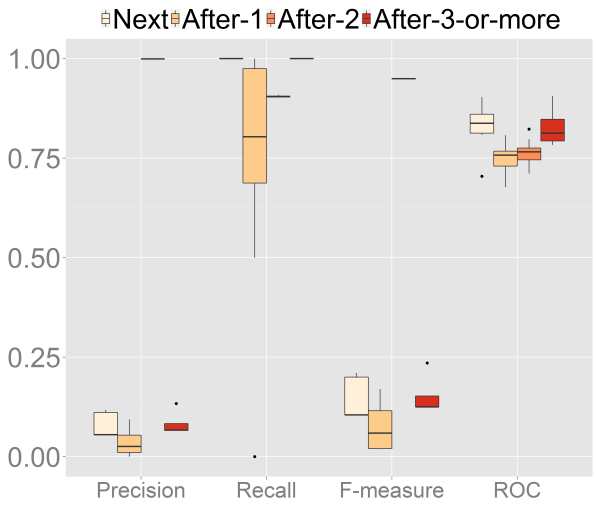
RQ3: What are the most influential attributes for estimating release integration delay?

RQ3: Motivation

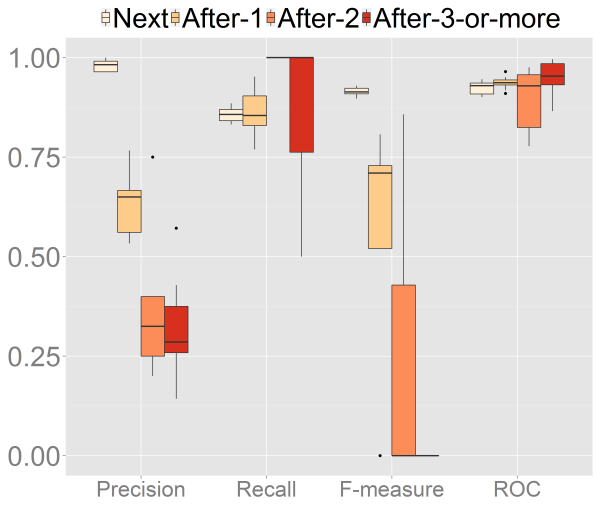
In RQ2, we found that our models can accurately explain the release integration delay of addressed issues. To build the models, we use attributes collected from ITSs and VCSs. As described in Tables 2 and 3, the attributes belong to different families of



(a) Eclipse



(b) Firefox



(c) ArgoUML

Figure 9 – The performance of our random forest models.

relationships with the addressed issues. In RQ3, we investigate which attributes are influential in estimating the release integration delay of an addressed issue.

RQ3: Approach

To identify the most influential attribute that estimate integration delay of an addressed issue, we compute *variable importance* for each attribute in our models. The *variable importance* implementation that we use in our study is available in the *bigrf* R package. This implementation computes the importance of an attribute based on *out of the bag* (OOB) estimates. Each attribute of the dataset is randomly permuted in the OOB data. Then, the average a of the differences between the votes for the correct class in the permuted OOB and the original OOB is computed. The result of a is the importance of an attribute.

The final output of the variable importance is a rank of the attributes indicating their importance for the model. Hence, if a specific attribute has the highest rank, then it is the most influential attribute that our explanatory model is using to estimate integration delay.

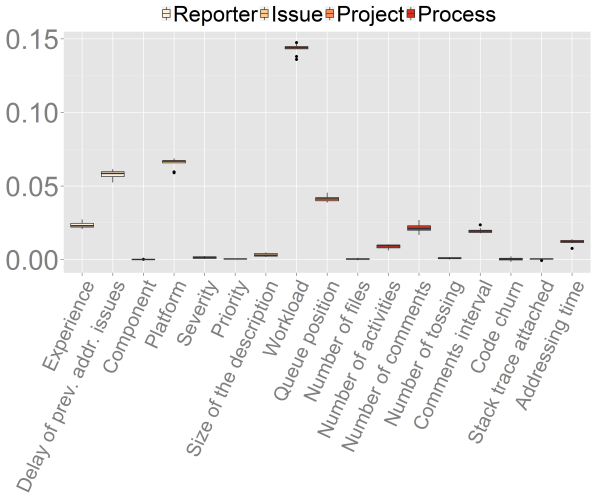
Finally, to compute a statistically stable ranking of the attributes, we use the Scott-Knott technique (SCOTT; KNOTT, 1974), which clusters the attributes based on the importance scores obtained by our random forest models. First, two groups of statistically distinct scores are divided. Then, the Scott-Knott technique recursively divides those groups until no statistically distinct groups can be created.

RQ3: Results

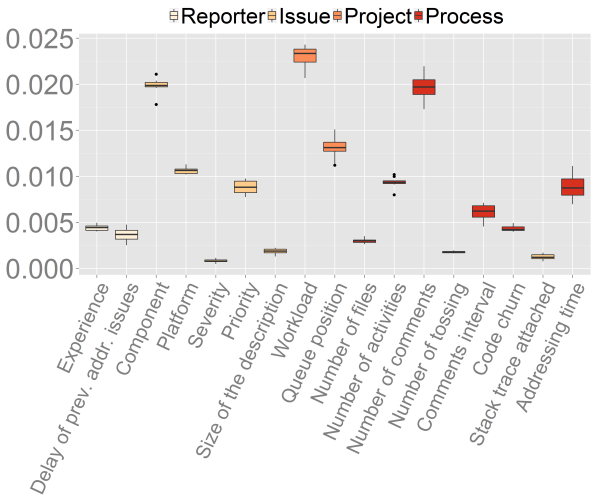
The workload of integrators is the most influential attribute. By *integrators* we refer to team members that are responsible for integration tasks (JIANG; ADAMS; GERMAN, 2013) (*e.g.*, Eclipse's release engineering team).[9] Figure 10 shows the distribution of the variable importance values computed for the ten folds of our models. The most influential attributes is the workload, which measures the amount of addressed issues still not integrated when a given issue is addressed. These results suggest that the integration backlog introduces overhead that a software team must manage. Otherwise, the integration backlog may lead to integration delays.

In addition, Table 4 shows the Scott-Knott results for the importance scores of the attributes in each of the models. Indeed, the workload of integrators is in the top group of significance in the ArgoUML,Eclipse, and Firefox systems. These results further emphasize the importance of the integration backlog, since it may lead to delays in the integration of future issues.

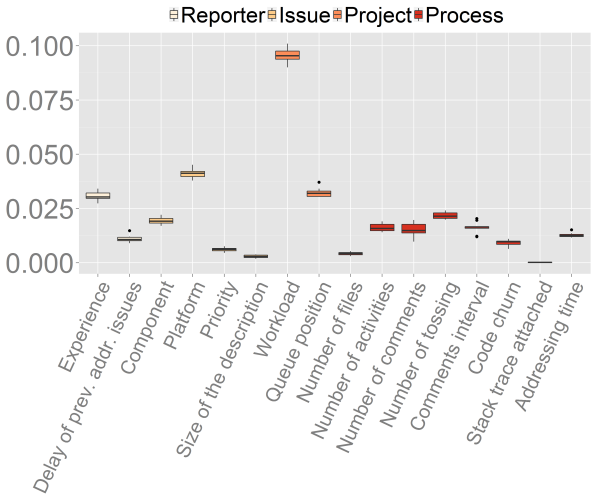
[9] <http://goo.gl/q4vZzl>



(a) Eclipse



(b) Firefox



(c) ArgoUML

Figure 10 – Distributions of variable importance values computed for the 10 folds performed to train the models.

Table 4 – Scott-Knott test results for the importance of the attributes in the explanatory models. The attributes are divided into groups that have a statistically significant difference in the mean of importance scores (p-value < 0.05)

Eclipse		Firefox		ArgoUML	
Group	**Attribute**	**Group**	**Attribute**	**Group**	**Attribute**
1	Workload	1	Workload	1	Workload
2	Platform	2	Component	2	Platform
3	Delay of prev. addr. issues		Number of comments	3	Queue position
4	Queue position	3	Queue position		Experience
5	Experience	4	Platform	4	Number of tossing
6	Number of comments	5	Number of activities	5	Component
7	Comments interval		Addressing time	6	Comments interval
8	Addressing time		Priority		Number of activities
9	Number of activities	6	Comments interval		Number of comments
10	Size of the description	7	Experience	7	Addressing time
11	Severity		Code churn	8	Delay of prev. addr. issues
	Number of tossing	8	Delay of prev. addr. issues	9	Code churn
	Priority		Number of files	10	Priority
	Number of files	9	Size of the description	11	Number of files
	Stack trace attached		Number of tossing		Size of the description
	Code churn	10	Stack trace attached	12	Stack trace attached
	Component		Severity		

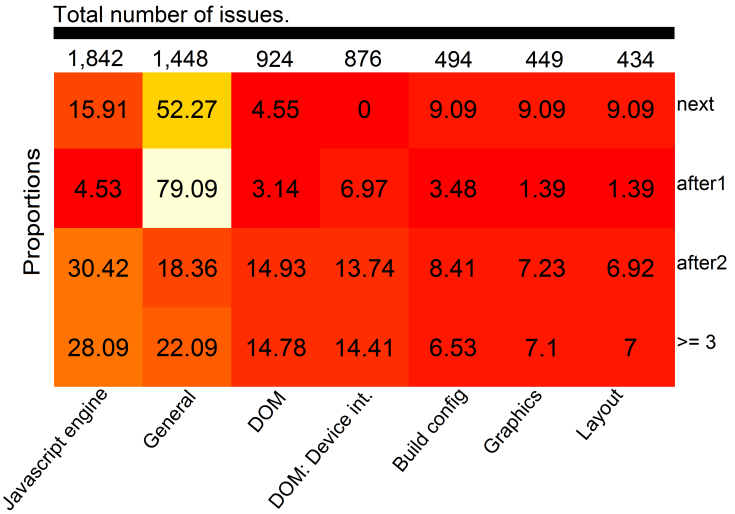
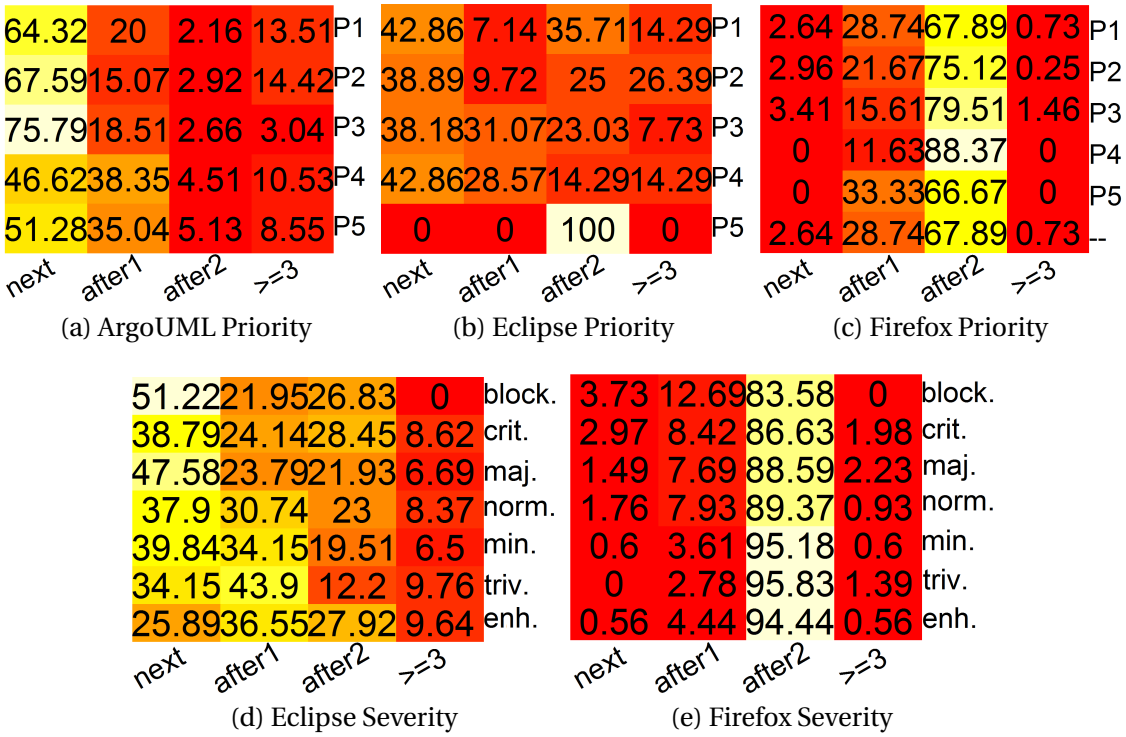


Figure 11 – The spread of issues among Firefox components. The darker the colors, the smaller the proportion of issues that impact that component.

Furthermore, in the Firefox system, *component* and *comments* are also influential attributes with scores close to workload. To better understand why *component* have such influence, we study the distribution of addressed issues across components. Figure 11 shows the top seven Firefox components, each having more than 400 addressed issues. We analyze the proportion of delayed integration in the top seven components. Figure 11 shows that, for classes *next* and *after-1*, the majority of issues are related to the *General component*, whereas for *after-2* and *after-3-or-more* the majority are related to the *Javascript engine* component. Addressed issues related to the *General* component may be easy to integrate, whereas issues related to the *Javascript Engine* may require more careful analysis before integration.



(a) ArgoUML Priority (b) Eclipse Priority (c) Firefox Priority

(d) Eclipse Severity (e) Firefox Severity

Figure 12 – The percentage of priority and severity levels in each issue class. We expect to see light colour in the upper left corner of these graphs, indicating that high priority/severity issues are integrated rapidly. Surprinsingly, we are not seeing such a pattern in our datasets.

Severity and priority have little influence on release integration delay. Users and contributors of software projects can denote the importance of an issue using the *priority* and *severity* fields. Previous studies have shown that priority and severity have little influence on bug fixing time (HERRAIZ et al., 2008; MOCKUS; FIELDING; HERBSLEB, 2002). For example, while an issue might be severe or of high priority, it might be complex and would take a long time to fix.

However, in the integration context, we expect that priority and severity would play a bigger role, since the issue has already been addressed. One would expect that integrators would try to fast-track the integration of such high priority and severe issues. For instance, according to Eclipse guidelines for filing issue reports, priority value P1 is used for serious issues and specifies that the existence of a P1 issue should prevent a release from shipping.[10] Hence, it is surprising that priority and severity play such a small role in determining the release in which an issue will appear in. Indeed, Table 4 shows that the priority and severity metrics appear in the 5th-11th Scott-Knott importance ranks in the studied projects.

We performed an additional analysis to investigate how integration delays are related to *priority* and *severity* among the studied projects and why they had such little

[10] <http://goo.gl/XuR43g>

influence in our models. Figure 12 shows the percentage of issues with a given priority (*y-axis*) in a given delay class (*x-axis*). Note that the integration of 36% to 97% of priority P1 addressed issues were delayed for at least one release, whereas the percentages for P2 were 32% to 96%.

In ArgoUML, while the majority of priority P1 issues (64%) were integrated in the *next* release, 36% of them were delayed by at least one release. For Firefox, 97% of the P1 issues and 96% of the *blocker* issues were delayed by at least one release. Finally, for Eclipse, 57% of P1 issues and 49% of blocker issues were delayed by at least one release. Hence, our data shows that, in the context of issue integration, *priority* and *severity* have little influence on integration delay.

The workload of the integration stage is the most influential attribute in our models. We also find that priority and severity have little influence in estimating integration delay. Indeed, 36% to 97% of top priority (P1) addressed issues were delayed by at least one release.

3.4 Abnormal Integration Delay

In this section, we present the *abnormal delay* theme, which comprises our investigations regarding addressed issues that are delayed abnormally in a given studied system. Such theme encompasses RQ4 and RQ5.

RQ4: Can we accurately explain which addressed issues will take longer to be integrated than most others?

RQ4: Motivation

In Section 3.3, we analyze integration delay with respect to the number of releases that an addressed issue is delayed. Additionally, we study systems with different releasing cycles. For instance, in the Firefox system, we observed that addressed issues usually get delayed — 89% of the addressed issues have an integration delay of two releases (*after-2*). These results indicate that what is an abnormal integration delay in one studied system may be normal in another. Hence, we set out to complement our previous findings by investigating if we can accurately explain if an addressed issue will be abnormally delayed.

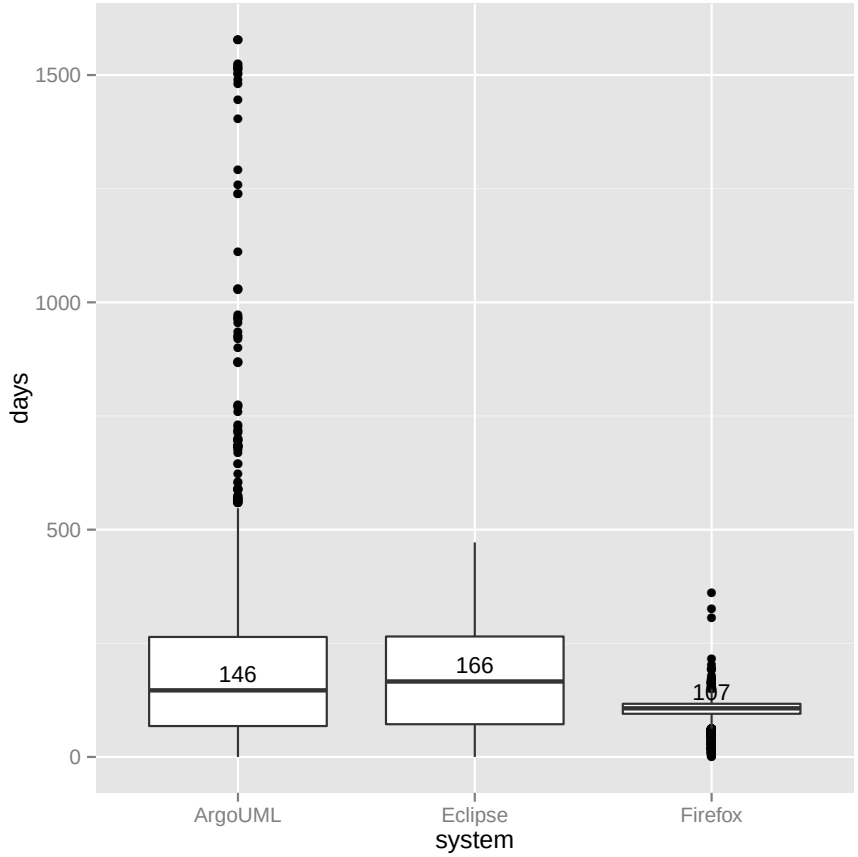
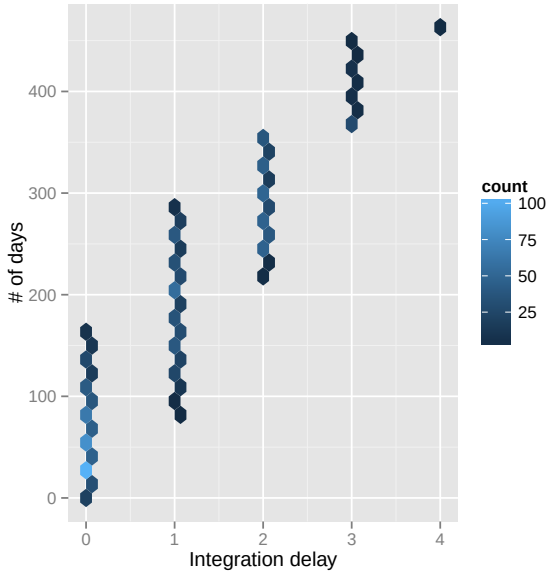


Figure 13 – Distribution of the integration delay of the addressed issues measured in days.

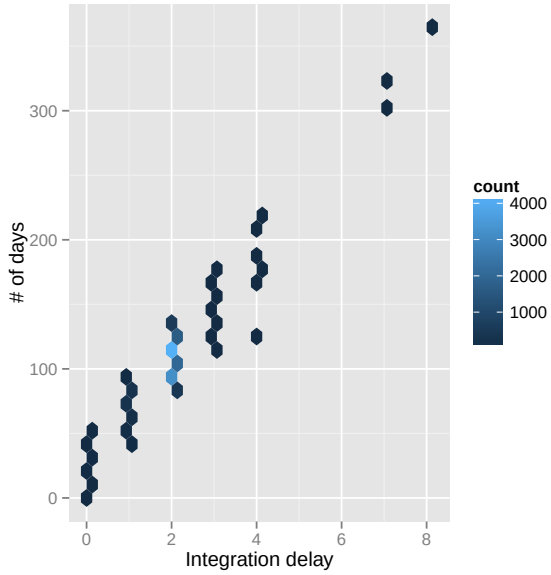
RQ4: Approach

In order to study the average delay of the addressed issues, we count the number of days between: (1) the moment when an addressed issue is changed to RESOLVED-FIXED, and (2) the date of the release note that such addressed issue is listed. Figure 13 shows the distribution of the integration delays measured in days for each studied system. We observe that the data of Eclipse and ArgoUML systems are more skewed than the data of Firefox.

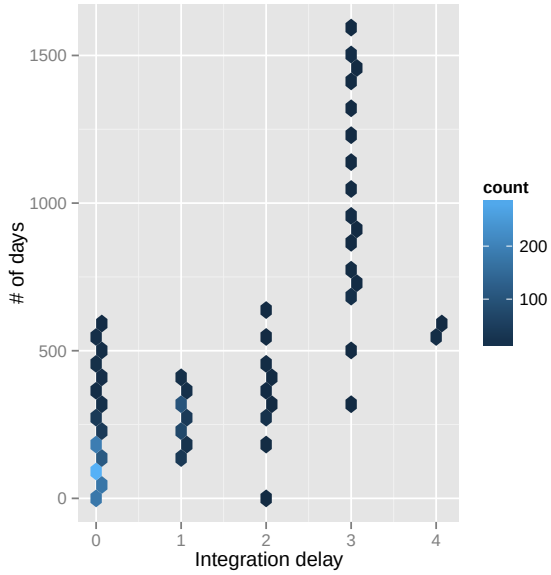
Next, we use the hexbin plots of Figure 14 to investigate the relationship between the delay measured by number of days and the delay measured by number of releases. Hexbin plots are scatterplots that represent several data points with hexagon-shaped bins. The lighter the shade of the hexagon, the more data points that fall within the bin. Indeed, Figure 14 suggests that the greater the number of days, the greater is the delay in releases. This tendency is more clear in the Eclipse and Firefox systems than the ArgoUML one. However, in ArgoUML, we observe addressed issues with a longer release delay but with a smaller delay in days. For instance, we observe addressed issues with an release delay of four releases that have a shorter integration delay in days than addressed issues with an release delay of three releases. Such behaviour in



(a) Eclipse



(b) Firefox

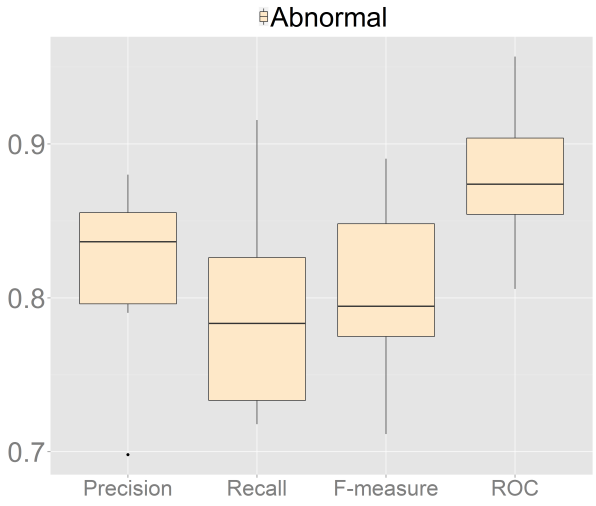


(c) ArgoUML

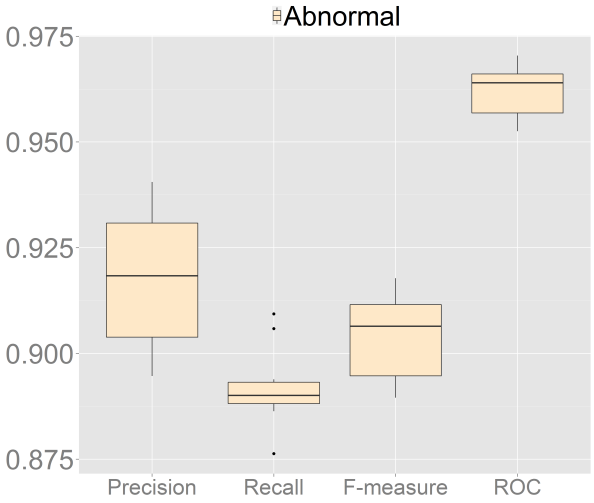
Figure 14 – Relationship between integration delay measured by number of days and number of releases.

the ArgoUML data may be explained by the skew in the distance between the releases of this studied system (*cf.* Figure 7).

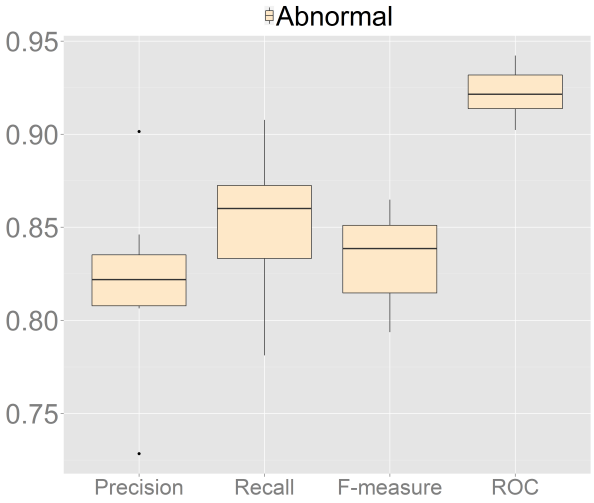
After analyzing the integration delay measured in days, we label addressed issues as abnormally delayed by computing the median in days of the integration delay in each of the studied systems. If a given addressed issue has an integration delay greater than the median, we classify it as abnormally delayed. Thus, we produce a dichotomous response variable Y, where $Y = 1$ means that the addressed issue was



(a) Eclipse



(b) Firefox



(c) ArgoUML

Figure 15 – The performance of our random forest models.

abnormally delayed, and $Y = 0$ means otherwise.

Finally, we build random forest models to explain if a given addressed issue will be abnormally delayed. Similar to RQ2, we evaluate our models using *precision*, *recall*, *f-measure*, and *ROC*.

RQ4: Results

Our models obtained a median precision of 0.82 to 0.91 and a median recall of 0.78 to 0.89. Figure 15 shows the results that we obtained using our explanatory models. Our Firefox models obtained the higher precision (median of 0.91), achieving a median F-measure of 0.90. On the other hand, our models built for ArgoUML obtained a median precision of 0.82, achieving a median F-measure of 0.78. Moreover, Our obtained ROC areas are of at least 0.87. These results indicate that our models outperform random

guessing (ROC area of 0.50).

Our models achieve better F-measure values than Zero-R. We also compared the F-measures of our models to the F-measures of Zero-R. For the Eclipse, Firefox, and ArgoUML datasets, Zero-R obtained F-measures of 0.33, 0.33, and 0.45, respectively. On the other hand, our models obtained F-measures of 0.79, 0.90, and 0.78, respectively. These results indicate that our models vastly outperform naive techniques, such as Zero-R.

We are able to accurately explain when an addressed issue will be abnormally delayed. Our models outperform naive techniques, such as Zero-R and random guessing, achieving ROC areas of at least 0.87 (median).

RQ5: What are the most influential attributes for estimating abnormally delayed issues?

RQ5: Motivation

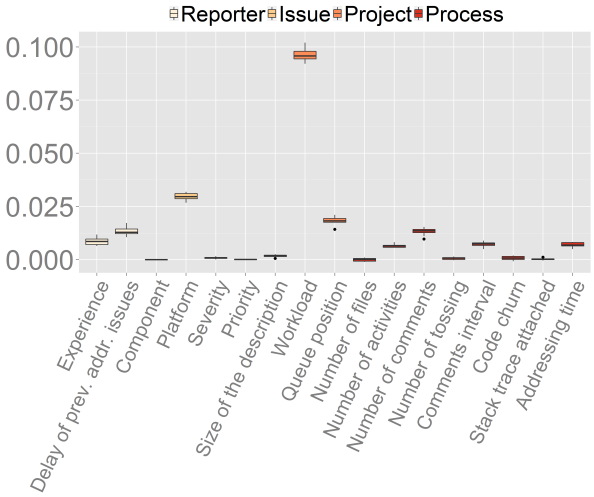
RQ4 shows that we can accurately explain if an addressed issue will be abnormally delayed or not. However, it is also important to understand what attributes are more influential to identify abnormally delayed issues — from which variables our models derive the most explanatory power.

RQ5: Approach

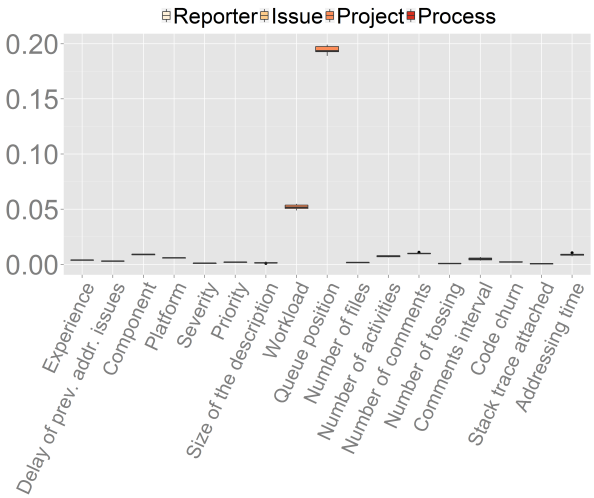
Similar to RQ3, in this research question, we analyze our explanatory models by: (1) computing the variable importance scores of our attributes, and (2) using the Scott-Knott test to identify the most influential groups of attributes.

RQ5: Results

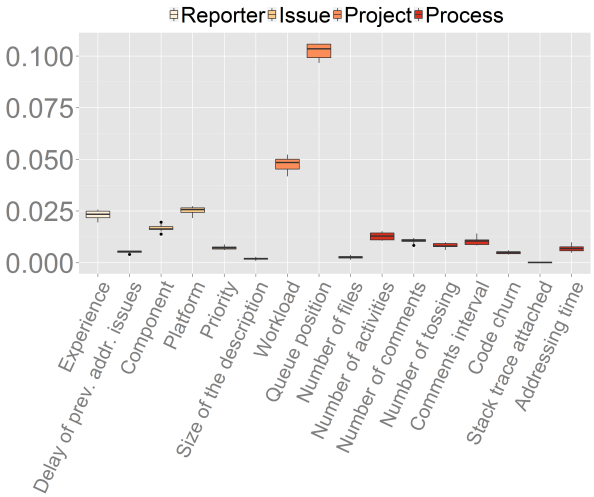
The queue position of an addressed issue and the workload of integrators are the most influential attributes. Figure 16 shows the importance scores of the 10 folds that we compute for our random forest models. We observe that the attribute "queue position" is the most influential for our Firefox and ArgoUML models. Such results corroborate with the intuition that, if an addressed issue is in the front of the queue of addressed issues in the release cycle, it is more likely to be integrated earlier. On the other hand, the "workload" is the most influential attribute for our Eclipse model and the second most influential in our Firefox model. These results corroborates with our previous finding that the integration backlog introduces an overhead that the software team needs to manage.



(a) Eclipse



(b) Firefox



(c) ArgoUML

Figure 16 – Distributions of variable importance values computed for the 10 folds performed to train the models.

Table 5 – Scott-Knott test results for the importance of the attributes in the explanatory models. The attributes are divided into groups that have a statistically significant difference in the mean of importance scores (p-value < 0.05)

Eclipse		Firefox		ArgoUML	
Group	**Attribute**	**Group**	**Attribute**	**Group**	**Attribute**
1	Workload	1	Queue position	1	Queue position
2	Platform	2	Workload	2	Workload
3	Queue position	3	Number of comments	3	Platform
4	Delay of prev. addr. issues	4	Component	4	Experience
	Number of comments	4	Addressing time	5	Component
5	Experience	5	Number of activities	6	Number of activities
6	Comments interval	6	Platform	7	Number of comments
	Addressing time		Comments interval		Comments interval
	Number of activities	7	Experience	8	Number of tossing
7	Size of the description		Delay of prev. addr. issues		Priority
	Severity	8	Code churn		Addressing time
	Code churn		Priority	9	Delay of prev. addr. issues
	Number of tossing		Number of files		Code churn
	Stack trace attached		Size of the description	10	Number of files
	Priority	9	Severity		Size of the description
	Component		Number of tossing	11	Stack trace attached
	Number of files		Stack trace attached		

By performing Scott-Knott tests, we observe that the "queue position" and "workload" are in the top groups of significance. These results suggest that abnormal integration delay is more closely related to characteristics of the release cycle than characteristics of the reporter, the issue report, or fixing process.

The queue position of the addressed issues and the workload of the integrators are the most influential attributes in identifying issues whose integration will be abnormally delayed.

RQ6: Does the integration delay of addressed issues relates to the components that they are being modified?

RQ6: Motivation

In RQ3, we find that *component* is one of the most important variables in our explanatory models of the Firefox system. We also observe that *JavaScript Engine* has the higher proportions of *after-2* and *after-3* release delays. While in the *abnormal delay* dimension, we are analysing integration delay measured in days, it is also important to investigate the relationship between this delay and the components that have been modified. Such information could be used to help users to better understand when they should expect an issue of interest to be addressed.

RQ6: Approach

We group each addressed issue according to the components that it modifies. Since ArgoUML and Firefox have more than 50 components, we focus our analyses on a subset of components with the greatest number of addressed issues of those studied systems. We then compare the distribution of integration delay in days in these components.

RQ6: Results

Figure 17 shows the distribution of integration delay per component for each studied system. We find that the distributions do not have a considerable difference in terms of integration delay in the ArgoUML and Firefox data. The boxplots for the components "General" and "Other" are more skewed, which is suggestive of their generic role — such components may encompass a more broad spectrum of addressed issues. On the other hand, 99% of the addressed issues in Eclipse (JDT) belongs to the "Core" component (thus its skewness). Finally, the "Debug" and "Text" Eclipse components contain only one addressed issue each.

Our component analysis reveals that there is no considerable difference between integration delays when grouped by components.

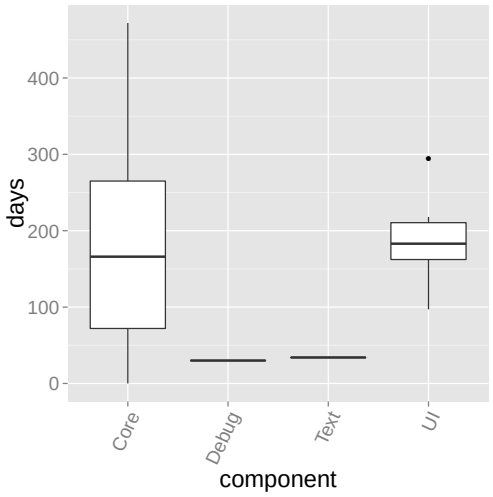
3.5 Exploratory Data Analysis

In this section, we discuss the "code freeze" stage of a release cycle. The "code freeze" stage is a period when the rules to make changes in the software system becomes more strict. For instance, new changes may only be integrated if it will solve special requirements such as translations or documentation fixing.[11] Such a period is important because it helps the development team to stabilize the project just before creating an official release.

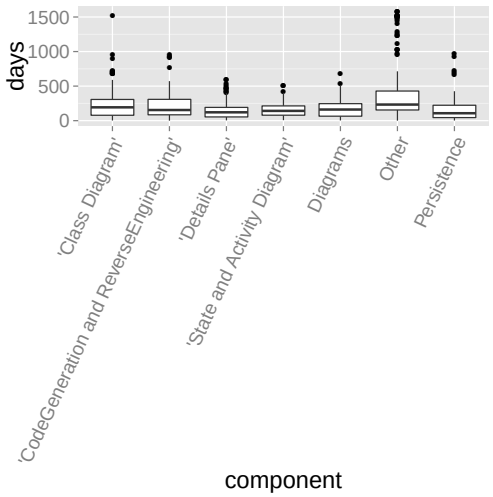
In Section 3.4, we compute the integration delay by counting the number of days between when an issue is addressed and when its integration occurs. Furthermore, in RQ1, we find that addressed issues are unlikely to be delayed solely because they are addressed too close to an upcoming release.

To further study the integration delay, we correct our analysis for the "code freeze" period, *i.e.*, we check if the addressed issues which integration is delayed are also addressed right before the "code freeze" stage of a release cycle.

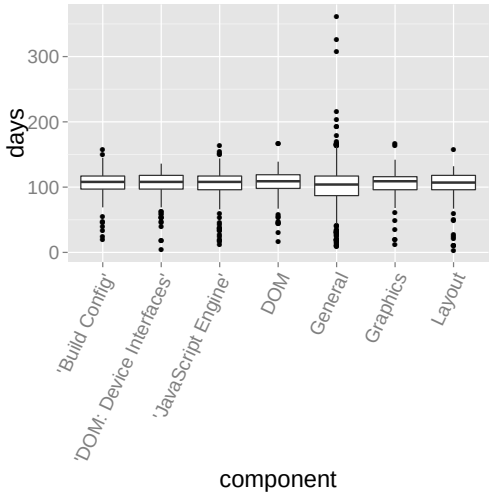
[11] <http://eclipse.org/eclipse/development/plans/freeze_plan_4_4.php>



(a) Eclipse



(b) Firefox



(c) ArgoUML

Figure 17 – Distributions of integration delay measured in days per components.

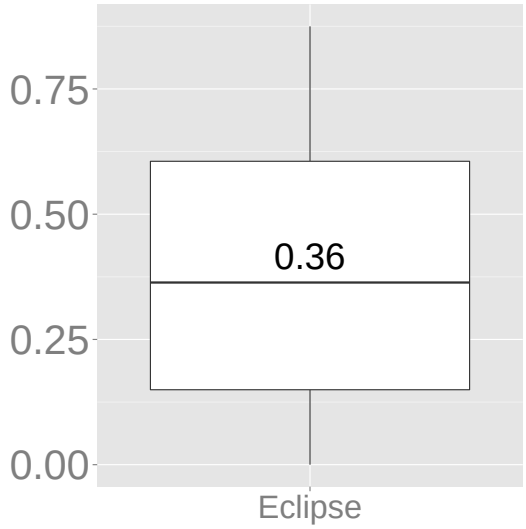


Figure 18 – The addressing stage measure for the “code freeze” period of the release cycle.

In order to understand the “code freeze” period in our studied systems, we contacted the project leaders of each studied system. We received a reply from the project leader of Eclipse, who provided us with the code freeze window of the Eclipse Luna 4.4 release cycle. The “code freeze” occurred between June 4th, 2014 and June 25th, 2014 (21 days).

Figure 18 shows the AS measure by using the “code freeze” start date rather than the official release date, *i.e.*, 21 days before the official release date. Indeed, we can observe a decrease in central tendency of the AS measure from 0.45 to 0.36 (median). However, we can observe that the majority of the data is between an AS value of 0.25 to 0.50 which is still a considerable distance in time before the “code freeze” period. In fact, the delayed addressed issues were addressed 43 days in the median before the “code freeze” stage. In addition, we find that 17% of the delayed addressed issues were addressed after the “code freeze” had started. The results indicate that even when considering the “code freeze” stage, the delayed addressed issues are unlikely to be delayed solely because they were addressed too close of a “code freeze” stage of the release cycle.

3.6 Threats to Validity

3.6.1 Construct Validity

A number of tools were developed in order 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 correct results.

3.6.2 Internal Validity

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

The main threat in this regard is the representativeness of the data. Although Firefox and Eclipse report the list of addressed issues in their release notes, we do not know how complete this list truly is. In addition, issues may be incorrectly listed in a release note. For example, an issue that should have been listed in the release note for version 2.0 but only appears in the release note for version 3.0. Such human specification error may produce noise in our datasets.

Another threat is the method that we use to map the addressed issues to releases in ArgoUML. This mapping is based on the *target_milestone* which may be more susceptible to human error. Nonetheless, our Firefox and Eclipse results are based on addressed issues that have been denoted in the release notes — and that we are more confident about their integration time.

In addition, the way that we segment the response variable of our explanatory models is also subjected to bias. For the release delay dimension (RQ1-RQ3), we segment the response variable into *next*, *after-1*, *after-2*, and *after-3-or-more*. Although we found it to be a reasonable classification, a different classification may yield different results. Also, we use the median to split the response variable studied in the abnormal delay dimension (RQ4-RQ5) into two categories. A different threshold to split the response variable may yield different results. However, we use the median as a threshold as it is resilient to outliers.

Finally, the attributes that we considered in our explanatory models are not exhaustive. The addition of other attributes would likely improve model performance. Nonetheless, our models performed well compared to random guessing and Zero-R models with the current set of attributes and dependent variable segmentation.

3.6.3 External Validity

External threats are concerned with our ability to generalize our results. In our work, we investigated three open source projects. Although the projects that we considered in our study are of different sizes and domains, and prescribing to different release policies, our findings may not generalize to other systems. Replication of this work in a large set of systems is required in order to arrive at more general conclusions.

3.7 Related Work

Since in this chapter we study the integration delay of addressed issues, we outline related work about the integration phase of software development and possible delays caused by software issues.

Jiang *et al.* (JIANG; ADAMS; GERMAN, 2013) studied attributes that could determine the acceptance and integration of a patch into the Linux kernel. A patch is a record of changes that is applied to a software system to address an issue. To identify such attributes, the authors built decision tree models and conducted top node analysis. Among the attributes studied, developer experience, patch maturity, and prior subsystem are found to play a major role in patch acceptance and integration time. Choetkiertikul *et al.* (MORAKOT et al., 2015a; MORAKOT et al., 2015b) study the risk of issues introducing delays that can postpone the shipment of new releases of a software project. The authors use local attributes (*i.e.*, attributes that can be collected in the issue report itself) and network attributes (*i.e.*, attributes that are extracted from the relationship between issues) to perform their analyses.

Similar to Jiang *et al.* (JIANG; ADAMS; GERMAN, 2013), we also investigate the integration of addressed issues. However, we focus on the integration delay of issues that have been addressed and not if a patch is more likely to be accepted than others. Differently from Choetkiertikul *et al.* (MORAKOT et al., 2015a; MORAKOT et al., 2015b), we study the attributes that may cause addressed issues to be delayed rather than the risk of an upcoming release to be postponed.

3.8 Conclusion

Once an issue is addressed, what users and code contributors most care about is when the software is going to reflect the addressed issue, *i.e.*, when the integration occurs. However, we observed that the integration of several addressed issues was delayed for several releases. In this context, it is not clear why certain addressed issues take longer to be integrated than others. Hence, we performed an empirical study of 20,995 issues from the ArgoUML, Eclipse and Firefox projects. In our study, we:

- found that despite being addressed well before of an upcoming release, 34% to 60% of the addressed issues were delayed by more than one release in ArgoUML and Eclipse. Furthermore, 98% of Firefox issues were delayed by at least one release.
- built random forest models to explain the integration delay of an addressed issue. Our models achieved a median ROC area of at least 0.78. Our models outperform baseline random and Zero-R models.

- computed the variable importance and the Scott-Knott tests to understand what attributes are the most important in our random forest models of integration delay. The workload of integrators is the most influential attribute in our models of release integration delay.
- found that, surprisingly, *priority* and *severity* have little impact on our models. Indeed, 36% to 97% of priority P1 addressed issues were delayed by at least one release.
- verified that our models of abnormal integration delay can outperform random guessing achieving ROC areas of at least 0.87 (median).
- found that the moment when an issue is addressed during the release cycle (queue position) and the workload of integrators are the most influential attributes for abnormal integration delay.

Our work provides some initial insights as to why some addressed issues are integrated prior to others. Our results suggest that characteristics of the release cycle are the ones that mostly impact on abnormal and release integration delay. Therefore, we believe that our findings highlight the importance of research and tools that support integrators of software projects. It is important to improve the integration stage of a release cycle, since the availability of an addressed issue in a release is what users and contributors care most about.

4 Impact of rapid release cycle on integration delay

An earlier version of the work in this chapter appears in the International Conference on Mining Software Repositories (MSR) (COSTA et al., 2014)

Key question: *What impact has the release cycle choice on the delivery delay?*

4.1 Introduction

Prior research investigated the impact of adopting rapid releases (MÄNTYLÄ et al., 2014; SOUZA; CHAVEZ; BITTENCOURT, 2014; SOUZA et al., 2015; BAYSAL; DAVIS; GODFREY, 2011; KHOMH et al., 2012). For example, Khomh *et al.* (KHOMH et al., 2012) found that bugs related to crash reports tend to be fixed faster in rapid Firefox releases than traditional Firefox releases. Mäntylä *et al.* (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, 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 new content. In Chapter 3, 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 rapid releases may not be delivering content as quickly as its proponents purport.

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). We use the term *issue* to refer to bug-fixes, enhancements, and new features (ANTONIOL et al., 2008). We set out to comparatively study the integration delay of addressed issues in the traditional and rapid releases of the Firefox system. In particular we address the following research questions:

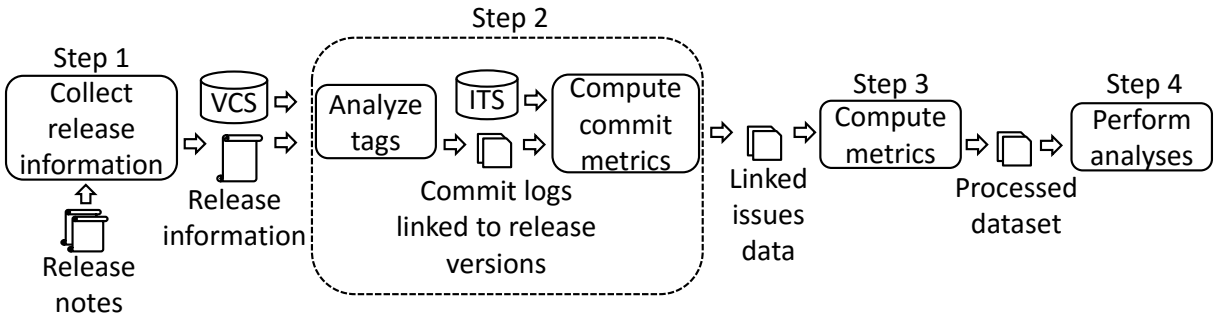


Figure 19 – Overview of the process to construct the dataset that is used in our analyses.

- ***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: How can traditional releases integrate addressed issues more quickly?*** We find that minor-traditional releases are one of the reasons 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 (median of 40 and 42 days for traditional and rapid releases, respectively).
- ***RQ3: Has the change in the release strategy caused a change in 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 addressed early in the release cycle of the current release are less likely to be delayed.

This chapter is organized as follows: in. In Section 4.2, we explain how we set up our empirical study. In Section 4.3, we present the results of our empirical study, while we discuss additional analyses in Section 4.4. Section 4.5 discloses the threats to the validity of our analyses. Section 4.6 outlines the related work. Finally, Section 4.7 draws conclusions.

4.2 Empirical Study

Figure 19 provides an overview of the steps of our study. Each step of the process is described below.

Table 6 – Traditional and rapid releases that are investigated in our study.

Strategy	Version range	Time period	#Major	#Minor
Trad.	1.0 - 4.0	Sep/2004 - Mar/2012	7	104
Rapid	1 - 27	Jun/2011 - Sep/2014	23	50

Step 1: Collect release information

We manually collect the date and version number of each Firefox release (minor and major releases of each release strategy) using the Firefox release history wiki.[1] Table 6 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 *Version Control System* (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.[2,3] 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 ($2,559$ were issues 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 were issues discarded).
3. Any potential ID that is the name of a file, *e.g.*, "159334.js" (607 were issues discarded).

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 nearest user-intended release after 3.7a1. In case

[1] https://en.wikipedia.org/wiki/Firefox_release_history
[2] http://cvsbook.red-bean.com/cvsbook.html
[3] https://mercurial.selenic.com/

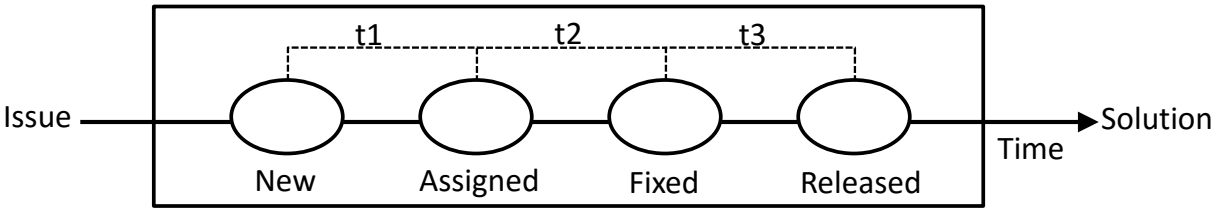


Figure 20 – The basic life-cycle of an issue.

a particular issue is found in the commit log of more than one release, we consider that particular issue to pertain to the earliest release, since it is the first one to contain the fix. 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 behave as *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).[4]

Steps 3 and 4: Compute metrics and perform analyses

We use the data from Step 2 to compute metrics that we use in our analyses. We select these metrics because we suspect that they share a relationship with integration delay. These metrics are described in greater detail in Section 4.3.

4.3 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?

RQ1: Motivation

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

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

RQ1: Approach

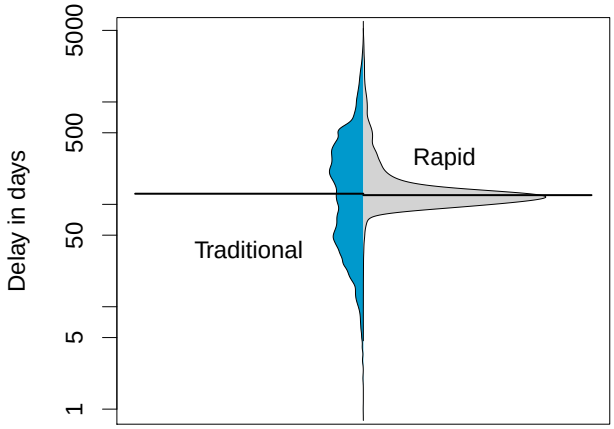
Figure 20 shows the basic life-cycle of an issue, which includes the triaging phase ($t1$), the fixing phase ($t2$), and the integration phase ($t3$). The *lifetime* of an issue is composed of all three phases (from *new* to *released*). We first observe the lifetime of the issues of traditional and 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 (KAMPSTRA et al., 2008) to compare the distributions of our data. A beanplot is a plot that describes data using one or multiple beans (see Figure 21a). The higher the amount of data with a particular value, the thicker the bean at that particular value in the y axis. We also use statistical tests such as Mann-Whitney-Wilcoxon (MWW) tests (WILKS, 2011) followed by Cliff's delta effect-size tests (CLIFF, 1993). MWW tests are nonparametric tests of the *null hypothesis* that two distributions come from the same population. In case we obtain a p value under 0.05 using a MWW test, we consider that the two distributions come from different populations (*i.e.*, we reject the null hypothesis). On the other hand, Cliff's delta is a non-parametric effect-size test 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. A positive Cliff's delta indicates how much larger are the values of the first distribution, while a negative Cliff's delta indicates the inverse. Such test is important to verify how significant is the difference between distributions even if they come from different populations. 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 over its median.

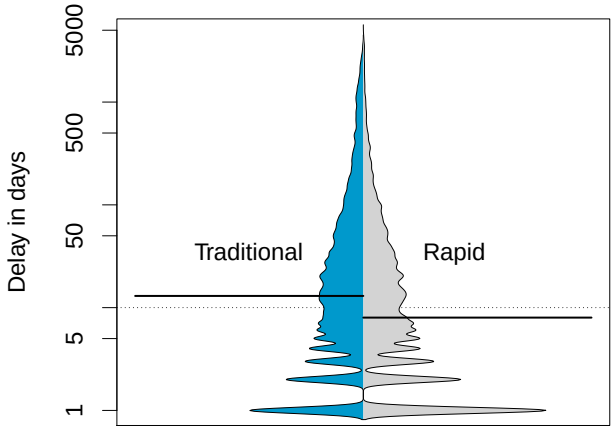
RQ1: Results

There is no significant difference between traditional and rapid releases regarding issue lifetime. Figure 21a shows the distributions of the lifetime of the issues in traditional and rapid releases. We observe a significant $p < 0.05$ but a $negligible$ ($delta = -0.07$) difference between distributions. We also observe that traditional releases have a greater MAD over the distribution (141 days) compared to rapid releases (30 days). Our results indicate that the difference in the issues' lifetime between traditional and rapid releases is not obvious as one would speculate. However, rapid releases deliver addressed issues more consistently. We then look at the triaging, fixing, and integration time spans to better understand the differences between traditional and rapid releases.

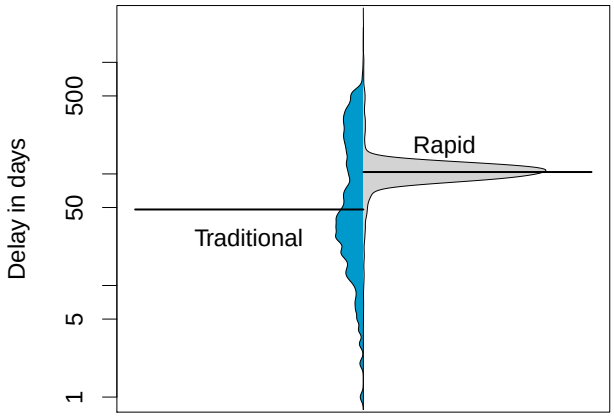
Addressed issues are triaged and fixed faster in rapid releases, but tend to wait for a longer time before being released. Figures 21b and 21c show the fixing and integra-



(a) Lifetime



(b) Fixing phase



(c) Integration phase

Figure 21 – Time spans of the phases involved in the lifetime of an issue.

tion time spans, respectively. We observe that addressed issues take 47 days on average (median) to be integrated into traditional releases, while taking 104 days (median) to be integrated into rapid releases ($p < 0.05$ with a $small$ effect-size of $delta = -0.30$).

Regarding fixing time span, an issue takes 6 days (median) to be fixed in rapid releases, while it takes 9 days (median) in traditional releases. For these results, we obtain a significant $p < 0.05$ but a $negligible$ ($delta = 0.13$) difference between distribu-

tions. Our results complement previous research. Khomh *et al.* (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 $p < 0.05$ but a $negligible$ ($delta = 0.03$) difference between traditional and rapid releases regarding triaging time. The median triaging time for traditional and rapid releases are 3 and 2 days, respectively.

When we consider both prior-integration phases together (triaging $t1$ plus fixing $t2$ in Figure 20), we observe that an issue takes 11 days (median) to the point that it gets addressed in rapid releases, while it takes 19 days (median) in traditional releases ($p < 0.05$ 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 have more consistency in fixing and integrating issues. Rapid releases achieve MADs of 9 and 18 days for fixing and integration, respectively. The values for traditional releases are 13 and 53 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 the rapid releases than the traditional ones.

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

RQ2: Motivation

In RQ1, we find that traditional releases integrate addressed issues more quickly on average (median) compared to 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 on average than a rapid release strategy?

RQ2: Approach

We group traditional and rapid releases into major and minor releases and study their integration delay. Similar to RQ1, we use beanplots (KAMPSTRA et al., 2008), Mann-Whitney-Wilcoxon tests (WILKS, 2011), Cliff's delta tests (CLIFF, 1993), and MADs (HOWELL, 2005; LEYS et al., 2013) to analyze the data.

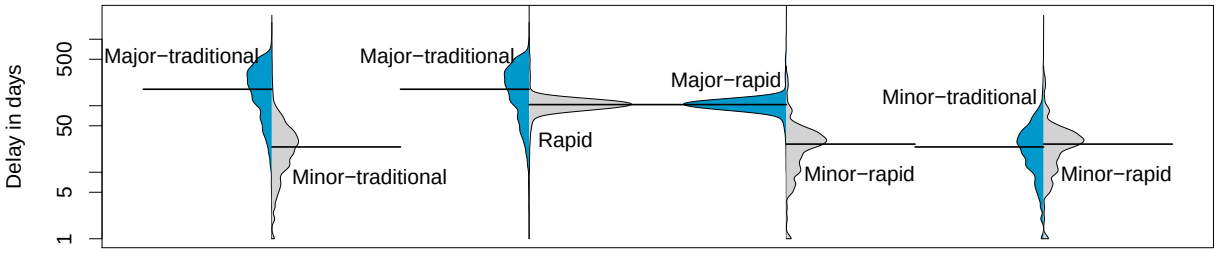


Figure 22 – Distributions of integration delay of addressed issues grouped by minor and major releases.

RQ2: Results

Minor-traditional releases tend to have less integration delay than major/-minor-rapid releases. Figure 22 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 *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 ($p < 0.05$ with a $medium$ effect-size, *i.e.*, $delta = 0.37$).

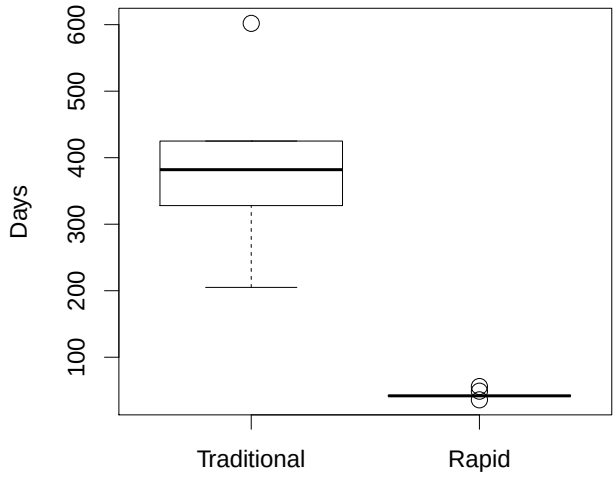
The Firefox rapid release cycle includes ESR releases (*cf.* Section 2.5) and a few minor stabilization and security releases. These releases also integrate addressed issues more quickly than major-rapid releases (*major-rapid vs. minor-rapid*) with a $p < 0.05$ and a $large$ effect-size, *i.e.*, $delta = 0.92$.

Although we do not observe a statistically significant difference between distributions in the comparison of *minor-traditional vs. minor-rapid* ($p > 0.05$), it is interesting to note how minor-traditional releases tend to have shorter integration delay compared to minor-rapid releases (23 and 26, respectively).

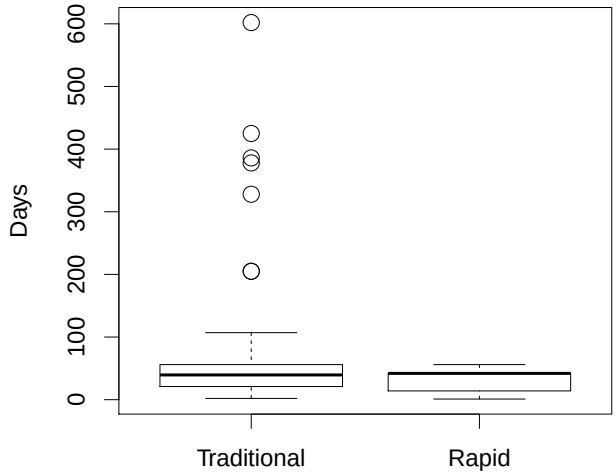
Minor releases possibly have the fastest integration delay because they are more focused on a particular set of issues that, once addressed, should be released immediately. For example, the release history documentation of Firefox shows that minor releases are usually related to stability and security issues.[5]

When considering both minor and major releases, the time span between traditional and rapid releases are roughly the same on average. 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 23a shows that, at first glance, one may speculate that rapid releases should deliver addressed issues faster because releases are produced more frequently. However, if we consider

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



(a) Only Major



(b) Major and Minor

Figure 23 – Release frequency (in days).

both major and minor releases — as shown in Figure 23b — 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 traditional releases can integrate addressed issues more quickly than rapid releases. Furthermore, the length of the release cycles are roughly the same between traditional and rapid releases when considering both minor and major releases.

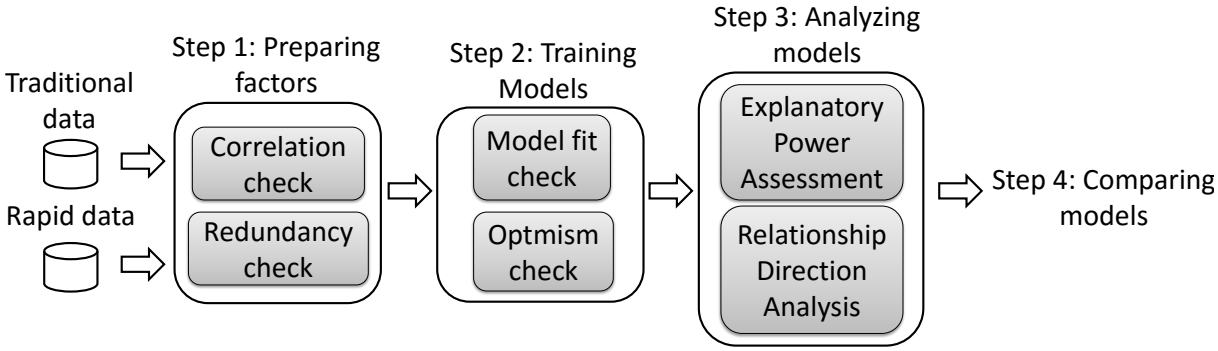


Figure 24 – Overview of the process that we use to build our explanatory models.

RQ3: Has the change in the release strategy caused a change in the characteristics of delayed issues?

RQ3: Motivation

To better understand the differences between traditional and rapid releases regarding integration delay, it is also important to study if the change in the release strategy had an impact on the strength of the relationship between metrics and integration delay.

RQ3: Approach

We build explanatory models (*i.e.*, logistic regression models) for the traditional and rapid releases data using the metrics that are presented in Tables 7 , 8, and 9. We model our response variable Y as $Y = 1$ for addressed issues that are delayed, *i.e.*, missed at least one release before integration (Chapter 3) and $Y = 0$ otherwise.

We follow the guidelines of Harrell Jr. (HARRELL, 2001) for building explanatory models. Figure 24 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 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 assigned

Table 7 – Metrics used in our explanatory models (Reporter, Resolver, and Issue dimensions).

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 (SHIHAB et al., 2010).
	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 (SHIHAB et al., 2010).
	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 and integrated of the addressed issue (SCHROTER; BETTENBURG; PREMRAJ, 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 addressed in the Eclipse project (PANJER, 2007).
	Priority	Nominal	**d:** The priority level of the issue report. Issues with higher severity levels (*e.g.*, blocking) 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.

by our explanatory models.

Next, we subtract the bootstrap-calculated *optimism* from the initial ROC area and Brier score estimates (EFRON, 1986). This *optimism* is a robust measure to evaluate overestimation of the fit of explanatory models.

Finally, we evaluate the impact that each metric has on the models that we

Table 8 – Metrics used in our explanatory models (Project dimension).

Dimension	Attributes	Value	Definition (d)|Rationale (r)
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.

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 response variable (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 (FE,).

We also plot nomograms (IASONOS et al., 2008; FE,) to evaluate the impact of the metrics on our models. Nomograms are used as a user-friendly chart to visually interpret explanatory models. For instance, Figure 29 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 higher the effect of that metric in the explanatory model. We compare which metrics are more important in both traditional and rapid releases, to better understand the differences between these release strategies.

Table 9 – Metrics used in our explanatory models (Process dimension).

Dimension	Attributes	Value	Definition (d)|Rationale (r)
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 (JIANG; ADAMS; GERMAN, 2013).
	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 (JIANG; ADAMS; GERMAN, 2013; NAGAPPAN; BALL, 2005).
	Fix time	Numeric	**d:** Number of days between the creation of the issue report and the date that it was addressed (GIGER; PINZGER; GALL, 2010). **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 addressed the issue, which may impact the integration of the issue into a release (JIANG; ADAMS; GERMAN, 2013).
	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 (GIGER; PINZGER; GALL, 2010), which might impact the integration delay (JIANG; ADAMS; GERMAN, 2013).
	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 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; KIM; ZIMMERMANN, 2009; JIANG; ADAMS; GERMAN, 2013).

RQ3: Results

Our models achieve a Brier score of 0.05-0.16 and ROC areas of 0.80-0.84. The models that we fit to traditional releases achieve a Brier score of 0.16 and an ROC area of 0.84, while the models that we fit to the rapid release data achieve a Brier score of 0.05 and an ROC area of 0.80. Moreover the bootstrap-calculated optmism is below 0.0044 for both ROC and Brier score in our models. These results show that: (i) our models outperform naïve approaches, such as random guessing and (ii) that our models are stable enough to perform our statistical inferences that follows.

Table 10 – Overview of the regression results. 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		$5,367$	$2,711$
Budgeted Degrees of Freedom		168	158
Degrees of Freedom Spent		26	25
		Overall	Overall
Reporter experience	D.F.	1	1
	χ^2	2.4^{***}	1.5^{***}
Reporter integration	D.F.	1	1
	χ^2	2.2^{***}	4.0^{***}
Resolver Experience	D.F.	1	$\oslash$
	χ^2	≈ 0	
Resolver integration	D.F.	1	1
	χ^2	5.5^{***}	5.3^{***}
Issue addressing time	D.F.	1	1
	χ^2	2^{***}	8.9^{***}
Severity	D.F.	6	6
	χ^2	0.4^{**}	1.2^{***}
Priority	D.F.	5	5
	χ^2	0.7^{***}	0.2
Size of description	D.F.	1	1
	χ^2	≈ 0	0.8^{***}
Stack trace attached	D.F.	1	1
	χ^2	≈ 0	≈ 0
Number of files	D.F.	1	1
	χ^2	≈ 0	0.5^{**}
Number of comments	D.F.	1	1
	χ^2	≈ 0	29.9^{***}
Number of tossing	D.F.	1	1
	χ^2	0.2^{**}	0.1
Number of activities	D.F.	1	1
	χ^2	5.3^{***}	4.4^{***}
Interval of comments	D.F.	$\oslash$	$\oslash$
	χ^2		
Code churn	D.F.	1	1
	χ^2	0.2^{**}	0.2^{*}
Queue position	D.F.	1	1
	χ^2	14.9^{***}	1.6^{***}
Queue rank	D.F.	1	1
	χ^2	54.8^{***}	12.8^{***}
Cycle queue rank	D.F.	1	1
	χ^2	11.4^{***}	28.5^{***}
Cycle queue position	D.F.	$\oplus$	$\oslash$
	χ^2		

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

Traditional releases prioritize the integration of backlog issues, while rapid releases prioritize the integration of issues of the current release cycle. Table 10 shows

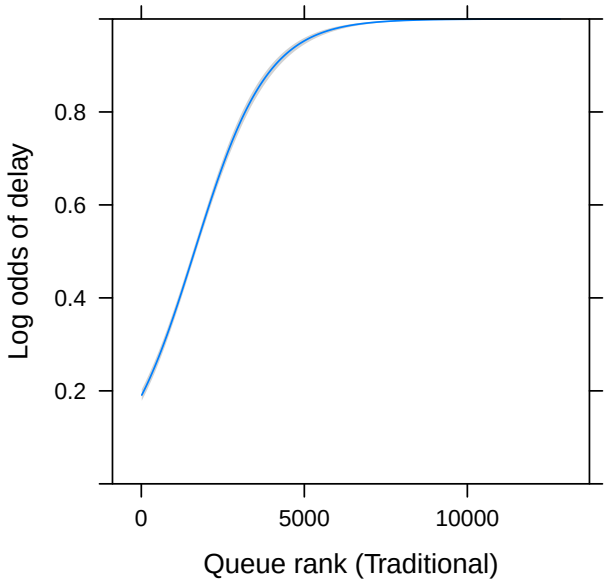


Figure 25 – The later an issue is addressed in the project backlog (higher ranks) the higher the log odds of an addressed issue being delayed. The blue line stands for the values of our model fit, whereas the grey area shows the 95% confidence interval based on models fit to 1,000 bootstrap samples

the metrics that we use in our regression models, the explanatory power (χ^2) of each metric, and which metrics were discarded during the *correlation* and *redundancy* analysis.

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 (*cf.* Table 8). Figure 25 shows the relationship that queue rank shares with integration delay. Our models reveal that the addressed issues in traditional releases have a higher log odds of being delayed (y axis) if they are addressed more recently 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. Figure 26 shows the relationship that cycle queue rank shares with integration delay. Our models reveal that the addressed issues in rapid releases have a higher odds of being delayed (y axis) 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 *number of comments*. Figure 27 shows the relationship that *number of comments* shares with integration delay. We observe that the greater the number of comments of an addressed issue, the greater the log odds of integration delay. This result corroborates the intuition that an intense discussion may indicate a complex issue, which may be more likely to be delayed.

Moreover, Figures 28 and 29 shows the estimated effect of our metrics using

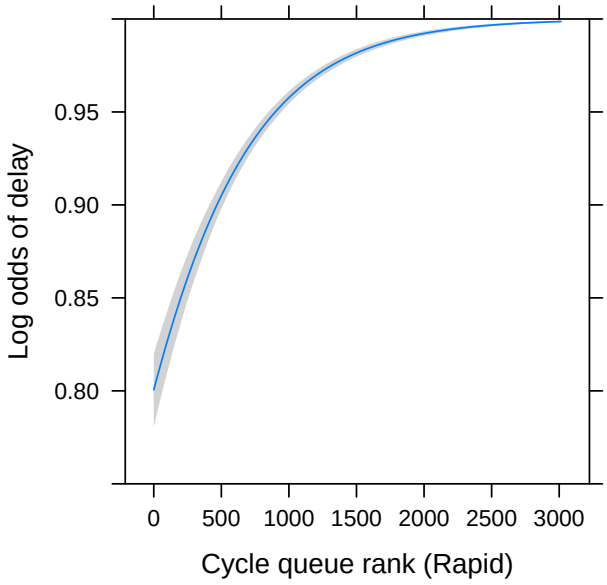


Figure 26 – The relationship between integration delay and the *cycle queue rank* metric. The blue line stands for the values of our model fit, whereas the grey area shows the 95% confidence interval based on models fit to 1,000 bootstrap samples

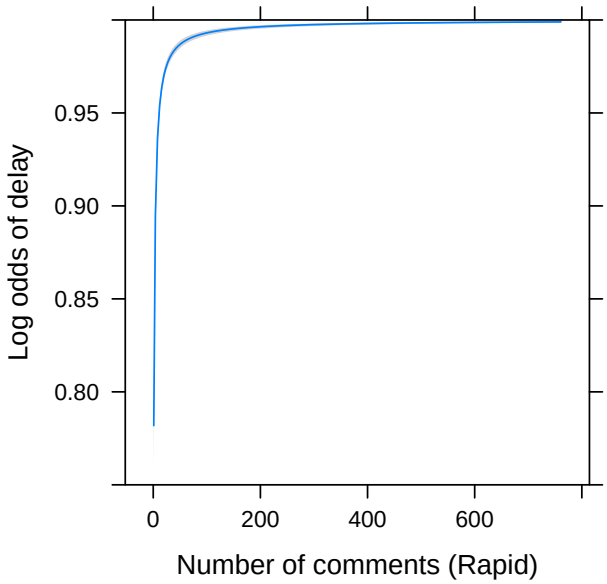


Figure 27 – The relationship between integration delay and the *number of comments* metric. The blue line stands for the values of our model fit, whereas the grey area shows the 95% confidence interval based on models fit to 1,000 bootstrap samples

nomograms (IASONOS et al., 2008). Indeed, our nomograms reiterate the high impact of *number of comments* (100 points) and *cycle queue rank* (89 points) in rapid releases, and the high impact of *queue rank* (100 points) in traditional releases. We also observe that *stacktrace attached* seems to have a large impact on traditional releases (65 points) despite not being a significant contributor to the fit of our models (*cf.* Table 10). The large impact shown in our nomogram for *stacktrace attached* is due to the skewness

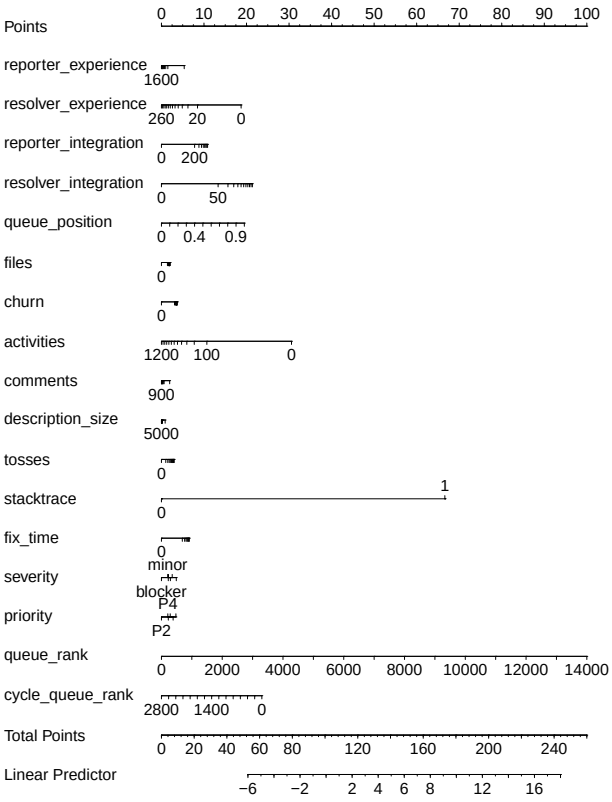


Figure 28 – Nomograms of our explanatory models for traditional releases.

of our data, *i.e.*, only 5 instances within the traditional release data have the *stacktrace attached* set to true. Thus, we cannot expect that *stacktrace attached* strongly contributes to the overall fit of our models.

Our results suggest that another *main* 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 less likely it is to be delayed. 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 less likely it is to be delayed.

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 are addressed early in the release cycle of the current release are less likely to be delayed.

4.4 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

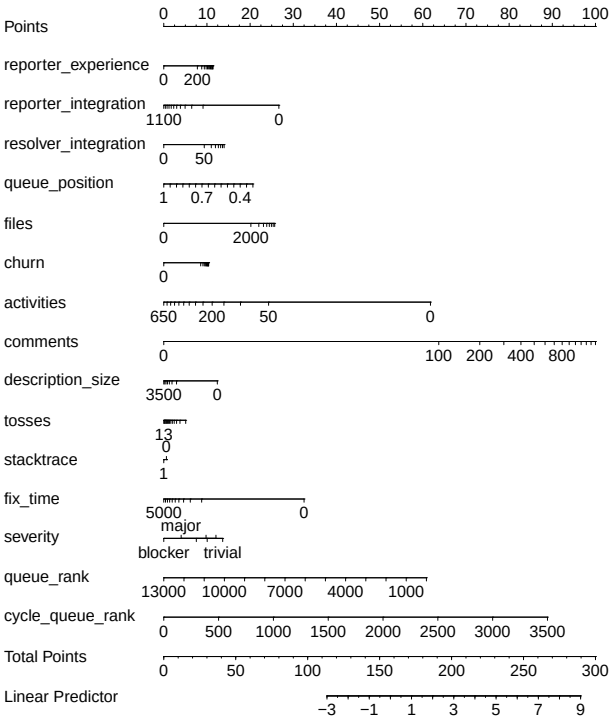
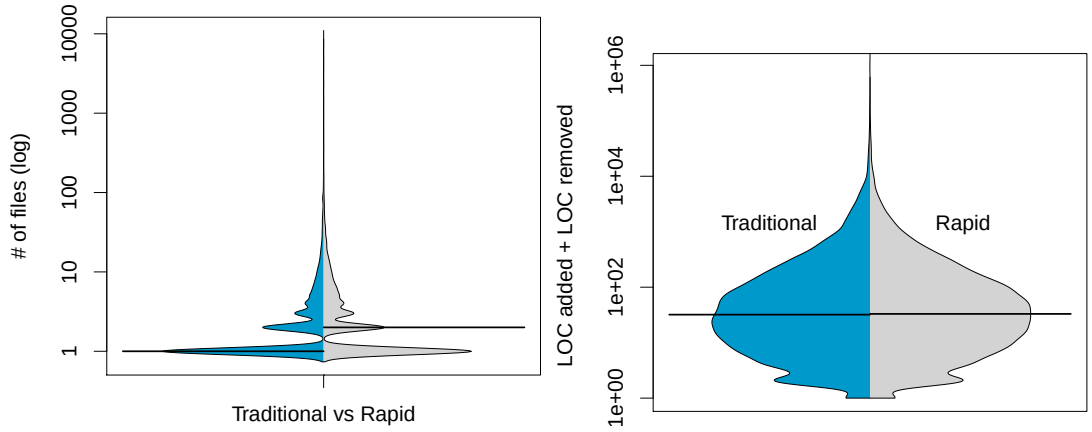


Figure 29 – Nomograms of our explanatory models for rapid releases.

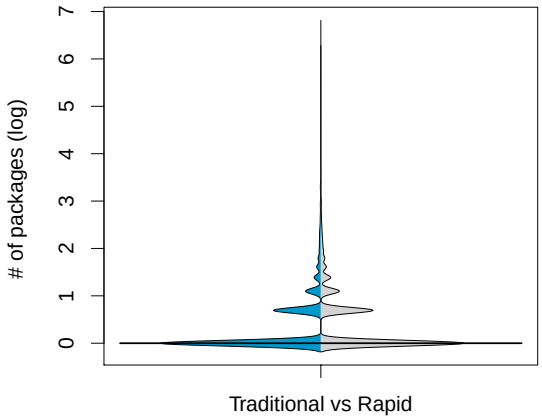
addressed issues.

The integration delay of addressed issues is unlikely to be related with 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* (sum of code additions and code deletions), and *number of packages* that were involved in the fix of an issue to approximately measure the *size of the issue*. Figure 30 shows the distributions of the metrics that measure the *size of the issues*. We observe that there is no statistically significant difference between distributions of *LOC* ($p > 0.05$). As for *number of files* and *number of packages*, although we obtain a significant $p < 0.05$, we respectively obtain $negligible$ effect-sizes of $delta = -0.05$ and $delta = -0.07$ between distributions.

The difference between traditional and rapid releases is unlikely to be related with enhancements and bug-fixes. We also investigate if the observed difference in the integration delay between traditional and rapid releases is related with the kind of addressed issue. For example, rapid releases could be delivering more enhancements, which could be associated with a greater integration delay. Figure 31 shows the distributions of delays among release strategies grouped by bugs and enhancements. We observe no clear distinction between integration delay and the kind of addressed issues being integrated.



(a) Files (b) LOC



(c) Packages

Figure 30 – Size of the addressed issues in the traditional and rapid release data.

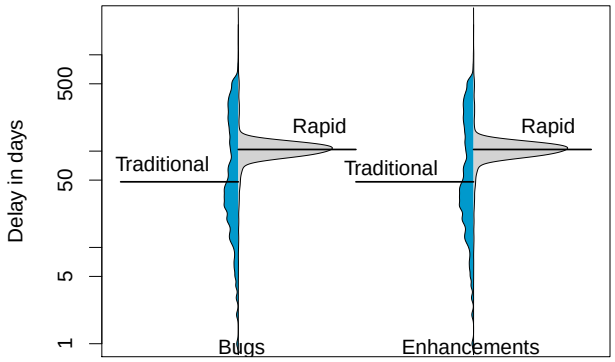


Figure 31 – We group the addressed issues into “bugs” and “enhancements” by using the *severity* field. However, the difference in the integration delay between release strategies is unlikely to be related with the kind of the issue.

4.5 Threats to the Validity

In this section, we describe the threats to the validity of our study.

4.5.1 Construct Validity

A number of 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.

4.5.2 Internal Validity

The 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 the releases may not represent the total addressed issues per each release. For example, although Firefox developers record the issue ID in the commit logs, we do not know how many of the addressed issues were recorded in the VCS systems. In addition, the rate of issue IDs that were recorded in the VCS could have changed overtime. Nevertheless, we use the same approach to compare both traditional and rapid release data, and focus on relative values, not absolute ones.

In Section 4.4, 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 (HERRAIZ et al., 2008) (*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).

4.5.3 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.

4.6 Related Work

In this chapter, we compare rapid releases and traditional releases in the Firefox studied system with respect to the integration delay of addressed issues. In this section, we situate our work with respect to prior research on the impact of adopting rapid releases and the process of integrating and delivering addressed issues.

4.6.1 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.* (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.* (SOUZA; CHAVEZ; BITTENCOURT, 2014) found that the number of reopened bugs increased by 7% when Firefox changed to a rapid release cycle. Souza *et al.* (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 Firefox traditional releases.

It is not clear yet if rapid releases lead to faster fixing of bugs. Baysal *et al.* (BAYSAL; DAVIS; GODFREY, 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.* (KHOMH et al., 2012) found that bugs are fixed faster in Firefox rapid releases when compared to its traditional releases. However, less bugs are fixed in rapid releases, proportionally.

Rapid releases may cause users to adopt new versions of the software earlier. Baysal *et al.* (BAYSAL; DAVIS; GODFREY, 2011) 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.* (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.

Inspired by past work on the differences between rapid and traditional release cycles, we set out to study the impact that the shift of release strategies has had on integration delay.

4.6.2 Delays and Software Issues

Prior research has studied delays related to the integration and delivery of addressed issues to end users. Jiang *et al.* (JIANG; ADAMS; GERMAN, 2013) studied the integration process of the Linux kernel. In Chapter 3 we found that although issues are addressed well before an upcoming release they may still be delayed. Indeed, 98% of addressed issues in the Firefox system were delayed by at least one release. Choetkiertikul *et al.* (MORAKOT et al., 2015a; MORAKOT et al., 2015b) study the risk of issues introducing delays to deliver new releases of a software project.

The integration of addressed issues is costly. Rahman and Rigby (RAHMAN; RIGBY, 2015) 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.* (JIANG; ADAMS, 2014) 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.

4.7 Conclusions

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

- Although rapid releases address issues faster, traditional releases integrate addressed issues more quickly on average (median).
- The time to triage issues is not significantly different between traditional and rapid releases.
- The total lifetime of issues is not significantly different between traditional and rapid releases.
- Minor-traditional releases is one of the reasons why traditional releases can integrate addressed issues more quickly than rapid releases.
- In traditional releases, addressed issues are more likely to be delayed if they are addressed recently in the backlog. On the other hand, in rapid releases, addressed issues are more likely to be delayed if they are addressed recently in the current release cycle.
- The difference in the integration delay between traditional and rapid releases is unlikely to be related to the size and kind of addressed issues.

Our results suggest that the differences between traditional and rapid release strategies are not *mainly* in the length of time between official releases (considering both major and minor) nor in the time to deliver addressed issues as one may speculate. Instead, we observe that the *main* difference is in the planning of how issues are delivered. Rapid releases are more consistent when delivering addressed issues. Organizations that plan to choose a rapid release strategy over a traditional release strategy should not rely on the faster delivering of addressed issues as a main motivator.

5 Staged Delivery

Key question: *How much time is needed to stabilize addressed issues?*

6 Conclusions

Issue tracking systems has long been used to manage bug-fixes, enhancements or new features (*i.e.*, issues). For a software project to achieve a sustained success, it has to keep including new exciting features, fixing bugs and improving the existent functionality. In addition, failing to address issues, may cause a software project to lose its credibility before its users.

On the other hand, addressed issues may suffer undesirable delay before being released (*e.g.*, delivery delay). In this thesis, we empirically study the delays that are involved prior the release of addressed issues to end users. In the remainder of this chapter, we outline the contributions of this thesis and disclose promising venues for future work.

6.1 Contributions and Findings

> ***Thesis Statement:*** *Even though issues are addressed, they still suffer a considerable delivery delay that development teams need to manage. Historical data recorded in software repositories can be used to understand and estimate delivery delay.*

The overarching goal of this thesis is to understand why addressed issues may suffer delivery delay after being addressed. We leverage data that is recorded in version control systems and issue tracking systems to perform our studies. In general, we find that:

- 34 to 98% of addressed issues are delayed by at least one release before being shipped to users. Additionaly, the *workload of integrators* and the *timing* when an issue is addressed within a release cycle are the most important metrics to explain the (abnormal) integration delay of addressed issues (Chapter 3).
- The priority and severity fields of addressed issues have little impact to explain integration delay. Moreover, the component of an issue do not share a relationship with integration delay (Chapter 3).
- Even though rapid releases address issues faster, traditional releases integrate addressed issues more quickly on average (median). In addition, we observe no significant difference between traditional and rapid releases in the speed to deliver addressed issues, *i.e.*, from the issue creation until shipment (Chapter 4).

- Minor-traditional releases can explain why a traditional release cycle can integrate addressed issues faster than rapid releases (Chapter 4).
- Metrics related to release cycle timing can accurately explain integration delay of addressed issues in traditional and rapid releases. Nevertheless, issues in traditional releases are queued up – issues addressed earlier in the project backlog are less likely to be delayed. On the other hand, issues in rapid releases are queued on a per release basis, in which issues addressed early the current release cycle are less likely to be delayed (Chapter 4).

Our main findings suggest that the delivery delay that is incurred by addressed issues introduce a non-negligible bottleneck in the software development process. Organizations should invest on improving their integration process to mitigate such delivery delay if a more efficient delivering of addressed issues is desired. Our findings also suggest that shorter release cycles may not accomplish the faster delivery of addressed issues. Instead, shorter release cycles provide a more consistent delivery. Future research should focus on developing tools to improve the integration of addressed issues into shippable branches in other to improve the time to deliver addressed issues.

Bibliography

ANBALAGAN, P.; VOUK, M. On predicting the time taken to correct bug reports in open source projects. In: *Proceedings of the 2009 IEEE International Conference on Software Maintenance*. [S.l.: s.n.], 2009. (ICSM '09), p. 523–526. ISSN 1063-6773. Cited 4 times on pages 8, 14, 18, and 27.

ANTONIOL, G. et al. 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, October 27-30, 2008, Richmond Hill, Ontario, Canada*. [S.l.: s.n.], 2008. p. 23. Cited 3 times on pages 7, 12, and 50.

ANVIK, J.; HIEW, L.; MURPHY, G. C. Coping with an open bug repository. In: *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*. New York, NY, USA: ACM, 2005. (eclipse '05), p. 35–39. ISBN 1-59593-342-5. Disponível em: <http://doi.acm.org/10.1145/1117696.1117704>. Cited 2 times on pages 8 and 18.

ANVIK, J.; HIEW, L.; MURPHY, G. C. Who should fix this bug? In: *Proceedings of the 28th International Conference on Software Engineering*. [S.l.: s.n.], 2006. (ICSE '06), p. 361–370. ISBN 1-59593-375-1. Cited 3 times on pages 7, 12, and 13.

BAYSAL, O.; DAVIS, I.; GODFREY, M. W. A tale of two browsers. In: ACM. *Proceedings of the 8th Working Conference on Mining Software Repositories*. [S.l.], 2011. p. 238–241. Cited 2 times on pages 50 and 70.

BHATTACHARYA, P.; NEAMTIU, I. Bug-fix time prediction models: Can we do better? In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2011. (MSR '11), p. 207–210. ISBN 978-1-4503-0574-7. Disponível em: <http://doi.acm.org/10.1145/1985441.1985472>. Cited on page 14.

BREIMAN, L. Random forests. In: *Machine Learning*. [S.l.: s.n.], 2001. (Springer Journal no. 10994), p. 5–32. Cited on page 30.

CLIFF, N. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin*, American Psychological Association, v. 114, n. 3, p. 494, 1993. Cited 2 times on pages 54 and 56.

COSTA, D. A. d. et al. An empirical study of delays in the integration of addressed issues. In: IEEE. *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. [S.l.], 2014. p. 281–290. Cited 2 times on pages 18 and 50.

DELONE, W. H.; MCLEAN, E. R. The delone and mclean model of information systems success: a ten-year update. *Journal of management information systems*, Taylor & Francis, v. 19, n. 4, p. 9–30, 2003. Cited on page 7.

EFRON, B. How biased is the apparent error rate of a prediction rule? *Journal of the American Statistical Association*, Taylor & Francis, v. 81, n. 394, p. 461–470, 1986. Cited on page 60.

FE, H. J. *rms: Regression Modeling Strategies, http://biostat.mc.vanderbilt.edu/rms*. Accessed: 22-01-2015. Cited on page 61.

GIGER, E.; PINZGER, M.; GALL, H. Predicting the fix time of bugs. In: *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. New York, NY, USA: ACM, 2010. (RSSE '10), p. 52–56. Cited 6 times on pages 8, 14, 18, 27, 29, and 62.

GUO, P. J. et al. Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. New York, NY, USA: ACM, 2010. (ICSE '10), p. 495–504. ISBN 978-1-60558-719-6. Disponível em: <http://doi.acm.org/10.1145/1806799.1806871>. Cited on page 14.

HARRELL, F. E. *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. [S.l.]: Springer, 2001. Cited on page 59.

HERBSLEB, J. D.; MOCKUS, A. An empirical study of speed and communication in globally distributed software development. *Software Engineering, IEEE Transactions on*, IEEE, v. 29, n. 6, p. 481–494, 2003. Cited on page 7.

HERRAIZ, I. et al. Towards a simplification of the bug report form in eclipse. In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. New York, NY, USA: ACM, 2008. (MSR '08), p. 145–148. ISBN 978-1-60558-024-1. Disponível em: <http://doi.acm.org/10.1145/1370750.1370786>. Cited 4 times on pages 14, 30, 36, and 69.

HOOIMEIJER, P.; WEIMER, W. Modeling bug report quality. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2007. (ASE '07), p. 34–43. ISBN 978-1-59593-882-4. Disponível em: <http://doi.acm.org/10.1145/1321631.1321639>. Cited on page 13.

HOWELL, D. C. Median absolute deviation. *Encyclopedia of Statistics in Behavioral Science*, Wiley Online Library, 2005. Cited 2 times on pages 54 and 56.

IASONOS, A. et al. How to build and interpret a nomogram for cancer prognosis. *Journal of Clinical Oncology*, American Society of Clinical Oncology, v. 26, n. 8, p. 1364–1370, 2008. Cited 2 times on pages 61 and 65.

JEONG, G.; KIM, S.; ZIMMERMANN, T. Improving bug triage with bug tossing graphs. In: ACM. *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. [S.l.], 2009. p. 111–120. Cited 2 times on pages 29 and 62.

JIANG, Y.; ADAMS, B. How much does integrating this commit cost? - a position paper. *2nd International Workshop on Release Engineering (RELENG)*, 2014. Cited on page 71.

JIANG, Y.; ADAMS, B.; GERMAN, D. M. 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*. Piscataway, NJ, USA: IEEE Press, 2013. (MSR '13), p. 101–110. ISBN 978-1-4673-2936-1. Disponível em: <http://dl.acm.org/citation.cfm?id=2487085.2487111>. Cited 9 times on pages 8, 9, 15, 18, 29, 33, 48, 62, and 70.

KAMPSTRA, P. et al. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software*, v. 28, n. 1, p. 1–9, 2008. Cited 2 times on pages 54 and 56.

KHOMH, F. et al. Do faster releases improve software quality? an empirical case study of mozilla firefox. In: IEEE. *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. [S.l.], 2012. p. 179–188. Cited 3 times on pages 50, 56, and 70.

KIM, S.; WHITEHEAD JR., E. J. How long did it take to fix bugs? In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. New York, NY, USA: ACM, 2006. (MSR '06), p. 173–174. ISBN 1-59593-397-2. Disponível em: <http://doi.acm.org/10.1145/1137983.1138027>. Cited 4 times on pages 8, 14, 18, and 27.

LEYS, C. et al. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology*, Elsevier, v. 49, n. 4, p. 764–766, 2013. Cited 2 times on pages 54 and 56.

MÄNTYLÄ, M. V. et al. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, Springer, p. 1–42, 2014. Cited 2 times on pages 50 and 70.

MANTYLA, M. V. et al. On rapid releases and software testing. In: IEEE. *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. [S.l.], 2013. p. 20–29. Cited on page 13.

MARKS, L.; ZOU, Y.; HASSAN, A. E. Studying the fix-time for bugs in large open source projects. In: *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. New York, NY, USA: ACM, 2011. (Promise '11), p. 11:1–11:8. ISBN 978-1-4503-0709-3. Cited 4 times on pages 8, 14, 18, and 27.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. D. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, ACM, New York, NY, USA, v. 11, n. 3, p. 309–346, jul. 2002. ISSN 1049-331X. Disponível em: <http://doi.acm.org/10.1145/567793.567795>. Cited on page 36.

MORAKOT, C. et al. Characterization and prediction of issue-related risks in software projects. In: *12th IEEE/ACM Working Conference on Mining Software Repositories, MSR 2015, Florence, Italy, May 16-17, 2015*. [S.l.: s.n.], 2015. p. 280–291. Cited 4 times on pages 8, 15, 48, and 70.

MORAKOT, C. et al. Predicting delays in software projects using networked classification. In: *30th IEEE/ACM International Conference on Automated Software Engineering ASE, Lincoln, Nebraska, USA, November 9-13, 2015*. [S.l.: s.n.], 2015. Cited 4 times on pages 8, 15, 48, and 70.

NAGAPPAN, N.; BALL, T. Use of relative code churn measures to predict system defect density. In: IEEE. *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. [S.l.], 2005. p. 284–292. Cited 2 times on pages 29 and 62.

PANJER, L. D. Predicting eclipse bug lifetimes. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007. (MSR '07), p. 29–. ISBN 0-7695-2950-X. Disponível em: <http://dx.doi.org/10.1109/MSR.2007.25>. Cited 3 times on pages 14, 28, and 60.

RAHMAN, M. T.; RIGBY, P. C. Release stabilization on linux and chrome. *IEEE Software*, IEEE, n. 2, p. 81–88, 2015. Cited on page 70.

SAHA, R.; KHURSHID, S.; PERRY, D. An empirical study of long lived bugs. In: *2014 Software Evolution Week - IEEE Conference onSoftware Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*. [S.l.: s.n.], 2014. p. 144–153. Cited on page 14.

SCHROTER, A.; BETTENBURG, N.; PREMRAJ, R. Do stack traces help developers fix bugs? In: IEEE. *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. [S.l.], 2010. p. 118–121. Cited 2 times on pages 28 and 60.

SCOTT, A.; KNOTT, M. A cluster analysis method for grouping means in the analysis of variance. *Biometrics*, JSTOR, p. 507–512, 1974. Cited on page 33.

SHIHAB, E. et al. Predicting re-opened bugs: A case study on the eclipse project. In: IEEE. *Reverse Engineering (WCRE), 2010 17th Working Conference on*. [S.l.], 2010. p. 249–258. Cited on page 60.

SOUZA, R. et al. Rapid releases and patch backouts: A software analytics approach. *Software, IEEE*, IEEE, v. 32, n. 2, p. 89–96, 2015. Cited 2 times on pages 50 and 70.

SOUZA, R.; CHAVEZ, C.; BITTENCOURT, R. A. Do rapid releases affect bug reopening? a case study of firefox. In: IEEE. *Software Engineering (SBES), 2014 Brazilian Symposium on*. [S.l.], 2014. p. 31–40. Cited 2 times on pages 50 and 70.

SUBRAMANIAM, C.; SEN, R.; NELSON, M. L. Determinants of open source software project success: A longitudinal study. *Decision Support Systems*, Elsevier, v. 46, n. 2, p. 576–585, 2009. Cited on page 7.

WEIB, C. et al. How long will it take to fix this bug? In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. Washington, DC, USA: IEEE Computer Society, 2007. (MSR '07), p. 1–. ISBN 0-7695-2950-X. Cited 4 times on pages 8, 14, 18, and 27.

WILKS, D. S. *Statistical methods in the atmospheric sciences*. [S.l.]: Academic press, 2011. v. 100. Cited 2 times on pages 54 and 56.

ZHANG, F. et al. An empirical study on factors impacting bug fixing time. In: *Reverse Engineering (WCRE), 2012 19th Working Conference on*. [S.l.: s.n.], 2012. p. 225–234. ISSN 1095-1350. Cited 2 times on pages 14 and 15.

ZHANG, H.; GONG, L.; VERSTEEG, S. Predicting bug-fixing time: An empirical study of commercial software projects. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 1042–1051. ISBN 978-1-4673-3076-3. Cited 4 times on pages 8, 14, 18, and 27.