Daniel Alencar da Costa

Understanding the Delivery Delay of Addressed Issues in Large Software Projects

Natal, RN, Brazil

February, 2017

Daniel Alencar da Costa

Understanding the Delivery Delay of Addressed Issues in Large Software Projects

A thesis submitted to the Computer Science Graduation Program of the *Centro de Ciências Exatas e da Terra* in conformity with the requirements for the Degree of Doctor of Philosophy

Federal University of Rio Grande do Norte – UFRN

Centro de Ciências Exatas e da Terra

Programa de Pós-Graduação em Sistemas e Computação

Supervisor: Uirá Kulesza

Co-supervisor: Ahmed E. Hassan

Natal, RN, Brazil

February, 2017

Abstract

The timely delivery of addressed software issues (i.e., bug fixes, enhancements, and new features) is what drives software development. Previous research has investigated what impacts the time to triage and address (or fix) issues. Nevertheless, even though an issue is addressed, *i.e.*, a solution is coded and tested, such an issue may still suffer delay before being delivered to end users. Such delays are frustrating, since end users care most about when an addressed issue is available in the software system (i.e, released). In this matter, there is a lack of empirical studies that investigate why addressed issues take longer to be delivered compared to other issues. In this thesis, we perform empirical studies to understand which factors are associated with the delayed delivery of addressed issues. In our studies, we find that 34% to 98% of the addressed issues of the ArgoUML, Eclipse and Firefox projects have their integration delayed by at least one release. Our explanatory models achieve ROC areas above 0.74 when explaining delivery delay. We also find that the workload of integrators and the moment at which an issue is addressed are the factors with the strongest association with delivery delay. We also investigate the impact of rapid release cycles on the delivery delay of addressed issues. Interestingly, we find that rapid release cycles of Firefox are not related to faster delivery of addressed issues. Indeed, although rapid release cycles address issues faster than traditional ones, such addressed issues take longer to be delivered. Moreover, we find that rapid releases deliver addressed issues more consistently than traditional ones. Finally, we survey 37 developers of the ArgoUML, Eclipse, and Firefox projects to understand why delivery delays occur. We find that the allure of delivering addressed issues more quickly to users is the most recurrent motivator of switching to a rapid release cycle. Moreover, the possibility of improving the flexibility and quality of addressed issues is another advantage that are perceived by our participants. Additionally, the perceived reasons for the delivery delay of addressed issues are related to decision making, team collaboration, and risk management activities. Moreover, delivery delay likely leads to user/developer frustration according to our participants. Our thesis is the first work to study such an important topic in modern software development. Our studies highlight the complexity of delivering issues in a timely fashion (for instance, simply switching to a rapid release cycle is not a silver bullet that would guarantee the quicker delivery of addressed issues).

Keywords: Addressed Issues. Delivery Delay. Mining Software Sepositories. Software Maintenance.

Publications

Earlier versions of the work in this thesis were published as listed below:

- **Studying the Impact of Switching to a Rapid Release Cycle on Integration Delay of Addressed Issues - An Empirical Study of the Mozilla Firefox Project.** Daniel Alencar da Costa, Shane McIntosh, Uirá Kulesza, and Ahmed E. Hassan. In Proceedings of the 13th International Conference on Mining Software Repositories (MSR) , 2016, pp. 374–385.
  🏆*Received the ACM SIGSOFT distinguished paper award*🏆
- **An Empirical Study of Delays in the Integration of Addressed Issues.** Daniel Alencar da Costa, Shane McIntosh, Surafel Lemma Abebe, Uirá Kulesza, and Ahmed E. Hassan. In Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSME), 2014, pp. 281–290.
  🏆*Nominated for best paper award*🏆

The following publications are not directly related to the work that is presented in this thesis. Instead, they were produced in parallel to the research performed in this thesis.

- **A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes.** Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. In the Transactions of Software Engineering Journal (TSE), 2016, 18 pages.
- **How does the Shift to GitHub Impact Project Collaboration?** Luiz Felipe Dias, Igor Steinmacher, Gustavo Pinto, Daniel Alencar da Costa, and Marco Gerosa. In the 32nd International Conference on Software Maintenance and Evolution (ICSME-ERA), 2016, 5 pages.
- **Unveiling Developers Contributions Behind Code Commits: An Exploratory Study.** Daniel Alencar da Costa, Uirá Kulesza, Eduardo Aranha, and Roberta Coelho. In Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC), 2014, pp. 1152–1157.
- **Assessing and Evolving a Domain Specific Language for Formalizing Software Engineering Experiments: An Empirical Study.** Marília Freire, Uirá Kulesza, Eduardo Aranha, Gustavo Nery, Daniel Alencar da Costa, Andreas Jedlitschka, Edmilson Campos, Silvia Acuña, and Marta Gómez. International Journal of Software Engineering and Knowledge Engineering (IJSEKE), 2014, pp. 1509–1531.

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, lead their 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 their tasks.[1] Users can use ITSs to report issues within software systems. To do so, these users must fill in a report that contain information about the issue (*e.g.*, the description and severity of the issue).

The basic lifecycle of an issue is comprised of four steps. First, an issue is *reported* to the software project's team. Once reported, an issue has to be *triaged*, *i.e.*, the team members must decide whether an issue should be addressed or not. In case that an issue is deemed to be worth addressing, a team member with the right expertise is assigned to the issue (ANVIK; HIEW; MURPHY, 2006). After being triaged, an issue is *addressed* by its assignee, *i.e.*, a solution is provided and tested for that issue. Finally, the addressed issue is integrated and delivered to the end user through an official release of the software system. Issues may also be *reopened* if the solution that was provided to the issue is found to be incorrect. In this case, a new solution has to be provided, tested, and integrated.

1.1 Problem Statement

Once an issue is *addressed*, (*i.e.*, a solution for that issue is provided and tested), such addressed issue may still have a delay before reaching users. For instance, Jiang *et al.* (JIANG; ADAMS; GERMAN, 2013) find that a reviewed code change might take an additional 1-3 months to be integrated into the Linux kernel. Users care most about when addressed issues will be available in the software system (so they can benefit from those addressed issues). We use the term *delivery delay* to refer to the delay that addressed issues suffer prior to their delivery to end users.

[1] <https://www.bugzilla.org/>

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] Moreover, 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 a timely manner. In such a case, delivery delays frustrate these contributors.

The present thesis is an effort to reduce the lack of empirical understanding as to why addressed issues suffer delivery delay before being available to users. A good understanding of such delays will help software projects reduce such undesirable delays.

1.2 Current Research Limitations

Prior research has investigated the time that is needed to triage and 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. For example, issues might be addressed earlier given the estimated time that they are likely to take to be addressed. Nevertheless, after an issue is addressed, such an addressed issue may still require considerable time to be delivered to users.

Another line of prior work has investigated the integration stage of software development. Jiang *et al.* (JIANG; ADAMS; GERMAN, 2013) studied the likelihood of patches that were submitted to the Linux Kernel project of being integrated in the main code base. On the other hand, Choetkiertikul *et al.* (MORAKOT et al., 2015a; MORAKOT et al., 2015b) studied the risk of issues of postponing 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 release development strategies on such delays remain as open challenges.

1.3 Thesis Proposal

The general research question that is investigated in this thesis is *what leads addressed issues to 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. The studies that are

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

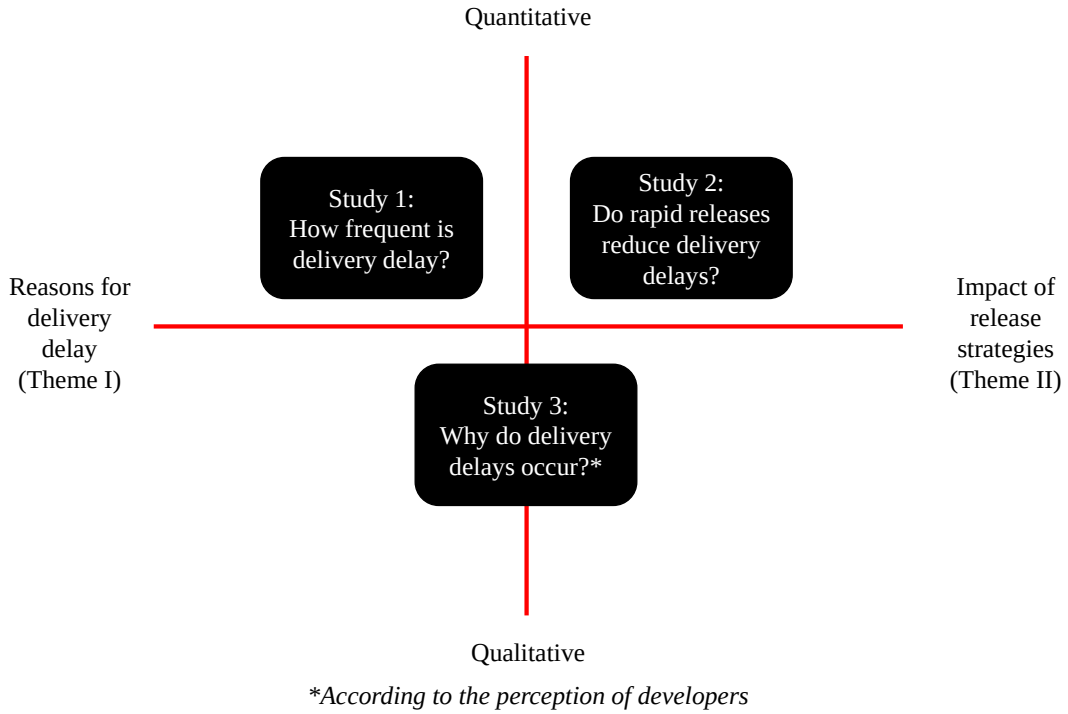


Figure 1 **–** An overview of the scope of the thesis.

performed in this thesis are grouped into two *themes*. *Theme I* is regarding *reasons for delivery delay*, which encompasses Studies 1 and part of Study 3. In *Theme II*, we investigate the *impact of release development strategies* on the delivery delay of addressed issues (Study 2 and part of Study 3). We present the motivation of performing each study of this thesis in the subsections below.

1.3.1 Study 1—How frequent is delivery delay?

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 stage may introduce non-trivial delays before delivering addressed issues. Since there is a lack of empirical studies that investigate the frequency of delivery delays of addressed issues, we perform a study using 20,995 addressed issues of the ArgoUML, Eclipse, and Firefox projects in Study 1. Our main goal is to analyze *(i)* how frequent delivery delays occur and *(ii)* which factors may impact delivery delay according to our studied data.

Also in this study, we investigate delivery delays that are considered to be *prolonged* in a particular project. For example, supposing that addressed issues are usually delivered within 60 days on a particular project, a delivery delay of 120 days would be abnormal for that project. This investigation is important because prolonged delays can be more frustrating to users, since they are not used to such delays.

1.3.2 Study 2—Do rapid releases reduce delivery delays?

After an issue is addressed, a release must be shipped in order to end users experience the addressed issue. The process of shipping releases varies according to the release cycles that are adopted by the project team. Recently, many organizations have shifted to shorter release cycles (*e.g.*, 6 weeks rathen than 12 months) with the allure of delivering software issues more quickly to end users. For instance, Firefox, Chrome, and Facebook have adopted shorter release cycles to ship major releases. In Study 2, we empirically study whether shorter release cycles quicken the delivery of addressed issues to end users. We set out to empirically compare the traditional and rapid releases of the Firefox project regarding the delivery delay of addressed issues. In total, we study 71,114 issue reports.

1.3.3 Study 3—Why do delivery delays occur?

In our other studies (Studies 1 and 2), we quantitatively investigate the delivery delay of addressed issues. We perform several statistical analyses based on the data that is publicly available on the ITSs and VCSs of our subject projects. Nevertheless, to better understand the reasons as to why delivery delays occur, we survey 37 participants from the ArgoUML, Firefox, and Eclipse projects about the delivery delay of addressed issues. We also perform follow up interviews with four participants to get deeper insights about the responses that we receive. Study 3 help us to *(i)* reach additional insights that could not be possible by only performing quantitative analysis and *(ii)* verify to which extent our participants agree with our findings from the quantitative studies.

1.4 Thesis Contributions

We outline the contributions of this thesis below. The contributions are grouped by their respective study.

Study 1—How frequent is delivery delay?

- Despite being addressed well before an upcoming release, 34% to 60% of the addressed issues are not integrated in more than one release in the ArgoUML and Eclipse projects. Furthermore, 98% of the Firefox project issues had their integration delayed by at least one release (Chapter 3).
- Heuristics that estimate the effort that teams invest in fixing issues are the most influential factors to estimate delivery delay in terms of number of releases

(Chapter 3).

- Surprisingly, *priority* and *severity* have little impact on delivery delay. Indeed, 36% to 97% of priority P1 addressed issues were delayed by at least one release (Chapter 3).
- Shorter delivery delays are associated with issues that are addressed during more controlled stages (*e.g.*, a code freeze stage) of a given release cycle (Chapter 3).
- The time at which issues are addressed and the resolvers of the issues have great impact on estimating the delivery delay (in terms of days) of an issue (Chapter 3).
- The time at which an issue is addressed (queue position), the integration workload (in terms of the backlog of addressed issues), and the heuristics that estimate the effort that teams invest in fixing issues (fixing time per resolver), are the most influential attributes for issues that have a prolonged delivery delay (Chapter 3).
- Our models that identify addressed issues that have a prolonged delivery delay outperform random guessing and Zero-R models, obtaining AUC values of 0.82 to 0.96 (Chapter 3).

Study 2—Do rapid releases reduce delivery delays?

- Although issues tend to be addressed more quickly in rapid release cycles, addressed issues tend to be integrated into consumer-visible releases more quickly in traditional release cycles. However, a rapid release cycle may improve the consistency of the delivery rate of addressed issues (Chapter 4).
- The total time that is spent from the issue report date to its integration into a release is not significantly different between traditional and rapid releases (Chapter 4).
- In traditional releases, addressed issues are less likely to be delayed if they are addressed recently in the backlog. On the other hand, in rapid releases, addressed issues are less likely to be delayed if they are addressed recently in the current release cycle (Chapter 4).

Study 3—Why do delivery delays occur?

- The perceived reasons for delivery delay of addressed issues are primarily related to activities such as development, decision making, team collaboration, and risk management (Chapter 4).

- The allure of delivering addressed issues more quickly to users is the most recurrent motivator for switching to a rapid release cycle. In addition, the allure of improving management flexibility and quality of addressed issues are other advantages of rapid releases that are perceived by our participants (Chapter 4).
- Integration rush and the increased time that is spent on polishing addressed issues (during rapid releases) emerge as one of the main explanations as to why traditional releases may have shorter delivery delays (Chapter 4).

1.5 Thesis Organization

The remainder of this thesis is organized as follows. In Chapter 2, we provide the background material to the reader. In Chapter 3, we present Studies 1 and **??**, while we present Studies 2 and 3 in Chapter 4. Finally, in Chapter 6, we draw our conclusions.

2 Background

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

2.1 Issue Reports

One of the main factors that drives software evolution is 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 may use an Issue Tracking System (ITS) such as Bugzilla[3] or JIRA.[4] Such ITSs allow for describing and monitoring the state 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 on a daily basis. For example, the Eclipse and Firefox projects respectively received an average of 65 and 89 issue reports daily (from January to October 2016) on their ITSs.[5,6] The number of filed issues is usually greater than the size of the development team.

Figure 2 shows the stages of an issue's life cycle. 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) (time *t1* of Figure 2).

After being triaged, issues are then *addressed* (or *fixed* in case of bugs), *i.e.*, solutions to the described issues are provided by developers (time *t2* of Figure 2). Generally speaking, an issue may be in an open or closed state. An issue is marked as open when a solution has not yet been found. We consider UNCONFIRMED, CONFIRMED, and IN_PROGRESS as open states. 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

[3] <https://www.bugzilla.org>
[4] <https://www.atlassian.com/software/jira>
[5] <https://bugs.eclipse.org/bugs>
[6] <https://bugzilla.mozilla.org/>

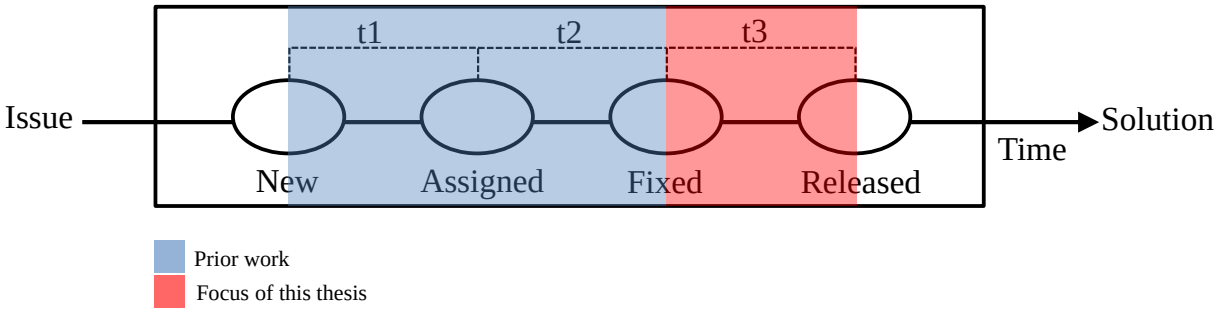


Figure 2 – An overview of an issue's life cycle.

made code changes to address an issue, the state and resolution combination should be RESOLVED-FIXED. However, if the developer was not able to reproduce the bug, then the state and resolution may be RESOLVED-WORKSFORME.[7]

Finally, addressed issues must be integrated into an official release (*i.e.*, releases that are intended for end users) in order to make them available (time *t3* of Figure 2), which is the life cycle stage that is mainly studied in this thesis. The life cycle of issues is documented in detail on the Bugzilla website.[8] In the next sections, we describe the stages of the life cycle of an issue.

Prior work studied the triaging and fixing time of issues (blue color in Figure 2). The focus of this thesis is the study of the delivery delay (red color in Figure 2), which is the needed time to deliver issues that are already addressed.

2.2 Triaging Issues

Issue triaging 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

[7] <https://bugzilla.mozilla.org/page.cgi?id=fields.html>
[8] <https://bugzilla.readthedocs.org/en/5.0/using/editing.html#life-cycle-of-a-bug>

the time to assign a developer and address such issues is approximately two years. Our research complements these prior studies by investigating the time that is necessary to deliver addressed issues rather than the time that is necessary to triage issues.

2.3 Addressing Issues

Once an issue is properly triaged, the assigned developer starts to address it. To estimate the required time to address issues, some approaches used the similarity of an issue report to prior 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 that was necessary to address issues in ArgoUML and PostgreSQL. They found that the median issue fixing 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 required time to address an issue. In our studies, however, we investigate the required time to deliver issues that are already addressed.

Recent empirical studies assess the relationship between the attributes that are used to build prediction models for estimating the fixing time of issues. Bhattacharya and Neamtiu (BHATTACHARYA; NEAMTIU, 2011) performed univariate and multivariate regression analyses to capture the significance of four attributes 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 that were reported to the Eclipse project and how severity and priority levels of the issues affect such a time. In their study, the authors used one way analysis of variance to group the different priority and severity levels that were used in the issue reports of the Eclipse project. Based on their results, the authors suggest the reduction of the currently used severity and priority levels to three levels.

Zhang *et al.* (ZHANG et al., 2012) investigated the delays incurred by developers in the issue addressing process. The authors extract the duration of an issue (*i.e.*, from open to closed) using interaction logs. The authors investigated the impact of three dimensions of attributes that are related to issues: issue report characteristics, source code, and code changes. The authors found that attributes such as severity, operating

system, issue description, and number of comments are likely to impact the needed time to start addressing an issue as well as the needed time to resolve an issue.

As Zhang *et al.* (ZHANG et al., 2012), we use attributes that are related to issue reports to build explanatory models. Nevertheless, our goal is to study attributes that share a relationship with the needed time to deliver addressed issues. We also investigate the impact that severity and priority levels have on the delivery delay of addressed issues.

2.4 Integrating Issues

After issues are addressed, they need to be integrated into an official release, so users can be benefited from them. Usually, user-intended releases are shipped along with release notes, which are documents that specify what was added, changed, or removed in such new releases.[9] Prior research has studied the integration of addressed issues. Jiang *et al.* (JIANG; ADAMS; GERMAN, 2013) studied the integration process of the Linux kernel. They found that 33% of 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. In this thesis, our focus is on the time that is required to deliver addressed issues rather than the process of patches acceptance or the risk of a release schedule slippage.

2.4.1 Delivery Delay

Delivery delay refers to the time between the moment at which an issue is addressed (*i.e.*, changed to the RESOLVED-FIXED status) to the time at which such an addressed issue is shipped to end users. In our quantitative studies (Studies 1, **??**, and 2), we analyze two *dimensions* of delivery delay. The first dimension is comprised of two types of delivery delay, which are: *(i)* delivery delay in terms of number of releases and *(ii)* delivery delay in terms of days. As for the second dimension, we study the *(iii)* prolonged integration time.

Definition 1—Delivery delay in terms of releases. Figure 3 provides an example of how we measure delivery delay. To compute the delivery delay in terms of number of releases, we count the number of releases that a given fixed issue is prevented from integration. In Figure 3, Issue #1 is reported at time t_1, fixed at t_3, and shipped at time t_5. The integration time in terms of releases for Issue #1 is the number of official releases

[9] <https://www.mozilla.org/en-US/firefox/releases/>

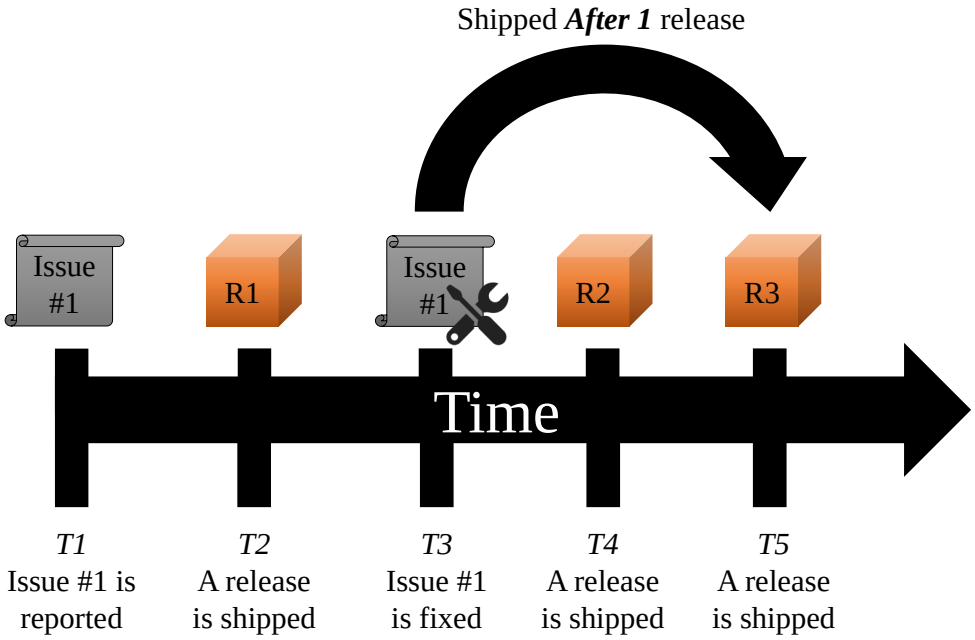


Figure 3 – An illustrative example of how we compute integration time.

that are shipped between t_3 and t_5. Therefore, Issue #1 has a delivery delay of one release.

Definition 2—Delivery delay in terms of days. We compute delivery delay in terms of days using an approach that is similar to Definition 1. However, instead of counting the number of official releases, we count the number of days between t_3 and t_5 (see Figure 3). For instance, if the number of days between t_i and $t_{(i+1)}$ in Figure 3 is 30 days, the delivery delay of issue #1 would be 60 days.

Definition 3—Prolonged delivery delay. Prolonged delivery delay occurs when the delivery delay in terms of days (see Definition 2) for a given addressed issue is above one *Median Absolute Deviation* (MAD) of the median delivery delay of a studied project. MAD is the median of the *absolute deviations* from one distribution's median. The higher the MAD, the greater is the variation of a distribution with respect to its median (HOWELL, 2005; LEYS et al., 2013).

2.5 Release Cycles

A *release cycle* is the time period that is required by the development team to develop and deliver a new release to end users. These releases 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 our studies, we consider a release cycle length in the scale of days or weeks

as a *rapid* release cycle. For example, the release cycle of the Firefox project is currently 6 weeks.[10] On the other hand, we consider a release cycle length of several months or years (*e.g.*, 12-18 months) as a traditional release cycle. In Study 2, we study the impact of adopting a rapid release cycle on the delivery delay of addressed issues.

2.6 Chapter Summary

In this chapter, we provide the key concepts that we use in our studies to the reader. We first present the concept of an *issue report*, which can either represent an enhancement, a new feature, or a bug that has to be addressed in a given project. Next, we describe the life cycle of an issue, which is basically comprised of the *triaging*, *addressing*, and *integration* stages. We then define the various types of *delivery delay* that we study in this thesis. Finally, we describe the two types of release cycles that are investigated in this thesis, which are the *rapid* and *traditional* release cycles.

[10] <https://wiki.mozilla.org/Release_Management/Release_Process>

3 How Frequent is Delivery Delay?

An ealier version of Study 1 appears in the proceedings of the International Conference on Software Maintenance and Evolution (ICSME'14) (COSTA et al., 2014).

3.1 Introduction

Since there is a lack of empirical studies that investigate the frequency of delivery delays of addressed issues, we perform a study using 20,995 addressed issues of the ArgoUML, Eclipse, and Firefox projects. Our main goal is to analyze *(i)* how frequent delivery delays occur and *(ii)* which factors may impact delivery delay according to our studied data. Finally, we also investigate *(iii)* what leads to a prolonged delivery delay. In this study, we address the following RQs:

- ***RQ1: How often are addressed issues prevented from being released?*** 34% to 60% of addressed issues within traditional release cycles (the ArgoUML and Eclipse projects) skip at least one release. Furthermore, the delivery of 98% of the addressed issues skip at least one release in the rapidly released Firefox project.

- ***RQ2: Does the stage of the release cycle impact delivery delay?*** We observe that issues that are addressed during more stable stages of a release cycle tend to have a shorter delivery delay. We also observe that addressed issues are unlikely to skip releases solely because they were addressed near a code freeze period.

- ***RQ3: How well can we model the delivery delay of addressed issues?*** Our models that are fit to study the delivery delay in terms of number of releases obtain AUC values of 0.62 to 0.93. Our models that are fit to study the delivery delay in terms of number of days obtain R^2 values of 0.39 to 0.65.

- ***RQ4: What are the most influential attributes for modeling delivery delay?*** We find that the total fixing time that is spent per resolver in the release cycle plays an

influential role in modeling the delivery delay in terms of releases of an addressed issue. On the other hand, we find that the time at which an issue is addressed and the resolver of the issue have a large influence on the delivery delay in terms of days. Moreover, attributes that are related to the state of the project are the most influential in both types of delivery delay.

- ***RQ5: How well can we identify the addressed issues that will suffer from a prolonged delivery delay?*** Our models outperform naïve models like random guessing, achieving AUC values of 0.82 to 0.96.

- ***RQ6: What are the most influential attributes for identifying the issues that will suffer from a prolonged delivery delay?*** Attributes that are related to the state of the project, such as the integration workload, the period during which issues are addressed, and the fixing time that is spent per resolver are the most influential attributes for identifying the issues that will suffer from a prolonged delivery delay.

Our results suggest that the total time that is invested per resolver in fixing the issues of a release cycle has a large influence later in the integration stage. Also, the number of issues that are waiting to be integrated can influence delivery delay. Such results warn us that in addition to the current focus of studies on triaging and fixing stages of the issue life cycle, the integration stage should also be the target of future research and tooling efforts in order to reduce the time-to-delivery of addressed issues.

Chapter Organization

This chapter is organized as follows. In Section 3.2, we present the methodology that is used in our study. In Section 3.3, we present our obtained results. In Section 3.4, we discuss and relate our observations along the studied types of delivery delay. We perform an exploratory analysis on the backlog of issues of each studied project in Section 3.5. In Section 3.6, we discuss the threats to the validity of our conclusions, while we position our work with respect to previous studies in Section 3.7. Finally, we draw conclusions in Section 3.8.

3.2 Methodology

In this section, we describe the studied projects, explain how the data was collected, and how we study the types of delivery delay that are presented in Section 2.4.1.

3.2.1 Subjects

In order to study delivery delay, we analyze three subject projects: the Firefox, ArgoUML, and Eclipse projects, which are from different domains and sizes. The ArgoUML project is a UML modeling tool that includes support for all standard UML 1.4 diagrams.[11] The Eclipse project is a popular open-source IDE, of which we study the JDT core subproject.[12] The Firefox project is a popular web browser.[13]

Figure 4 shows an exploratory analysis of our studied projects. We plot the proportion of issues per priority and severity level, as well as the proportion of issues that were addressed and not addressed (*e.g.*, resolution is *WONTFIX* or *WORKSFORME*). We observe that for the majority of the issues, the priority and severity levels remain at the default value. For example, the vast majority of the priority values are set to P3 (in the Eclipse and ArgoUML projects) or "- -" (in the Firefox project). We also observe that Firefox is the project with the highest proportion of addressed issues.

Table 1 shows the studied period and range of releases, as well as the number of releases and 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 projects. Each issue report corresponds to an issue that was addressed and could be mapped directly to a release. We present an overview of the release engineering processes of each studied project below.

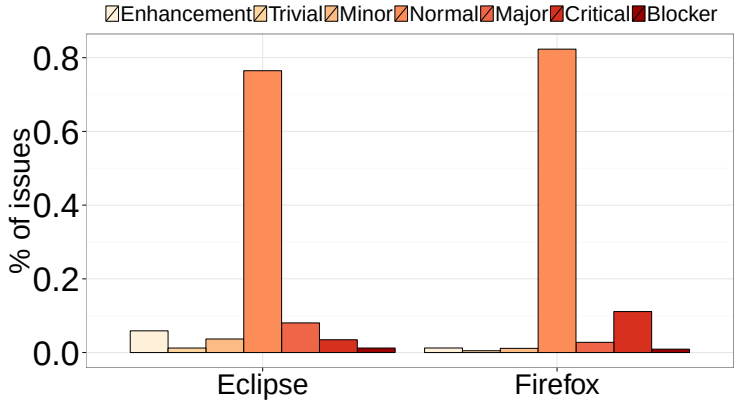
Eclipse Release Engineering

The release engineering of the Eclipse project is composed by *nightly/integration* builds that are followed by *milestones* builds and *release candidate* builds. Nightly or integration builds are the least stable builds and are tested by the early adopters that are following the eclipse developer mailing lists. For instance, integration builds are not supposed to be announced through links, blogs, or wikis that are related to the respective Eclipse project.[14]

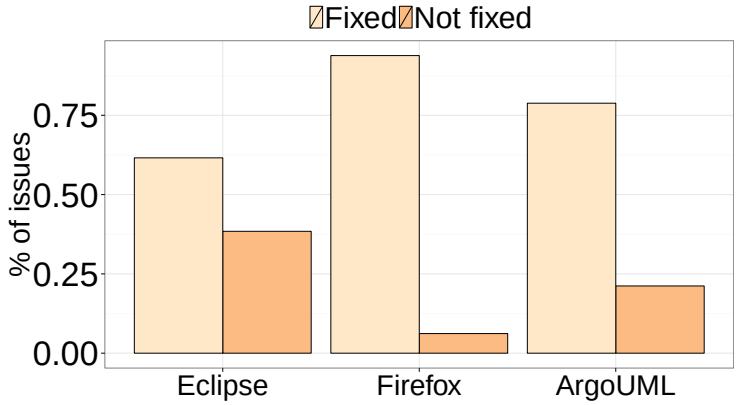
Milestone and *release candidate* builds are more stable and can be announced by external links such as blogs and wikis. The goal is to reach external early-adopters from outside the developer mailing lists. However, the external links that refer to such builds should warn that they are not as stable as official releases. The main difference between a release candidate build and a milestone build is that a release candidate

[11] <http://argouml.tigris.org/>
[12] <https://www.eclipse.org/>
[13] <https://www.mozilla.org>
[14] <https://eclipse.org/projects/dev_process/development_process.php#6_Development_Process>

(a) Priority values



(b) Severity values. The ArgoUML project does not use the severity field



(c) Fixed issues vs. Not fixed yet

***Figure 4* – Exploratory analysis of the studied projects.** We present the ratio of addressed issues per priority, severity, and the ratio of addressed vs. not addressed yet issues (*e.g.*, *WONTFIX* or *WORKSFORME*)

goes through a rigorous *testing pass* process.[15]

The testing pass process consists of intensive testing activities that are performed by the development team and community to find regression and *stop-ship* bugs. In case stop-ship bugs are found late in the process, the release schedule may be

[15] <https://www.eclipse.org/eclipse/development/plans/freeze_plan_4_4.php>

slipped to accommodate the fixes for such bugs.[15]

After the testing pass stage, a *fixing pass* stage starts. The fixing pass stage consists on prioritizing and fixing the most severe bugs that are found at the testing pass stage. By the end of a fixing pass stage, another release candidate is produced. The process of performing testing passes and fixing passes is done through several iterations (*i.e.*, many release candidates are produced).

The last release candidate is submitted to a *code freeze* stage. The *code freeze* is a period at which the rules to integrate changes in the software project becomes more strict. For instance, new changes may be integrated only if they are solving special requirements such as translations or documentation fixing.[15] Such a period is important because it helps the development team to stabilize the project just before creating an official release.

Official releases are categorized as *major*, *minor*, and *service* releases.[16] Major releases include API changes. Minor releases add new functionalities but are compatible with the API of prior versions. Finally, service releases include bug fixes only (*i.e.*, without significant addition of new functionality). Both major and minor releases have to pass through a *release review* process. A release review aims at getting feedback about the release cycle that was performed. The main goal is to find areas of improvement and if the development process is being open and transparent.[17]

Firefox Release Engineering

The release engineering process of the Firefox project uses a *rapid* (or *a short*) release cycle, *i.e.*, a release cycle of 6 weeks duration. In addition, the process also include *pipelining* releases (also known as *release training*) as a means to stabilize the official release, so that they can be shipped to end users.

The pipelining process develops releases through several channels. As the release progresses through these channels, the stability of the release increases and less severe bugs are more likely to be uncovered. The Firefox project team uses four channels to develop releases: *NIGHTLY*, *AURORA*, *BETA*, and *RELEASE* channels.[18]

The NIGHTLY channel produces a release every night (*i.e.*, as soon as features are ready). This nightly release is built from the *mozilla-central* repository and has the lowest stability of the channels.[19] The AURORA channel produces a release every six weeks. However, some new features may be disabled if they are not stable enough. At the end of the cycle of the AURORA channel (the sixth week), the release management

[16] <https://www.eclipse.org/projects/handbook/#release>
[17] <https://www.eclipse.org/projects/handbook/#release-review>
[18] <http://mozilla.github.io/process-releases/draft/development_overview/>
[19] <https://hg.mozilla.org/mozilla-central/>

team decides which of the issues that were further stabilized are good enough to migrate to the BETA channel. Again, the goal of the BETA channel is to stabilize the new features and disable the features that are not stable enough by the end of the cycle. Finally, the features that are stable enough to survive at the BETA channel are moved further to the RELEASE channel, from which an official major release is produced.[18]

In the Firefox release engineering process, the release schedule is not slipped to accommodate issues that are not stable enough by the end of the release cycle. Instead, the development team holds such issues back to be shipped in future releases when a greater degree of stability is achieved.[18] Also, an issue may be integrated directly into the AURORA or BETA channels (*i.e.*, the issue is *uplifted*), but such cases are exceptions (*e.g.*, very critical security issues that must be released as soon as possible).[18]

The Firefox project also ships *Extended Support Releases* (ESR) that are based on prior official Firefox releases. ESRs are meant to institutions such as business organizations, schools, and universities that need to manage their Firefox desktop client. ESRs provide one year of support for security and bug fixes of prior Firefox official releases. ESRs are important for organizations that are not able to follow the fast pace that the Firefox major release evolves.[20]

ArgoUML Release Engineering

In the ArgoUML release engineering process, there are five types of releases: *development*, *alpha*, *beta*, *stable*, and *stable patch* releases. *Development* releases are the least stable, while *stable* releases are the official releases that are intended to be widely adopted by the users.[21]

Development releases are generated during the *development* stage. The *development* stage may take from one to several months. During this stage, the development team strives to produce a *development* release each month. *Development releases* are not supposed to be used by end users. Such releases are only advertised to users if there is a purpose of recruiting new developers to implement and test new features.[21]

After the *development* stage, the *alpha* stage starts. The *alpha* stage is also referred as the *enhancement freeze* point. All of the enhancements that are not stable enough before the start of the *alpha* stage are not included into the *stable* release. According to the ArgoUML documentation, the *alpha* stage usually takes a "*couple of weeks*" and the development team strives to make a release each week.[21]

The *alpha* stage is followed by the *beta* stage. The *beta* stage is also referred as the *bug-fix freeze* point, *i.e.*, all of the (less severe) bug-fixes that could not be completed

[20] <https://www.mozilla.org/en-US/firefox/organizations/faq/>
[21] <http://argouml.tigris.org/wiki/How_to_Create_a_Stable_Release>

***Table 1* – Overview of the studied projects.** We present the number of studied releases, issues, the studied period and the median time between releases.

Project	Studied period	Releases	# of releases	# fixed issues	Median time between releases (weeks)
Eclipse (JDT)	03/11/2003 - 12/02/2007	2.1.1 - 3.2.2	11	3344	16
Firefox	05/06/2012 - 04/02/2014	13 - 27	15	3121	6
ArgoUML	18/08/2003 - 15/12/2011	0.14 - 0.34	17	14530	26

before the start of the *beta* stage are omitted from the *stable release*. Such remaining bugs are listed on the "*known problems*" document that is to be published along with the *stable* release. *Beta* releases are more stable than *alpha* releases and are also referred as *release candidates*. For instance, *beta* releases should not contain high priority bugs (*i.e.*, issues for which the priority is either P1 or P2). The *beta* stage is supposed to last for a couple of weeks with a *beta* release being generated each week. Finally, the *beta* stage is marked by intense testing activities after each release candidate. When the team is confident that the *beta* release is stable enough, the official *stable* release is generated with no code changes from the last *beta* release.[21]

The last type of ArgoUML release is the *stable patch* release. *Stable patch* releases are generated if critical bugs are found after the publication of the *stable* release. The *stable patch* release contains the fixes for the eventual critical bugs that are found upon *stable* releases.[21] The ArgoUML team strives to ship a *stable* release every 8 months.[22]

3.2.2 Data Collection

Figure 5 provides an overview of our data collection approach—how we collect and organize the data in order to perform our empirical study. We create a relational database that describes the integration of fixed issues in the studied projects. We briefly describe our data sources, and each step that are involved in the database construction process.

Step 1: Fetch integrated issue IDs

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

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

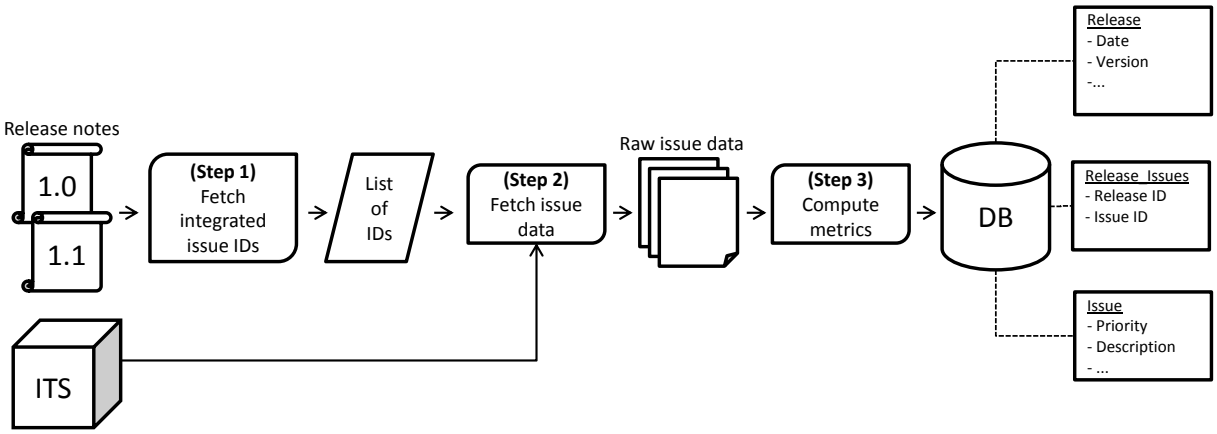


***Figure 5* – Data collection.** An overview of our approach to collect the needed data for studying delivery delay.

Eclipse, ArgoUML, and Firefox projects publish their release notes on their respective websites.[23]

Unfortunately, release notes may not mention all of the fixed issues that have been integrated into a release. This limitation hinders the possibility of studying issues that were fixed but have not been integrated, since we cannot claim that an issue that is not listed in a release note was not integrated (*e.g.*, the development team may forget to list some integrated fixed issues). However, the fixed issues that are listed in a release note are more likely to have been shipped to the end users (*i.e.*, it is unlikely that a release note would mention a fixed issue that was not integrated). Hence, we choose to use release notes as a means of linking fixed issues to releases in our database, despite the incompleteness of such release notes—the release where we claim that an issue has been integrated is more likely to be correct (we elaborate more on this point in Section 3.6).

The output of Step 1 is a list of the issue IDs that have been fixed and integrated. To retrieve such a list for the Eclipse and Firefox projects, we wrote a script to extract the listed issue IDs from all the release notes and insert them into our database. The retrieved issue IDs are used to fetch the issue report meta-data from the corresponding ITSs. In our database, we also store the dates and version number of each release.

Step 2: Fetch issue data

We use the collected issue IDs from Step 1 to retrieve information from their corresponding issue reports, which are recorded in the ITSs. Not all release notes of the ArgoUML project list the fixed issues of an official release. When they do, only a few issues are listed (*e.g.*, 1-4).[24] To increase our sample of fixed issues for the ArgoUML

[23] <https://www.mozilla.org/en-US/firefox/releases/>
[24] <http://argouml.tigris.org/wiki/ReleaseSchedule/Past_Releases_in_Detail>

project, we rely on its ITS. We use the milestone field of the issue reports to approximate the release into which an issue was integrated. Development milestones are counted towards the next official releases. For instance, the development milestone 0.33.7[25] is counted towards the official release 0.34. The output of Step 2 is the raw issue report data that is collected from ITSs.

Finally, to determine when an issue was fixed, we use the latest change to the RESOLVED-FIXED status of that issue. For instance, if an issue has its status changed from RESOLVED-FIXED to REOPENED at t_1 and the status changes back to RESOLVED-FIXED at t_2 (without changing again), we consider the corresponding date of t_2 as the fix date. Also, we use the RESOLVED-FIXED status rather than the VERIFIED-FIXED status, since we found that all of the issues that are mapped to releases went through the RESOLVED-FIXED state before being integrated, while only a small percentage went through the VERIFIED-FIXED state. For example, only 17% of fixed issues in the Firefox project went through the VERIFIED-FIXED state. We focus on issues that were resolved as RESOLVED-FIXED because they involve changes to the source and/or test code that must be integrated into a release before becoming visible to end users.

Step 3: Compute metrics

After collecting the release date for each addressed issue, we compute all of the attributes that may share a relationship with the types of delivery delay that are presented in Section 2.4.1.

We first compute the delivery delay of addressed issues in terms of number of releases (see Definition 1). We group this type of delivery delay into four buckets: *next*, *after-1*, *after-2*, and *after-3-or-more*. The *next* bucket contains addressed issues that are integrated immediately. The *after-1*, *after-2*, and *after-3-or-more* buckets contain addressed issues for which integration is skipped by one, two, or three or more releases, respectively. Figure 6 shows the distribution of the addressed issues among buckets for each studied project. The ArgoUML project has the highest percentage of addressed issues that fall into the *next* bucket (66%), whereas *next* accounts for only 2% and 38% of addressed issues in the Firefox and Eclipse projects, respectively.

Next, we compute the delivery delay in terms of number of days (see Definition 2). Figure 7 shows the distribution of delivery delay in terms of days for each studied project. The Firefox project has the least skewed distribution of delivery delay. We use both Definitions 1 and 2 of delivery delay to address RQ1-RQ4.

Finally, we identify issues that have a prolonged delivery delay in each studied project (see Definition 3). We group addressed issues into *prolonged delay* and *normal*

[25] <http://argouml.tigris.org/issues/show_bug.cgi?id=4914>

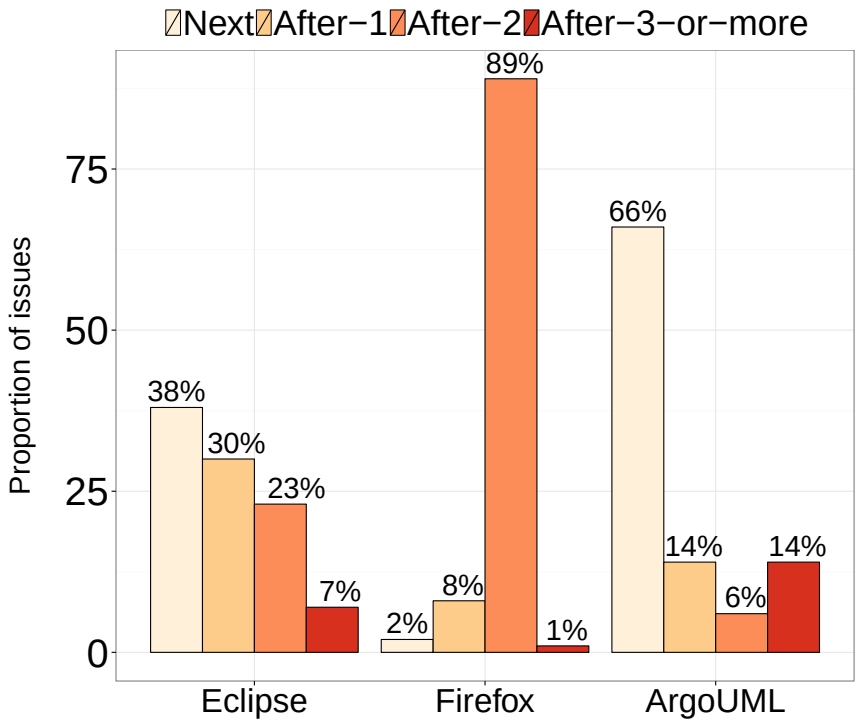


***Figure 6* – Distribution of addressed issues per bucket.** The issues are grouped into *next*, *after-1*, *after-2*, and *after-3-or-more* buckets.

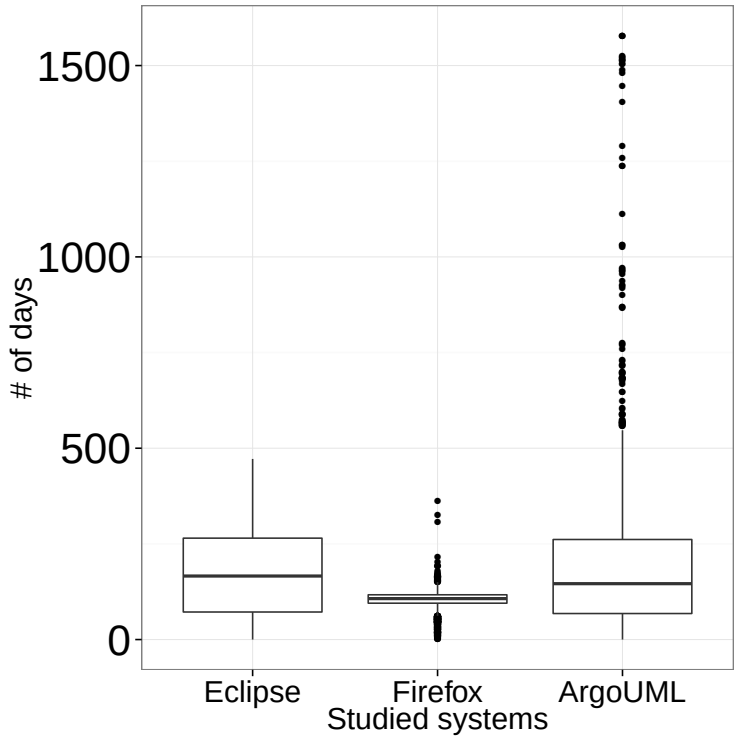


***Figure 7* – Delivery delay in terms of days.** The medians are 166, 107, and 146 days for the Eclipse, Firefox, and ArgoUML projects, respectively.

delay buckets. Addressed issues, of which delivery delay is at least one MAD above the median delivery delay of a subject project, fall into the *prolonged delay* bucket. Figure 7 shows that a prolonged delivery delay in one project may be a normal delivery delay in another project (*e.g.*, the ArgoUML project *vs.* the Firefox project). This figure highlights the importance of performing this analysis for each project individually. We use this data to address RQ5 and RQ6.

We use exploratory models to study the relationship between attributes of addressed issues (*e.g.*, severity and priority) and delivery delay. Our goal is to understand which attributes are important for modeling the delivery delay of addressed issues.

3.3 Results

In this section, we present the motivation, approach, and results for each investigated RQ.

3.3.1 RQ1: How often are addressed issues prevented from being released?

RQ1: Motivation

Users and contributors care most about the time for an addressed issue to become available rather than the time duration to fix it. In this regard, it is important to investigate whether addressed issues are being integrated immediately (*e.g.*, in the next possible release) or not, since a large delivery delay may frustrate users. In RQ1, we investigate how often addressed issues are being prevented from integration. The analysis of RQ1 is our first step toward understanding how long is the delivery delay of addressed issues.

RQ1: Approach

We compute the delivery delay of addressed issues in terms of number of releases and number of days (as shown in Definitions 1 and 2). Next, we analyze if addressed issues are being prevented from being released solely because their fix occurs in the end of their release cycle. For instance, Rahman and Rigby (RAHMAN; RIGBY, 2015) observe a rush-to-release in which many issues are addressed near the release date. For each addressed issue, we compute the *fix timing* metric, which is the ratio between (i) the remaining number of days—after an issue is addressed—for an up-

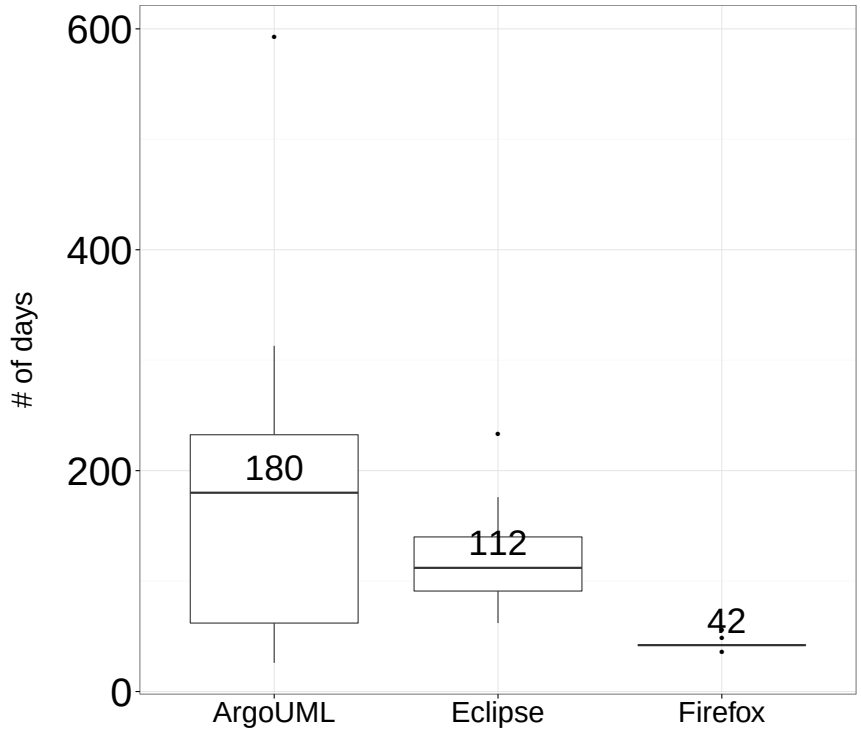


***Figure 8* – Number of days between the studied releases of the ArgoUML, Eclipse, and Firefox projects.** The number shown over each boxplot is the median interval.

coming release over (ii) the duration in terms of days of its respective release cycle (see Equation 3.1). The *fix timing* values range from 0 to 1. A *fix timing* value close to 1 indicates that an issue is addressed early in the release cycle, since the numerator and denominator of Equation 3.1 would be close to each other.

$$\frac{\text{\# days that is remaining for a release}}{\text{release cycle duration}} \tag{3.1}$$

RQ1: Results

Addressed issues usually miss the next release in the Firefox project. Figure 8 shows the difference between the studied projects in terms of the time interval between their releases. The median time in days for the Firefox project (42 days) is approximately $\frac{1}{4}$ that of the ArgoUML project (180 days), and $\frac{1}{3}$ that of the Eclipse project (112 days). Unlike the Eclipse and Firefox projects, the distribution for the ArgoUML project is skewed. In addition, Figure 6 shows that the vast majority of addressed issues for the Firefox project is integrated *after-2* releases, whereas for the Eclipse and ArgoUML projects, the majority is integrated in the *next* release.

The reason for the difference may be due to the release policies that are followed in each project. For example, Figure 8 shows that the Firefox project releases

consistently every 42 days (six weeks), whereas the time intervals between the releases of the ArgoUML project vary from 50 to 220 days. Indeed, the release guidelines for the ArgoUML project state that the ArgoUML team should release at least one stable release every 8 months (see Section 20). The delivery consistency of the Firefox releases might lead to addressed issues being prevented from a greater number of releases, since the Firefox project rigidly adhere to a six-week release schedule despite accumulating issues that could not be integrated (see Section 17).

Although an addressed issue usually misses the next release in the Firefox project, issues are usually shipped faster when compared to the other projects. Indeed, Figure 7 shows that addressed issues in the Firefox project take a median of 107 days to be released, while it takes 166 and 146 days in the Eclipse and ArgoUML projects, respectively.

34% to 60% of addressed issues had their integration prevented from at least one release in the traditionally released projects. Figure 6 shows that 98% of the addressed issues in the Firefox project are prevented from integration in at least one release. However, for the projects that adopt a more traditional release cycle, *i.e.*, the ArgoUML and Eclipse projects, 34% to 60% of the addressed issues are prevented from integration in at least one release. This result indicates that even though an issue is addressed, integration may be prevented by one or more releases, which can frustrate end users.

Many issues that were prevented from integration are addressed well before the upcoming release date. addressed issues could be prevented from integration because they were addressed late in the release cycle, *e.g.*, one day or one week before the upcoming release date. To check whether addressed issues are being prevented from integration mostly because they are being addressed late in the release cycle, we compute the *fix timing* metric.

Figure 9 shows the distribution of the *fix timing* metric for each project. The smallest *fix timing* median is observed for the Eclipse project, which is 0.45. For the ArgoUML and Firefox projects, the median is 0.52 and 0.53, respectively. The *fix timing* 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, issues that are prevented from integration are usually addressed $\frac{1}{4}$ to $\frac{3}{4}$ of the way through a release. Hence, it is unlikely that most addressed issues are prevented from integration solely because they were addressed too close to an upcoming release date.

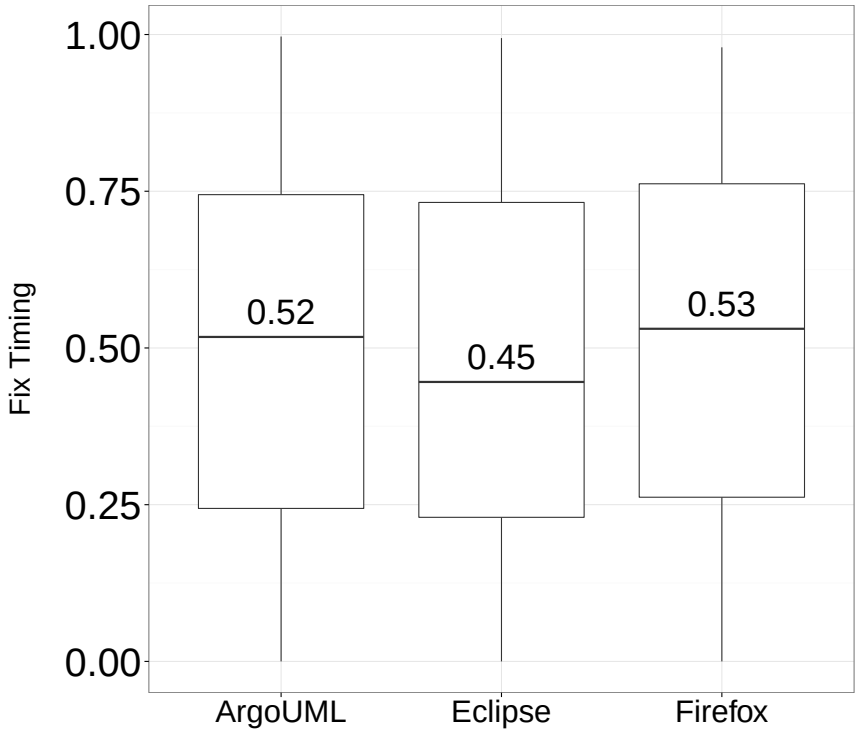


***Figure 9* – Fix timing metric.** We present the distribution of the *fix timing* metric for addressed issues that are prevented from integration in at least one release.

The integration of 34% to 60% of the addressed issues in the traditionally released projects and 98% in the rapidly released project were prevented from integration in at least one release. Furthermore, we find that many issues which integration was prevented, were addressed well before the releases from which they were omitted.

3.3.2 RQ2: Does the stage of the release cycle impact delivery delay?

RQ2: Motivation

An issue that is *addressed* before the production of a *release candidate* may receive more attention, which may lead to a shorter delivery delay. Analysis of the impact of integration stage may help researchers and practitioners to reflect on how to reduce delivery delay or to increase awareness about it.

RQ2: Approach

For each studied project, we tag addressed issues according to the stage during which they were addressed. For example, if an issue was addressed during the *beta*

stage of the Firefox project (*i.e.*, at the BETA channel), we tag such issue as being "*addressed during beta*". We then compare the distributions of delivery delay in terms of days (Definition 2) among the different stages of a release cycle. For example, in the Firefox project, we compare the distributions of delivery delay between the *NIGHTLY*, *ALPHA*, and *BETA* stages, since the *RELEASE* stage corresponds to the official release itself.

To check whether there is at least one statistically significant difference among distributions of delivery delay, we use the Kruskal-Wallis test (KRUSKAL; WALLIS, 1952). This test checks if two or more samples are likely to come from the same population (*null hypothesis*). However, when there are three or more distributions, the Kruskal-Wallis test does not indicate which distribution is statistically different with respect to the others. For specific comparisons between distributions, we use the Dunn test (DUNN, 1964). The Dunn test shows which distribution is statistically different from the others. To counteract the problem of multiple comparisons (DUNN, 1961), we use the Bonferroni correction to adjust our obtained $p\text{-}values$.

Finally, we use Cliff's delta to check the magnitude of the observed differences (CLIFF, 1993). For example, two distributions may be statistically different, but the magnitude of such a difference may be negligible. The higher the value of the Cliff's delta, the greater the magnitude of the difference between distributions. We use the thresholds provided by Romano *et al.* (ROMANO et al., 2006) to perform our comparisons: $delta < 0.147$ (*negligible*), $delta < 0.33$ (*small*), $delta < 0.474$ (*medium*), and $delta >= 0.474$ (*large*).

We also compute the *fix timing* metric (as in RQ1). However, this time we check whether addressed issues are being prevented mostly because they were performed near a code freeze date—rather than the upcoming release date. Equation 3.2 shows how we adapt Equation 3.1 to compute the *fix timing* metric to account for the code freeze date. For the Eclipse project, we consider the date of the last *release candidate* as the code freeze stage, while we consider the date of a *beta stage* as the code freeze stage in the ArgoUML project.

$$\frac{\text{\# days that is remaining for a code freeze}}{\text{release cycle duration}} \tag{3.2}$$

RQ2: Results

Issues that are addressed during more stable stages of a release cycle have a shorter delivery delay. Figure 10 shows the distributions of delivery delay (in terms of days) per each release cycle stage of the studied projects. For the Eclipse project, the stages are divided into *milestones*, *RCs* (Release Candidates), and *code freeze* (see the release

***Table 2* – Statistical analysis.** An overview of the $p\text{-}values$ and $deltas$ that are observed during our statistical analyses.

	Comparison	**Kruskal-Wallis** (p)	**Dunn** ($p.adjusted$)	**Effect-size** ($delta$)
Eclipse	Milestones vs RCs	1.87×10^{-51}	1.47×10^{-52}	(large) 0.63
	RCs vs Code freeze		0.56	Not apply
	Milestones vs Code freeze		0.02	(negligible) 0.09
Firefox	Nightly vs Aurora	2.99×10^{-76}	5.07×10^{-49}	(medium) 0.40
	Aurora vs Beta		1.72×10^{-03}	(medium) 0.40
	Nightly vs Beta		1.43×10^{-31}	(large) 0.57
ArgoUML	Development vs Alpha	2.73×10^{-135}	7.24×10^{-89}	(large) 0.94
	Alpha vs Beta		3.98×10^{-09}	(large) 0.98
	Development vs Beta		1.14×10^{-78}	(large) 0.99

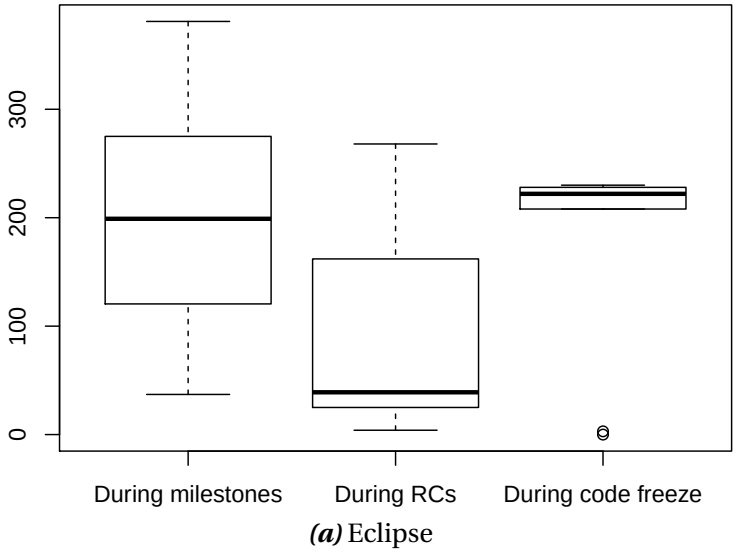
engineering process of Eclipse). Indeed, issues that are addressed during RCs have a shorter delivery delay when compared to issues that were addressed during milestone releases. For the difference between *milestones* and *RCs*, we observe a $p = 1.47 \times 10^{-52}$ and a *large* effect-size of $delta = 0.63$. All of the $p\text{-}values$ and $deltas$ of our statistical analysis are shown in Table 2. Even though delivery delay seems to be larger during the *code freeze* stage, we do not observe a significant $p\text{-}value$ when comparing the *code freeze* stage to the other stages. In fact, only ten issues were addressed during the *code freeze* stage in our data, which impairs statistical observations of trends in such a stage.

For the Firefox project, we observe that delivery delay tends to be shorter as fixes are performed along more stable stages. For example, by comparing the delivery delay values between the *NIGHTLY* and *AURORA* stages, we observe a $p = 5.1 \times 10^{-49}$ and a *medium* effect-size of $delta = 0.40$ (the other comparisons are shown in Table 2).

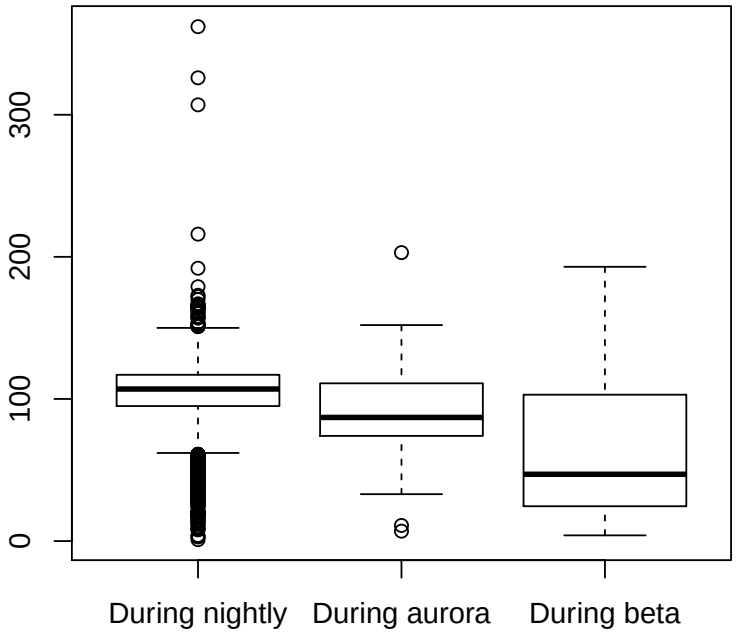
Finally, for the ArgoUML project, we also observe a trend of shorter delivery delay as the fixes are performed during more stable stages of release cycles. For instance, when we compare the delivery delay of addressed issues of the *alpha* and *beta* stages, we obtain a $p\text{-}value$ of 3.98×10^{-09} and a *large* effect-size of $delta = 0.98$.

Many issues that are prevented from integration are addressed well before the code freeze stage of their respective release cycle. We compute the *fix timing* metric that we present in RQ1. However, instead of counting the number of days until an upcoming release, we count the number of days until an upcoming code freeze stage (Equation 3.2). Our goal is to check whether addressed issues are being prevented from integration mostly because they are being addressed too close to a code freeze stage (*i.e.*, a period during which integration of new code changes would likely be minimal).

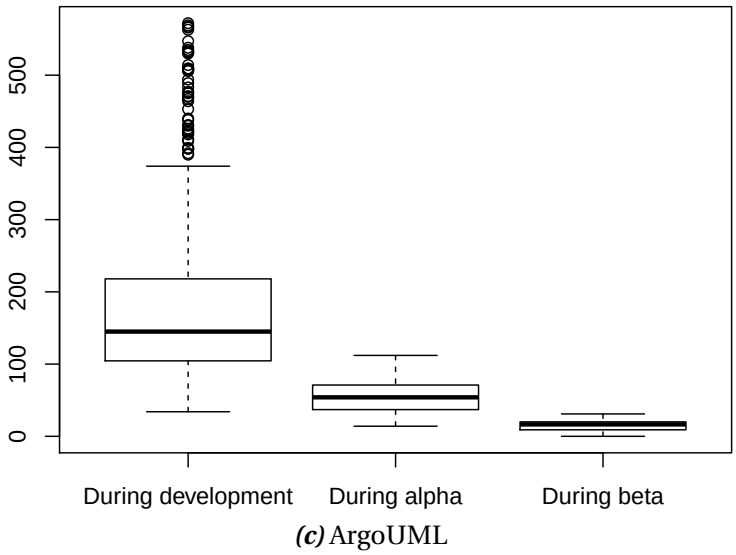
In Figure 11, we show the *fix timing* values for the Eclipse and ArgoUML projects, since both projects adopt a *code freeze* stage. For the Eclipse project, the code freeze starts after the last release candidate, while for the ArgoUML project, the code freeze starts at the beginning of the *beta* stage (see Section 3.2.1). Naturally, we observe a drop



(a) Eclipse



(b) Firefox



(c) ArgoUML

***Figure 10* – delivery delay during release cycle stages.** Issues that are addressed during more stable stages of a release cycle are likely to have a shorter delivery delay

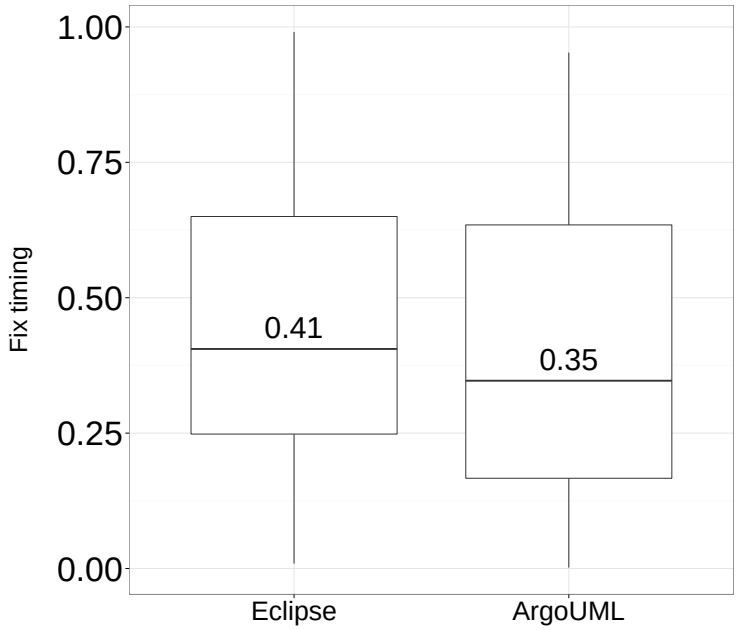


***Figure 11 – Fix timing* values for the code freeze period.** The median *fix timing* values drop from 0.45 and 0.52 to 0.41 and 0.35 in the Eclipse and ArgoUML projects, respectively.

in the *fix timing* values, since both code freeze stages start considerably before the official release dates. Nevertheless, we observe that even after correcting for the code freeze stages of the Eclipse and ArgoUML projects, it is unlikely that addressed issues are being prevented from integration solely because of an approaching code freeze stage. For instance, although the median *fix timing* for the ArgoUML project dropped from 0.52 to 0.35, the development team would still have 2 months to integrate an addressed issue—since the median duration of a release cycle in the ArgoUML project is 180 days.

We observe that issues that are addressed during more stable stages of release cycles are associated with a shorter delivery delay. We also observe that addressed issues are unlikely to be prevented from integration solely because they were addressed near an upcoming code freeze stage.

3.3.3 RQ3: How well can we model the delivery delay of addressed issues?

RQ3: Motivation

Several studies have proposed approaches to investigate the time that is required to fix 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 prevented from integration before reaching 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 delivery delay of addressed issues is of great interest. In RQ3, we investigate if we can accurately model integration time in terms of number of releases and days (*i.e.*, Definitions 1 and 2 of delivery delay). Our explanatory models are important to understand which attributes may impact delivery delay of addressed issues. Moreover, such type of models could be used by practitioners to estimate when an addressed issue will likely be integrated.

RQ3: Approach

In order to study when an addressed issue is integrated, we collect information from both the ITSs and VCSs of the studied systems. We train models using attributes that are grouped in the following families: *reporter*, *resolver*, *issue*, *project*, and *process*.

- **Reporter:** refers to the attributes regarding an issue reporter. Issues that are reported by a reporter who is known to report important issues may receive more attention from the integration team.

- **Resolver:** refers to team members that fix issues. Issues that are addressed by experienced resolvers may be easier to integrate and ship to end users.

- **Issue:** refers to the attributes of issues reports. Project teams use this information to triage, fix, and integrate issues. For example, integrators may not be able to properly assess the importance and impact of poorly described issues, which may increase delivery delay.

- **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 are likely to have a longer delivery delay.

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

***Table 3* – Reporter, Resolver and Issue families.** Attributes of the Reporter, Resolver and Issue families that are used to model the delivery delay of addressed issues

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.
	Integration speed	Numeric	**d:** Measured by the median integration time of prior issues that were reported by a given reporter.
			r: If issues that are reported by a given reporter are integrated quickly, future issues reported by the same reporter may also be integrated quickly.
Resolver	†Experience	Numeric	**d:** Experience in fixing issues for the project. It is measured by the number of prior issues that were addressed by a given resolver.
			r: An issue that is addressed by an experienced resolver may be easier to integrate.
	†Integration speed	Numeric	**d:** Measured by the median integration time of prior addressed issues.
			r: If the prior addressed issues of a particular resolver were quickly integrated, future issues that are addressed by the same resolver may also be quickly integrated.
Issue	Component	Nominal	**d:** The component to which an issue is being reported.
			r: Issues that are related to a given component (*e.g.*, authentication) might be more important, and thus, might be integrated more quickly than issues that are reported to 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 more quickly than issues that are reported to less important platforms.
	Severity	Nominal	**d:** The severity level that is recorded in the issue report.
			r: severe issues (*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 the Eclipse project (PANJER, 2007).
	Priority	Nominal	**d:** The priority that is assigned to the issue report.
			r: High priority issues will likely be integrated before low priority issues.
	†Stack trace attached	Boolean	**d:** We check whether the issue report has a stack trace attached in its description.
			r: A stack trace attached in the description of the issue report may provide useful information with respect to the cause of the issue, which may quicken the integration of that addressed issue (SCHROTER; BETTENBURG; PREMRAJ, 2010).
	Description Size	Numeric	**d:** Description of the issue measured by the number of words in its description.
			r: Issues that are well-described might be easier to integrate than issues that are difficult to understand.

† New attributes that did not appear in our previous work (COSTA et al., 2014)

more difficult to understand and integrate.

***Table 4* – Project family.** Attributes of the Project family that is used to model the delivery delay of an addressed issue.

Family	Attributes	Value	Definition (d)|Rationale (r)
Project	Backlog of issues	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 team members, which 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 at which an issue was addressed in relation to others in the current release cycle. The rank is divided by all of the issues that were addressed until the end of the current release cyle.
			r: An issue that is near the front of the queue is more likely to be quickly integrated.
	†Fixing time per resolver	Numeric	**d:** $\frac{\sum_{issue=1}^{total} \text{fixing time}}{\text{\# of resolvers}}$, the sum of the time (measured in terms of days) to fix the issues of the current release cycle over the number of resolvers that worked in that release cycle (BROOKS, 1975).
			r: The higher the total fixing time that is spent per resolver in fixing issues the less the likelihood of an addressed issue experiencing a large delivery delay.
	†Backlog of issues per resolver	Numeric	**d:** The number of issues in the RESOLVERD-FIXED state at a given time for each resolver of the development team.
			r: Having a large number of addressed issues per resolver might create a workload on that resolver to integrate the issue.

† New attributes that did not appear in our previous work (COSTA et al., 2014)

Tables 3, 4 and 5 describe the attributes that we compute for each family. For each attribute, Tables 3, 4 and 5 present our rationale for using it in our models. We choose these families of attributes because (i) we intend to study a variety of perspectives that might influence delivery delay and (ii) they are simple to compute using the publicly available data sources (*e.g.*, ITSs and VCs) from our studied systems.

We train exploratory models to study how many releases an addressed issue is likely to be prevented from integration (Definition 1). To study delivery delay in terms of releases, we use the *random forest* classification technique (BREIMAN, 2001). Random forest is an *ensemble learning* technique that operates by combining a multitude of decision trees at the training stage. Each decision tree uses a random subset of the attributes that are used to explain one phenomenon (*e.g.*, delivery delay). Next, each decision tree votes for the response bucket (*e.g.*, *next* or *after-1* release(s)) of a given instance. The majority of the votes for a given bucket will be the actual response of the random forest. We choose random forests because they are 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 tend to be noisy (HERRAIZ et al.,

***Table 5* – Process family.** Attributes of the Process family that is used to model the delivery delay of an addressed issue.

Family	Attributes	Value	Definition (d)|Rationale (r)
Process	Number of Impacted Files	Numeric	**d:** The number of files that are linked to an issue report.
			r: delivery delay might be related to a high number of impacted files because more effort would be required to properly integrate code 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 fix the issue, which can impact the integration time of an issue. (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 delivery delay (JIANG; ADAMS; GERMAN, 2013).
	Number of Tosses	Numeric	**d:** The number of times that the issue's assignee has changed.
			r: The number of changes in the issue assignee might indicate a complex issue to fix or a difficulty in understanding such an issue, which can impact delivery delay. One of the reasons for changing the assigned developer is because additional expertise may be required to fix an 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 fix the issue, and hence, verifying the impact of integrating the modifications may also be difficult (NAGAPPAN; BALL, 2005; JIANG; ADAMS; GERMAN, 2013).
	†Fixing time	Numeric	**d:** The number of days between the moment at which an issue is opened the moment at which the issue is addressed (*i.e.*, the issue reaches the RESOLVED-FIXED status) (GIGER; PINZGER; GALL, 2010).
			r: Issues that are addressed quickly might indicate that the necessary code changes are easy to integrate, wich may quicken integration time.

† New attributes that did not appear in our previous work (COSTA et al., 2014)

2008). In our study, we use the *random forest* implementation provided by the *bigrf* R package.[26]

Since our data has a temporal order, *i.e.*, the values of the attributes for each instance depends on the time at which the issue was addressed, we evaluate our models

[26] Bigrf package <https://cran.r-project.org/src/contrib/Archive/bigrf/>

by adapting the *Leave One Out Cross Validation* (LOOCV) technique. In our LOOCV variation, we first sort the data by the date at which the issues were addressed. Then, we train models to predict each next instance of the data. For example, if issue A is addressed before issue B, we train a model using A and test it using B. Furthermore, if issues A and B are addressed before C, we train a model using A and B and test it using C. This process is repeated until we test a model by using the last addressed issue in our data.

We evaluate the performance of our random forest models using the *precision*, *recall*, *F-measure*, and *AUC*. We also use *Zero-R* models as a baseline to compare the results of our models, since no prior models have been proposed to model delivery delay. We describe each one below.

Precision (P) measures the correctness of our models in estimating the number of releases that are necessary to ship an addressed issue. An estimation is considered correct if the estimated delivery delay is the same as the actual delivery delay of an addressed issue. Precision is computed as the proportion of correctly estimated delivery delay for each studied integration bucket (*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.

Area Under the Curve (AUC) is used to evaluate the degree of discrimination achieved by the model (HANLEY; MCNEIL, 1982). For instance, AUC can be used to evaluate how well our models can distinguish between addressed issues that are prevented from integration into one or two releases. The AUC is the area below the curve plotting the true positive rate against false positive rate. The value of AUC 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 AUC value for a given bucket b (*e.g.*, *next*) on a binary basis. In other words, the probabilities of the instances were analyzed as pertaining to a given bucket b or not. For example, when computing the AUC value for the *next* bucket, the AUC value is computed by verifying if an instance belongs to the *next* bucket or not. This process is repeated for each bucket. Therefore, each bucket has its own AUC value.

Zero-R models are naïve models that always select the bucket with the highest number of instances. For example, a Zero-R model trained with the Firefox project

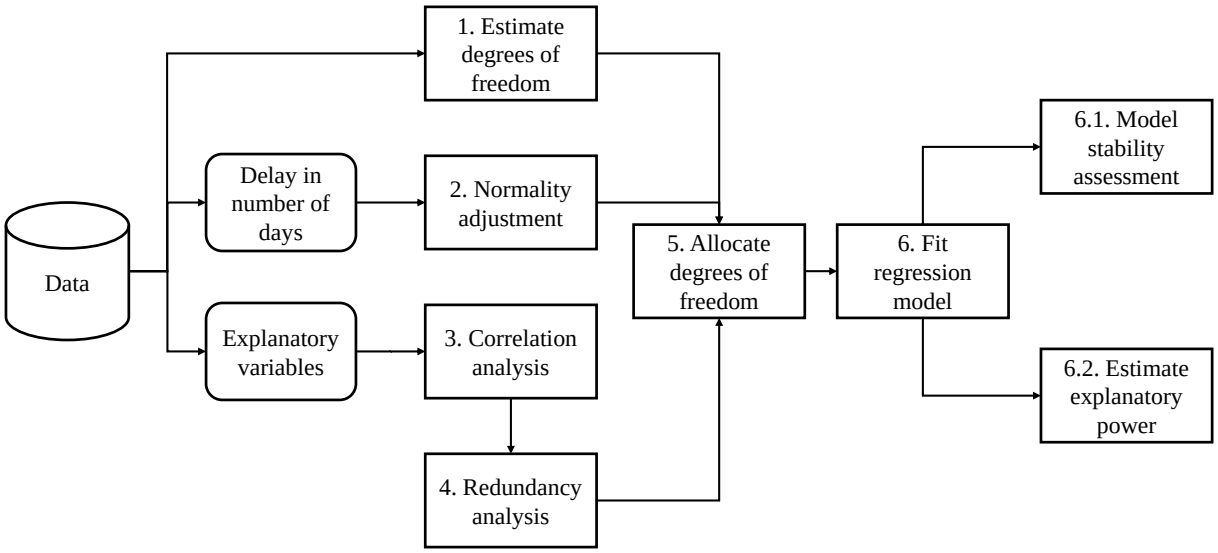


***Figure 12* – Training regression models.** We follow the guidelines that are provided by Harrell Jr. (HARRELL, 2001) to train regression models, which involves nine activities, from data collection to model validation. The results of Steps 6.2 and is presented in RQ4.

data would always select *after-2* as the response for each instance.

We also study the delivery delay in terms of number of days (Definition 2). We train linear regression models (using the *ordinary least squares* technique (OLKIN, 2002)) to study delivery delay in terms of days. Linear regression is an approach for modeling relationships between a dependent variable y and one or more explanatory variables x. When a single explanatory variable is used, the approach is called *simple linear regression*, whereas when several explanatory variables are used, the approach is called *multiple linear regression* (FREEDMAN, 2009). Regression models fit a curve of the form $y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n$. The y variable is the response variable (*i.e.*, delivery delay in terms of days in our case), while the set of X variables represent explanatory variables that may share a relationship with y. The set of β coefficients represent weights given by the model to adjust the values of X to better estimate the response y. The set of explanatory variables that we use in our study are the attributes that are outlined in Tables 3, 4 and 5.

We use the guidelines that are provided by Harrell Jr. (HARRELL, 2001) to fit our regression models. Figure 12 provides an overview of our model fitting approach. In Step 1, we compute the budget of degrees of freedom that our data can accommodate while keeping the risk of overfitting low. We compute this budget by using the formula $\frac{n}{15}$ (where n is the number of issues in our dataset). In Step 2, we verify the normality assumption of *ordinary least squares*, *i.e.*, it assumes that the response variable y should follow normal distribution. Through analysis of the delivery delay values (*i.e.*, the y variable), we find that it does not follow a normal distribution, and hence, we apply a

log transformation $[ln(y+1)]$ to mitigate such skewness.

In Step 3, we use a variable clustering analysis (SARLE, 1990) to remove highly correlated variables. For variables within a cluster that have a correlation of $|\rho| > 0.7$, we choose only one of them to include in our models—we choose the variable with the least skewed distribution and that we suspect that shares a stronger relationship with delivery delay. In Step 4, we check the redundancy of the surviving explanatory variables. Redundant variables do not add explanatory power to the models and can distort the relationship between explanatory and response variables. To remove redundant variables we use the `redun` function from the `rms` R package, which fits models to explain each explanatory variable using the other explanatory variables. We then discard those explanatory variables that could be estimated with an $R^2 \geq 0.9$ (the default threshold of the `redun` function).

In the following step (Step 5), we identify which explanatory variables may benefit from a relaxation of the linear relationship with the response variable. To identify such variables, we calculate the Spearman multiple ρ^2 between the response and explanatory variables. We spend more of our budgeted degrees of freedom on the explanatory variables that obtain the higher ρ^2 values.

In Step 6, we fit our regression models. In order to assess the fit of our models (Step 6.1) we use the R^2 metric. The R^2 measures the *"variability explained"* of the dependent variable that is analyzed (STEEL; JAMES, 1960). For instance, a R^2 of 0.4 indicates that 40% of the variability of the dependent variable is being modeled (*"explained"*) by the explanatory variables—the remaining 60% of the variability may be due to external factors that are not being modeled or cannot be controlled.

We also use the *Mean Absolute Error* (MAE) to verify how close are the estimates of our models ($\hat{y}$) to the actual observations (y). Then, we assess the stability of our models by using the *bootstrap-calculated optimism* of the R^2. The *bootstrap-calculated optimism* is computed by fitting models using bootstrap samples of the original data. For each model fit to a bootstrap sample, we subtract the R^2 of such a model from the model fit to the original data. This difference is a measure of the *optimism* in the original model. In this work, we obtain the *bootstrap-calculated optimism* by computing the average *optimism* obtained using 1,000 bootstrap samples. The smaller the *bootstrap-calculated optimism* the more stable are our models (EFRON, 1986).

RQ3: Results for delivery delay in terms of releases

Our explanatory models obtain a median precision of 0.81 to 0.88 and a median recall of 0.29 to 0.92. Figure 13 shows the precision, recall, F-measure, and AUC of our explanatory models. The bar charts show the values that we observe for each bucket.

Table 6 – The precision, recall, F-measure, and AUC values that are obtained for the Eclipse, Firefox, and ArgoUML projects.

Eclipse				
Bucket	**Precision**	**Recall**	**F-measure**	**AUC**
Next	0.95	0.71	0.81	0.84
After-1	0.75	0.89	0.81	0.88
After-2	0.82	0.95	0.88	0.94
After-3-or-more	0.80	0.98	0.88	0.98
Firefox				
Bucket	**Precision**	**Recall**	**F-measure**	**AUC**
Next	0.99	0.26	0.41	0.63
After-1	0.74	0.17	0.28	0.58
After-2	0.92	0.99	0.96	0.61
After-3-or-more	0.81	0.32	0.45	0.66
ArgoUML				
Bucket	**Precision**	**Recall**	**F-measure**	**AUC**
Next	0.96	0.98	0.97	0.93
After-1	0.89	0.87	0.88	0.92
After-2	0.67	0.31	0.42	0.65
After-3-or-more	0.88	0.89	0.88	0.94

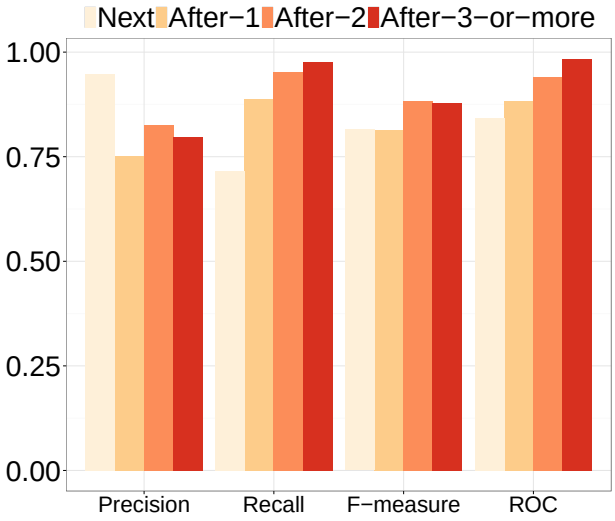
The values of precision, recall, F-measure, and AUC are also shown in Table 6.

The best precision/recall values that we obtain for the Eclipse, Firefox, and ArgoUML projects are related to the *after-2* (F-measure of 0.88), *after-2* (F-measure of 0.96), and *next* (F-measure of 0.97), respectively. However, for buckets with low number of instances, precision/recall values decrease considerably. For instance, the F-measures that are obtained by our models for the Firefox project are considerably low for the *next*, *after-1*, and *after-3-or-more* buckets (0.41, 0.28 and 0.45, respectively).

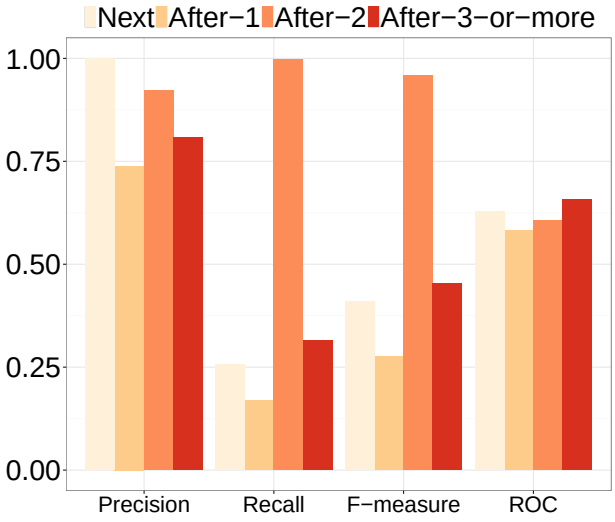
Moreover, our models obtain median AUCs between 0.62 to 0.96, which indicate that our model estimations are better than random guessing (AUC of 0.5). Summarizing the results, our models obtain a median precision of 0.81-0.88 (median) and a median recall of 0.29-0.92. Our models provide a sound starting point for studying the release into which an addressed issue will be integrated.

Our models obtain 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 bucket that contains the majority of the instances. Hence, the recall for the bucket 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.

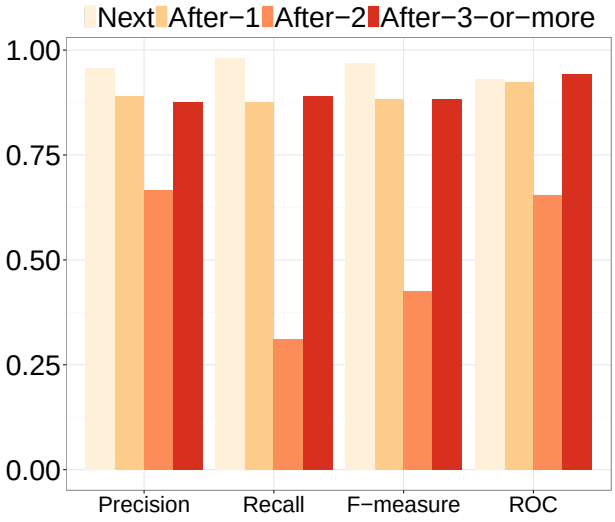
For the Firefox project, Zero-R obtains an F-measure of 0.95 for the *after-2* bucket, whereas our model obtains an F-measure of 0.96 for the same bucket. For the Eclipse project, Zero-R always selects *next* and obtains a F-measure of 0.58, while our



(a) Eclipse



(b) Firefox



(c) ArgoUML

***Figure 13* – Performance of random forest models.** We show the values of Precision, Recall, F-measure, and AUC that are computed using the LOOCV technique.

***Table 7* – Regression results of model fit.** Our explanatory models obtain R^2 values between 0.39 to 0.65 and MAE values between 7.8 to 66 days.

Metric/Project	Eclipse	Firefox	ArgoUML
R^2	0.48	0.39	0.65
MAE (days)	61	7.8	66
Release cycle duration (median in days)	112	42	180
Error ratio ($\frac{MAE}{cycle}$)	0.54	0.18	0.37
Optimism	0.0267	0.0162	0.0035

model obtains an F-measure of 0.81. Finally, for the ArgoUML project, Zero-R always selects *next* with an F-measure of 0.84, whereas our model obtains an F-measure of 0.97. These results show that our models yield better F-measure values than naïve techniques like Zero-R or random guessing (AUC = 0.5) in the majority of cases.

We are able to accurately model how many releases an addressed issue is likely to be prevented from integration. Our models outperform naïve techniques, such as Zero-R and random guessing, obtaining AUC values of 0.62 to 0.96.

RQ3: Results for delivery delay in terms of days

Our explanatory models obtain R^2 values of 0.39-0.65 and MAE values between 7.8 to 67 days. Our models obtain fair R^2 values to model the variability of delivery delay in days in the studied projects. Table 7 shows the R^2 and MAE values that are obtained by each of our regression models. The R^2 values for the Eclipse, Firefox, and ArgoUML projects are of 0.39, 0.48, and 0.65, respectively. Additionally, our regression models can provide fair estimations of delivery delay in days, specially for the Firefox project. For instance, the median interval in days between releases of the Firefox project is 42 days (see Figure 8), while the MAE value for the Firefox project is 7.8 days, which equates to an error ratio of 18% (see Table 7).

Our explanatory models obtain a good stability with bootstrap calculated optimism between 0.0035 to 0.0267 of the R^2 values. We also observe that our regression models are stable. Table 7 shows the *bootstrap-calculated* optimism of the R^2 values of our models. The optimism for the Eclipse, Firefox and ArgoUML projects are 0.0267, 0.0162, and 0.0035, respectively. Such results indicate that our explanatory models are unlikely to be overfitted to our data and that our models are stable enough for us to perform the statistical inferences that follow.

We are able to accurately estimate the delivery delay in terms of number of days. Our models obtain fair R^2 values of 0.39 to 0.65. Our exploratory models are quite stable with a maximum optimism of 0.0267.

3.3.4 RQ4: What are the most influential attributes for modeling delivery delay?

RQ4: Motivation

In RQ3, we found that our models can accurately model the delivery delay of addressed issues. To fit our models, we use attributes that we collect from ITSs and VCSs. As described in Tables 3, 4 and 5, the attributes belong to different families that are related to addressed issues. In RQ4, we investigate which attributes are influential to estimate the delivery delay of addressed issues. We present the approaches and results of RQ4 for each studied type of delivery delay (Definitions 1 and 2).

RQ4: Approach

To identify the most influential attributes for estimating the delivery delay in terms of releases (Definition 1), we compute the *variable importance* score for each attribute of our models. The *variable importance* implementation that we use in our study is available within the *bigrf* R package. This implementation computes the importance score 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 bucket 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 delivery delay. Finally, we use the models with the largest training corpus when performing the LOOCV to compute the variable importance scores.

We perform Step 6.2 of Figure 12 to identify the most influential attributes in our models that we fit to study the delivery delay in terms of number of days (Definition 2). We evaluate the explanatory power of each attribute by using the Wald χ^2 maximum likelihood test (Step 6.2). The larger the χ^2 value, the greater the power that a particular attribute has to model the variability of delivery delay in terms of days. To do so, we use the `anova` function of the `rms` R package.

RQ4: Results for delivery delay in terms of releases

The fixing time per resolver and integration workload attributes are the most influential attributes in our models. Figure 14 shows the variable importance values of the LOOCV of our models. The most influential attribute is the *fixing time per resolver*. The *fixing time per resolver* attribute measures the total time that is spent by each resolver on fixing issues in a release cycle. The second most influential attributes are integration workload attributes (*i.e.*, backlog of issues and backlog of issues per resolver). These integration workload attributes measure the competition of issues that were addressed but not yet integrated into an official release.

Our results suggest that the time that is invested by the resolvers on fixing issues have a strong association with delivery delay. This could be due to resolvers fixing issues more carefully—which would lead to a smoother integration of such issues—or issues that were less complex in overall (*e.g.*, a shorter time was invested), which might simplify the integration process. A deeper analysis of this attribute would be necessary to better understand the exact reasons behind this relationship (*e.g.*, consulting the development team through surveys and interviews).

We also observe that integration workload attributes (*i.e.*, *backlog of issues* and *backlog of issues per resolver*) are the second most influential attributes in the three studied projects. This finding suggests that the integration backlog introduces overhead that may lead to longer delivery delay.

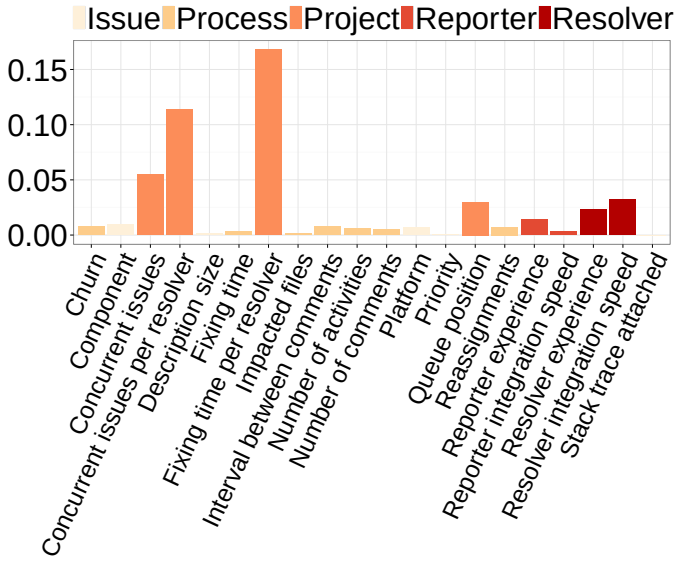
Furthermore, we study the distribution of addressed issues across components in the Firefox project. Figure 15 shows the top seven components of the Firefox project, each having more than 400 addressed issues. We analyze the proportion of addressed issues where integration was prevented in the top seven components. Figure 15 shows that, for buckets *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.

Severity and priority have little influence on delivery delay in terms of releases. 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 (TIAN et al., 2015; 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

(a) Eclipse

(b) Firefox



(c) ArgoUML

***Figure 14* – Variable importance scores.** We show the importance scores that are computed for the LOOCV of our models.

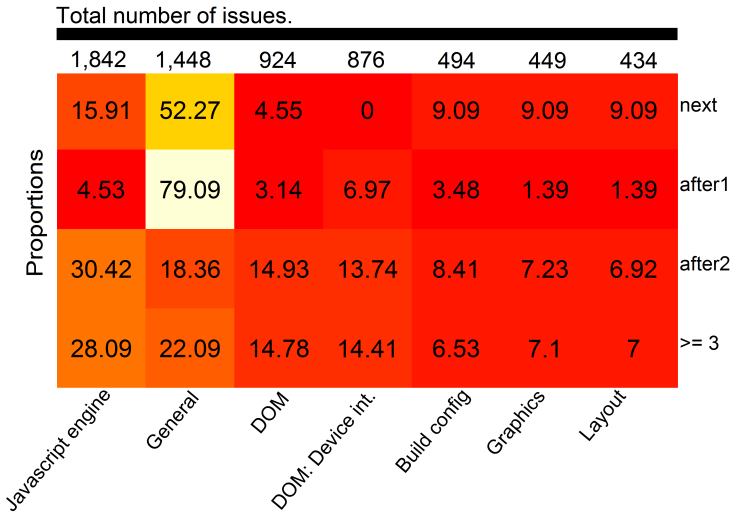


***Figure 15* – The spread of issues among the Firefox components.** The darker the colors, the smaller the proportion of issues that impact that component.

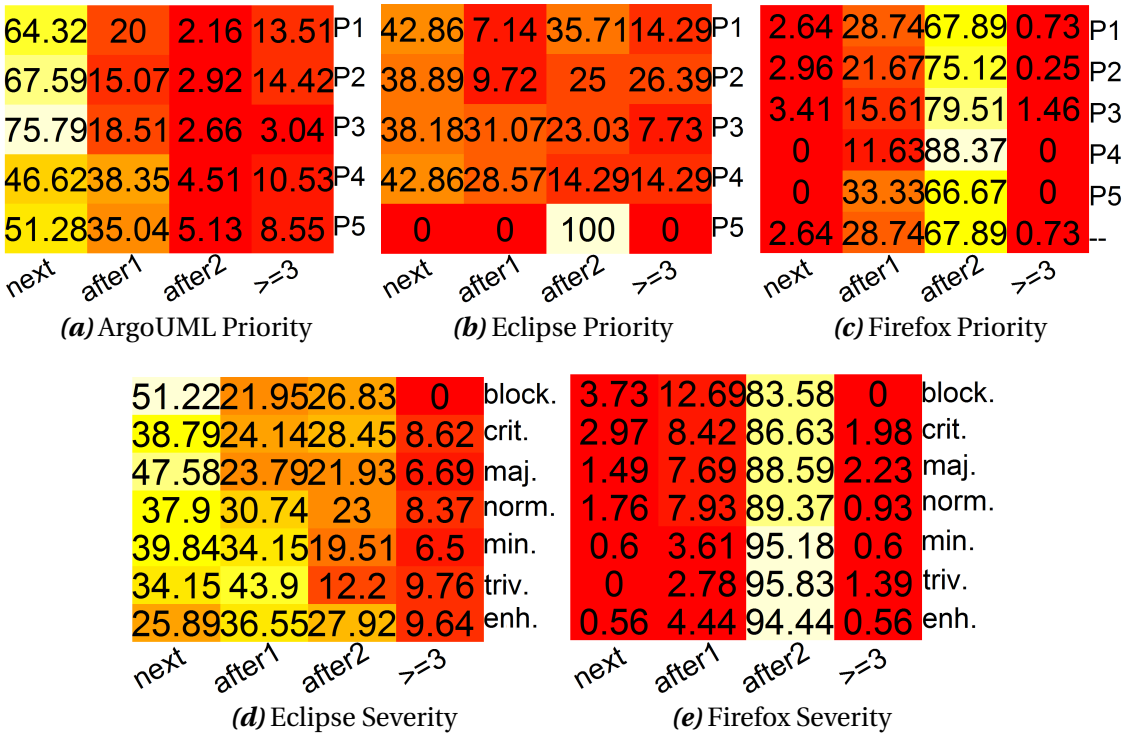
be more influential, since the issues have already been addressed. Even though priority and severity are often left at their default values (see Section 3.2.1), one would expect that the integrators would fast-track the integration of issues for which they care about increasing the levels of severity or priority. For instance, according to the Eclipse project guidelines for filing issue reports, a priority level of P1 is used for serious issues and specifies that the existence of a P1 issue should prevent a release from shipping.[27] Hence, it is surprising that priority and severity play such a small role in determining the release in which an addressed issue will appear. Indeed, Figure 14 shows that the priority and severity metrics obtain low importance scores.

Figure 16 shows the percentage of issues with a given priority (*y-axis*) in a given integration bucket (*x-axis*). The integration of 36% to 97% of priority P1 addressed issues had their integration prevented in at least one release, whereas the integration of 32% to 96% of priority P2 addressed issues were prevented from integration in at least one release.

In the ArgoUML project, while the majority of priority P1 issues (64%) were integrated in the *next* release, 36% of them had their integration prevented in at least one release. For the Firefox project, 97% of the P1 issues and 96% of the *blocker* issues were prevented from integration in at least one release. Finally, for the Eclipse project, 57% of P1 issues and 49% of blocker issues had their integration prevented in at least one release. Hence, our data shows that, in the context of issue integration, the *priority* and *severity* values that are recorded in the ITSs have little influence on delivery delay. Instead, addressed issues might be prioritized by the level of risk that are associated to them.[28] This might explain why the time that is invested on fixing issues during a

[27] <http://wiki.eclipse.org/Development_Resources/HOWTO/Bugzilla_Use>

[28] Two issues from our sample were promoted to stabler release channels due to low associated risk



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

(d) Eclipse Severity ***(e)*** Firefox Severity

***Figure 16* – The percentage of priority and severity levels in each studied bucket of delivery delay.** We expect to see light colour in the upper left corner of these graphs, indicating that high priority/severity issues are integrated rapidly. Surprisingly, we are not seeing such a pattern in our datasets.

release cycle reduces delivery delay—a risk of an addressed issue breaking the code would be smaller when more time is invested at fixing activities.

> *The total time that is invested in fixing issues of a release cycle and integration workload attributes are the most influential attributes in our models. We also find that priority and severity have little influence in estimating delivery delay.*

RQ4: Results for delivery delay in terms of days

Project family attributes, such as the backlog of issues and queue position provide most of the explanatory power of our models. Table 8 shows the explanatory power of each of the attributes of our models. The two most influential attributes for each model are shown in bold. *Queue position, i.e.*, the time at which an issue is addressed is the most influential attribute in all of the models that are fitted to our studied projects. Interestingly, we observe that *resolver integration speed*—the median delivery delay of

<https://bugzilla.mozilla.org/show_bug.cgi?id=724145> and <https://bugzilla.mozilla.org/show_bug.cgi?id=732962>, while another issue was prevented from integration due to code break <https://bugzilla.mozilla.org/show_bug.cgi?id=723793>.

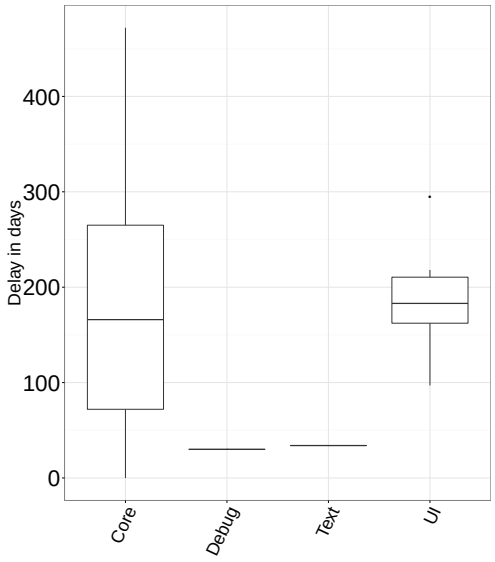
***Table 8* – Explanatory power of attributes.** We present the χ^2 proportion and the degrees of freedom that are spent for each attribute. The χ^2 of the two most influential attributes of each model are in bold.

		Eclipse	Firefox	ArgoUML
Wald χ^2		$1,180$	$8,560$	$2,803$
Budgeted Degrees of Freedom		87	879	102
Degrees of Freedom Spent		24	33	28
Reporter experience	D.F.	1	1	1
	χ^2	4^{***}	≈ 0	1^{**}
Resolver experience	D.F.	1	1	1
	χ^2	12^{***}	$\approx 0^{*}$	≈ 0
Reporter integration speed	D.F.	3	1	2
	χ^2	16^{***}	≈ 0	1^{*}
Resolver integration speed	D.F.	2	1	4
	χ^2	$\mathbf{22}^{***}$	≈ 0	$\mathbf{9}^{***}$
Fixing time	D.F.	2	$\oplus$	1
	χ^2	1^{*}		1^{**}
Severity	D.F.	6	6	$\ominus$
	χ^2	≈ 0	8^{***}	
Priority	D.F.	$\oslash$	5	5
	χ^2		5^{***}	1^{*}
Description size	D.F.	1	1	1
	χ^2	≈ 0	≈ 0	≈ 0
Impacted files	D.F.	1	1	1
	χ^2	≈ 0	$\approx 0^{**}$	$\approx$
Number of comments	D.F.	1	1	1
	χ^2	2^{**}	1^{***}	≈ 0
Reassignments	D.F.	1	1	1
	χ^2	≈ 0	≈ 0	1^{*}
Number of activities	D.F.	1	1	1
	χ^2	≈ 0	≈ 0	≈ 0
Interval between comments	D.F.	1	1	$\oslash$
	χ^2	1^{*}	≈ 0	
Churn	D.F.	1	1	1
	χ^2	≈ 0	≈ 0	1^{**}
Number of concurrent issues	D.F.	$\oslash$	2	$\oslash$
	χ^2		$\mathbf{8}^{***}$	
Number of concurrent issues per resolver	D.F.	1	2	2
	χ^2	7^{***}	2^{***}	$\mathbf{9}^{***}$
Queue position	D.F.	1	4	2
	χ^2	$\mathbf{23}^{***}$	$\mathbf{83}^{***}$	$\mathbf{67}^{***}$
Fixing time per resolver	D.F.	1	2	4
	χ^2	7^{***}	$\approx 0^{**}$	8^{***}

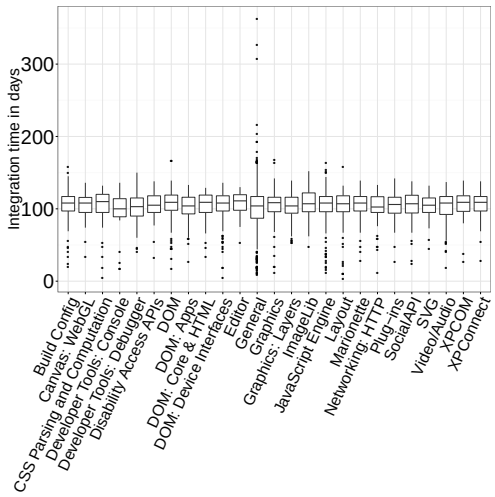
$^{\oslash}$ Discarded during correlation analysis
$^{\oplus}$ Discarded during redundancy analysis
$^{\ominus}$ The variable does not apply to the dataset
* $p < 0.05$
** $p < 0.01$
*** $p < 0.001$

the previously resolved issues of a particular resolver—plays an influential role in our models that are fit for the Eclipse and ArgoUML projects. Moreover, we also observe that integration workload attributes (*i.e.*, *backlog of issues,* and *backlog of issues per resolver*) are very influential in our models that are fit for the Firefox and ArgoUML projects.

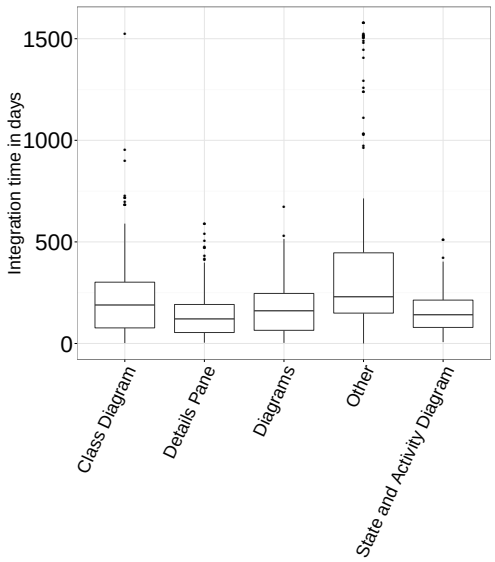
The component to which an issue is addressed has little impact in the delivery delay in terms of days. To demonstrate this, we group each addressed issue according to the



(a) Eclipse



(b) Firefox



(c) ArgoUML

Figure 17 **– Delivery delay per component.** The Figure shows the distributions of delivery delay in terms of days for each component of the studied projects.

components that such an issue modifies. We use components that have at least 100 addressed issues as a threshold for our analysis. We then compare the distribution of delivery delay in terms of days in these components. Figure 17 shows the distributions of delivery delay in terms of days per component. We do not observe a considerable difference between distributions of delivery delay in the ArgoUML or Firefox projects. The distribution of the "Other" component in the ArgoUML project is more skewed, which is suggestive of its generic role—such a component may encompass a more broad spectrum of addressed issues. On the other hand, 99% of the addressed issues in the Eclipse (JDT) project belong to the "Core" component (thus its skewness). Finally, the "Debug" and "Text" Eclipse components contain only one addressed issue each.

The workload in terms of backlog of issues awaiting integration and the integration speed of prior addressed issues of a given resolver play a important role to model delivery delay in terms of days. Moreover, the initial queue position is the most important attribute in all models that we fit to study delivery delay in terms of days.

3.3.5 RQ5: How well can we identify the addressed issues that will suffer from a prolonged delivery delay?

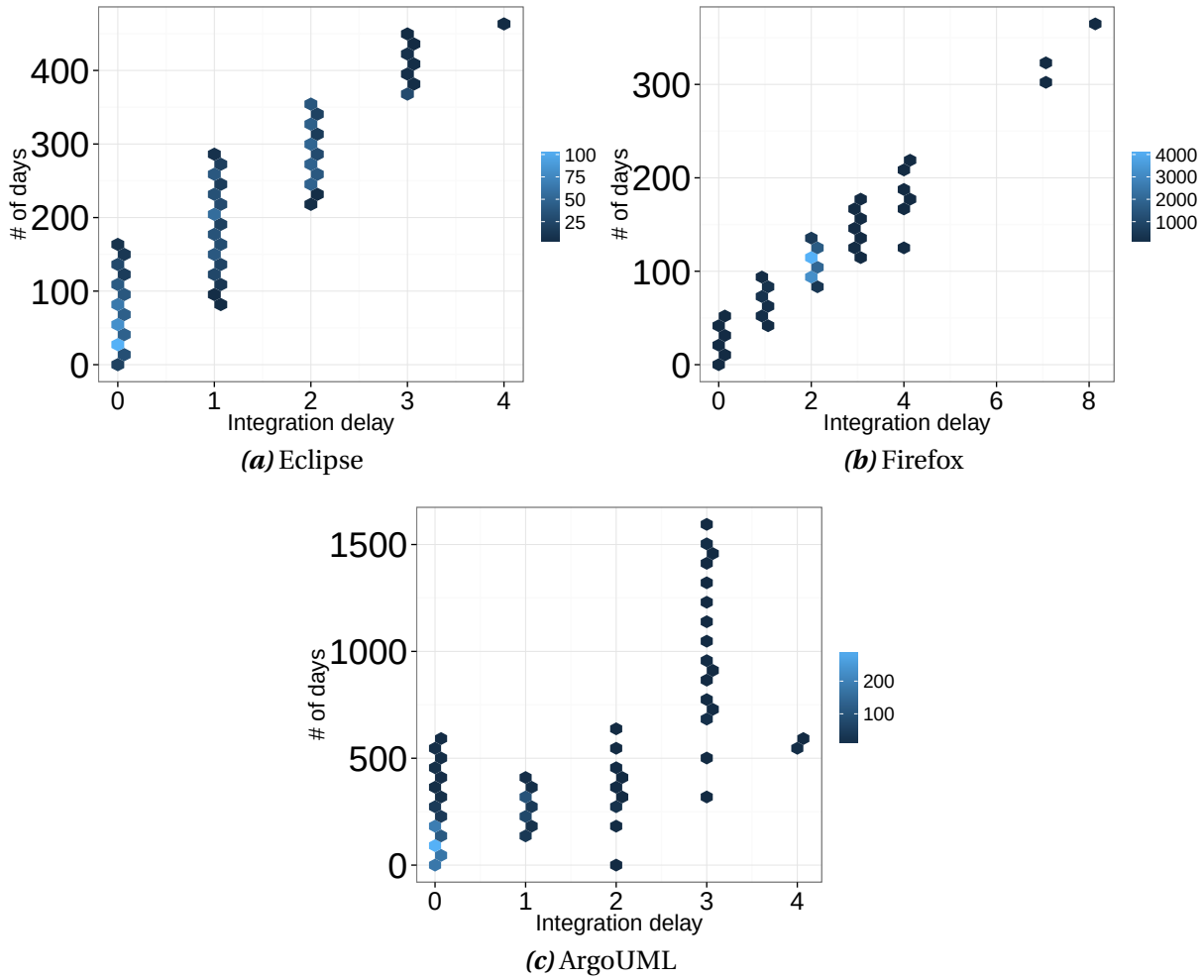
RQ5: Motivation

End users may get frustrated if an addressed issue that s/he is interested has a prolonged delivery delay. Furthermore, if such a delivery delay is unexpected for a particular system (*e.g.*, it is very long), the frustration of users may increase considerably, since they are not used to such a integration time. In RQ5 we investigate if we can accurately identify which addressed issues are likely to have a prolonged delivery delay. This investigation helps us mitigate the problem of prolonged delivery delay of addressed issues.

RQ5: Approach

We calculate prolonged delivery delay (Definition 3) as described in the Step 3 of our data collection process. Indeed, in Figure 7, we observe that the distribution of delivery delay of the Eclipse and ArgoUML projects have more variation than the distribution of the Firefox project.

The hexbin plots of Figure 18 show the relationship between the delivery delay in terms of releases and days. Hexbin plots are scatterplots that represent several data points with hexagon-shaped bins. The lighter the shade of the hexagon, the more



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

***Figure 18* – Relationship between delivery delay in terms of releases and days.** We observe that a longer delivery delay in terms of releases is associated with a longer delivery delay in terms of days.

data points that fall within the bin. Indeed, Figure 18 suggests that the longer the delivery delay in terms of days, the longer is the delivery delay in terms of releases. This tendency is more clear in the Eclipse and Firefox projects. On the other hand, in the ArgoUML project, we observe addressed issues with a longer delivery delay in terms of releases but with a shorter delivery delay in terms of days. For instance, we observe addressed issues with a delivery delay of four releases that have a shorter integration time in terms of days than addressed issues with a delivery delay of three releases. Such behaviour in the ArgoUML project may be explained by the skew in the distance between the releases of this project (*cf.* Figure 8).

Table 9 shows the medians and MADs for each project to identify addressed issues that have a prolonged delivery delay. For instance, an addressed issue have a prolonged delivery delay in the Firefox project when that issue takes more than 123 days to be integrated. Figure 19 shows the proportion of issues that have a prolonged delivery delay per project. We observe that 13%, 12%, and 22% of the addressed issues in the

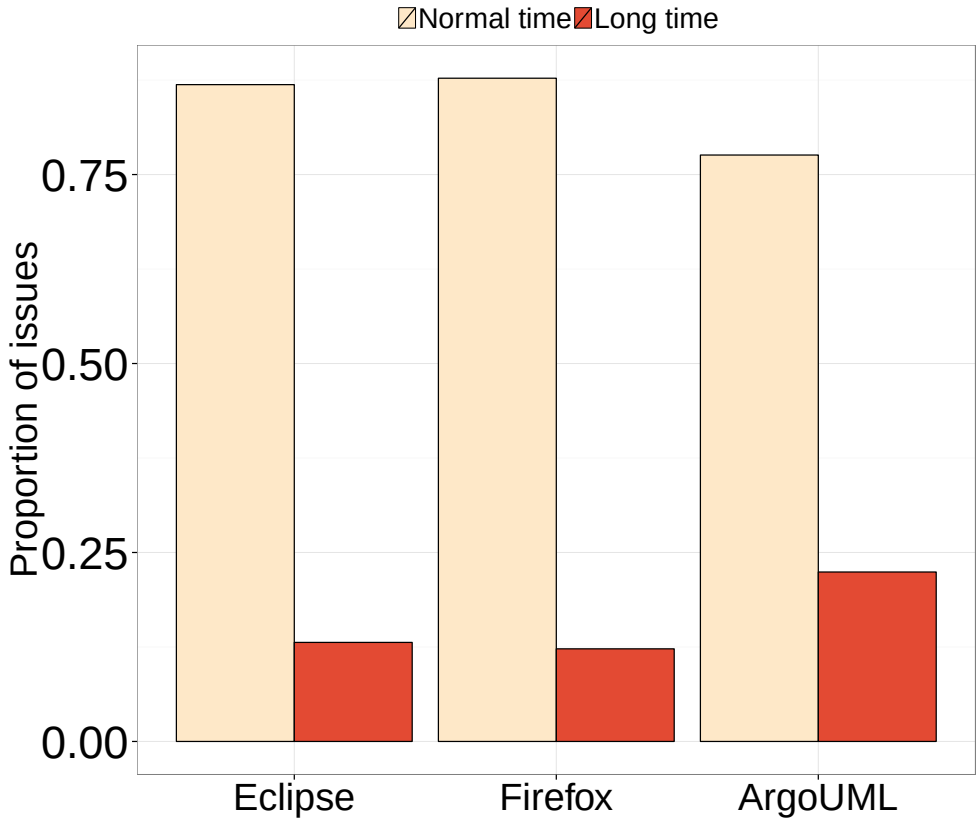


***Figure 19* – Addressed issues that have a prolonged delivery delay.** We present the proportion of addressed issues that have a prolonged delivery delay per project. 13%, 12%, and 22% of the addressed issues of the Eclipse, Firefox, and ArgoUML projects have a prolonged integration time, respectively.

***Table 9* – Prolonged delivery delay thresholds.** We present the median delivery delay in terms of days, the MAD, and the prolonged delivery delay threshold for each project.

	Eclipse	**Firefox**	**ArgoUML**
Median delivery delay	166	107	146
Median absolute deviation	142	16	131
Prolonged delivery delay	> 308	> 123	> 278

Eclipse, Firefox, and ArgoUML projects have a prolonged delivery delay, respectively.

To train our exploratory models, we produce a dichotomous response variable Y, where $Y = 1$ means that an addressed issue has a prolonged delivery delay, while $Y = 0$ means that the delivery delay of that issue is normal. Finally, we train random forest models to study whether a given addressed issue is likely to have a prolonged delivery delay. Similar to RQ2, we evaluate our models using *precision*, *recall*, *F-measure*, and *AUC*.

***Table 10* – Performance of the random forest models.** The table shows the values of Precision, Recall, F-measure, and AUC values that are computed for the LOOCV of our models.

	Eclipse	**Firefox**	**ArgoUML**
Precision	0.97	0.99	0.98
Recall	0.96	0.66	0.80
F-measure	0.96	0.79	0.88
AUC	0.96	0.82	0.89

RQ5: Results

Our models obtain F-measures from 0.79 to 0.96. Table 10 shows the performance of our exploratory models. Our models that we train for the Eclipse project obtain the highest F-measure (0.96). On the other hand, our models trained for the Firefox and ArgoUML projects obtain F-measures of 0.79 and 0.88, respectively. Moreover, our models obtain AUC values of 0.82 to 0.96. Such results suggest that our models vastly outperform naïve models, such as random guessing (AUC value of 0.50).

Our models obtain better F-measure values than Zero-R. For the Eclipse, Firefox, and ArgoUML projects, Zero-R obtain median F-measures of 0.22, 0.22, and 0.36, respectively. Meanwhile, our explanatory models obtain F-measures of 0.96, 0.79, and 0.88, respectively. Again, such results suggest that our models vastly outperform naïve classification techniques.

We are able to accurately identify whether an addressed issue is likely to have a long delivery delay in a given project. Our models outperform naïve techniques, such as Zero-R and random guessing, obtaining AUC values from 0.82 to 0.96 (median).

3.3.6 RQ6: What are the most influential attributes for identifying the issues that will suffer from a prolonged integration time?

RQ6: Motivation

RQ6 shows that we can accurately identify whether an addressed issue is likely to have a prolonged delivery delay. However, it is also important to understand what attributes are more influential when identifying addressed issues with prolonged delivery delay, *i.e.*, from which attributes do our models derive the most explanatory power?

RQ6: Approach

Similar to RQ4, in this research question, we analyze our explanatory models by computing the variable importance score of the attributes.

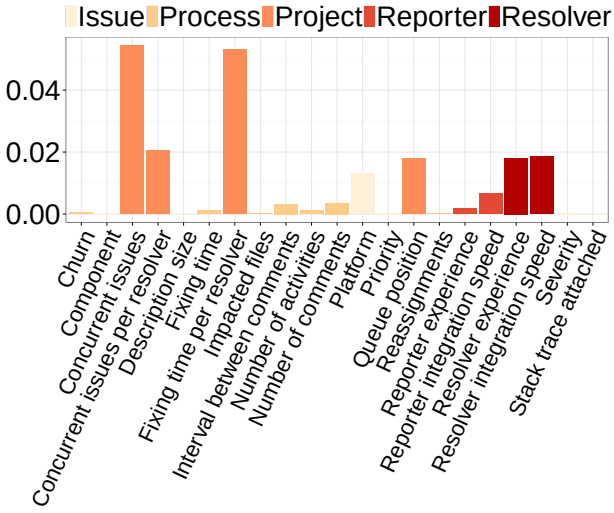
RQ6: Results

Prolonged delivery delay is most consistently associated with attributes of the project family. Figure 20 shows the importance scores that are computed for the LOOCV that we use to evaluate our random forest models. We observe that the attributes that are related to the ***project*** family are the most influential attributes in the projects. The *backlog of issues* is the most influential attribute in our Eclipse models, while *queue position* and *fixing time per resolver* are the most influential attributes in our Firefox and ArgoUML models, respectively. In addition, we observe that attributes that are related to workload, such as the *backlog of issues* and the *backlog of issues per resolver* are at least the third most influential attributes in all of our models. Such results suggest that a *prolonged delivery delay* is associated with project-related attributes and that the amount of addressed issues that are to be integrated also plays a major role to identify a *prolonged delivery delay*.

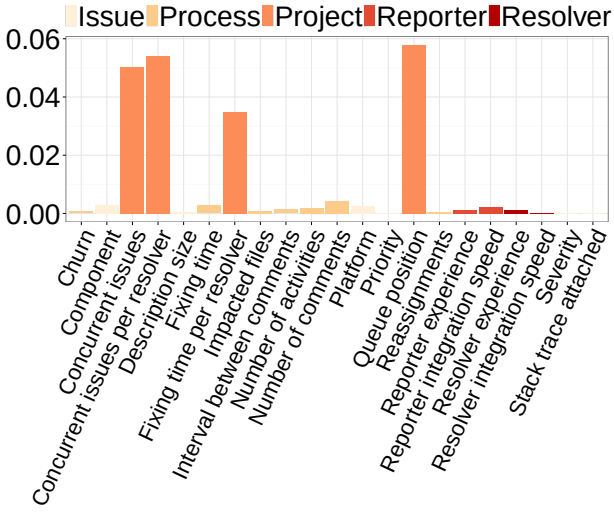
Our explanatory models suggest that prolonged delivery delay is more closely associated with project characteristics, such as the backlog of issues, queue position, and fixing time per resolver. Moreover, the backlog of issues plays an influential role in identifying a prolonged delivery delay in all of the studied projects.

3.4 Discussion

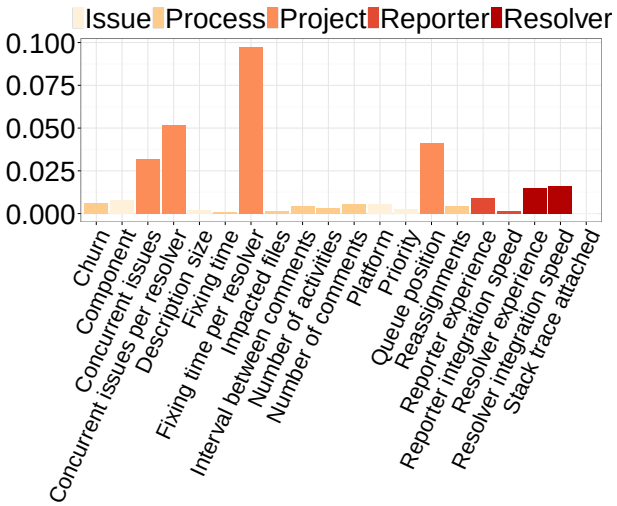
The most important attributes vary as we study different types of delivery delay. While we observe that *fixing time per resolver* is the most influential attribute to model delivery delay in terms of releases, the time at which an issue is addressed (*queue position*) is the most influential attribute to model delivery delay in days. This difference may be explained by what these types of delivery delay highlight. The delivery delay in terms of releases highlights the releases from which the integration of addressed issues is prevented. In this context, the *fixing time per resolver* attribute becomes influential, since it is a measure of the amount of time that was invested by the team to fix issues, which may lead to smoother integration of an issue in the upcoming releases. This smoother integration might be either because issues were addressed more carefully



(a) Eclipse



(b) Firefox



(c) ArgoUML

***Figure 20* – Variable importance scores.** We show the importance scores that are computed for the LOOCV of our models.

or because complex/risky issues had the necessary time to become stable enough to avoid breakage.

On the other hand, delivery delay in terms of days highlights the total time that is required to ship an addressed issue regardless the number of releases that are missed. In this case, the time at which an issue is addressed in the release cycle becomes more influential (*i.e.*, *queue position*). For example, a addressed issue might be shipped faster because it was addressed during a *beta stage* (see , *i.e.*, when the collaborators have to deal with a narrower *backlog of issues* so that fixes can be performed more carefully. The increased focus due to a narrower *backlog of issues* may lead the addressed issue to become easier to integrate in the next release cycle.

Moreover, in the Eclipse project, we observe that the speed at which the prior addressed issues of a particular resolver are integrated influences the integration time of new addressed issues (*resolver integration speed*). This result might be an indicator that resolvers/integrators who are experienced in fixing and integrating fixes for the project may reduce delivery delay.

As for prolonged delivery delays, we observe a similar behaviour in our models that are fit to the ArgoUML and Eclipse projects. The *fixing time per resolver* attribute and attributes that are related to the *backlog of issues* are the most important to identify addressed issues that have a prolonged delivery delay. On the other hand, the *queue position* is the most important attribute to model prolonged delivery delay in the Firefox project. One of the major differences between the former projects (ArgoUML and Eclipse) and the later one (Firefox) is the release cycle strategy that is adopted—ArgoUML and Eclipse use a more traditional release cycle compared to the rapid release cycles that are used in the Firefox project. Nevertheless, more empirical analyses are necessary to investigate if there is a relationship between release cycle strategies and prolonged delivery delay.

The backlog of addressed issues awaiting integration may introduce an overhead that needs to be managed by software teams. We observe that integration workload attributes (*e.g.*, *backlog of issues* and *backlog of issues per resolver*) are influential in all studied types of delivery delay. This finding suggests that the overhead that is introduced by the backlog of addressed issues that are awaiting integration may increase the integration time as a whole.

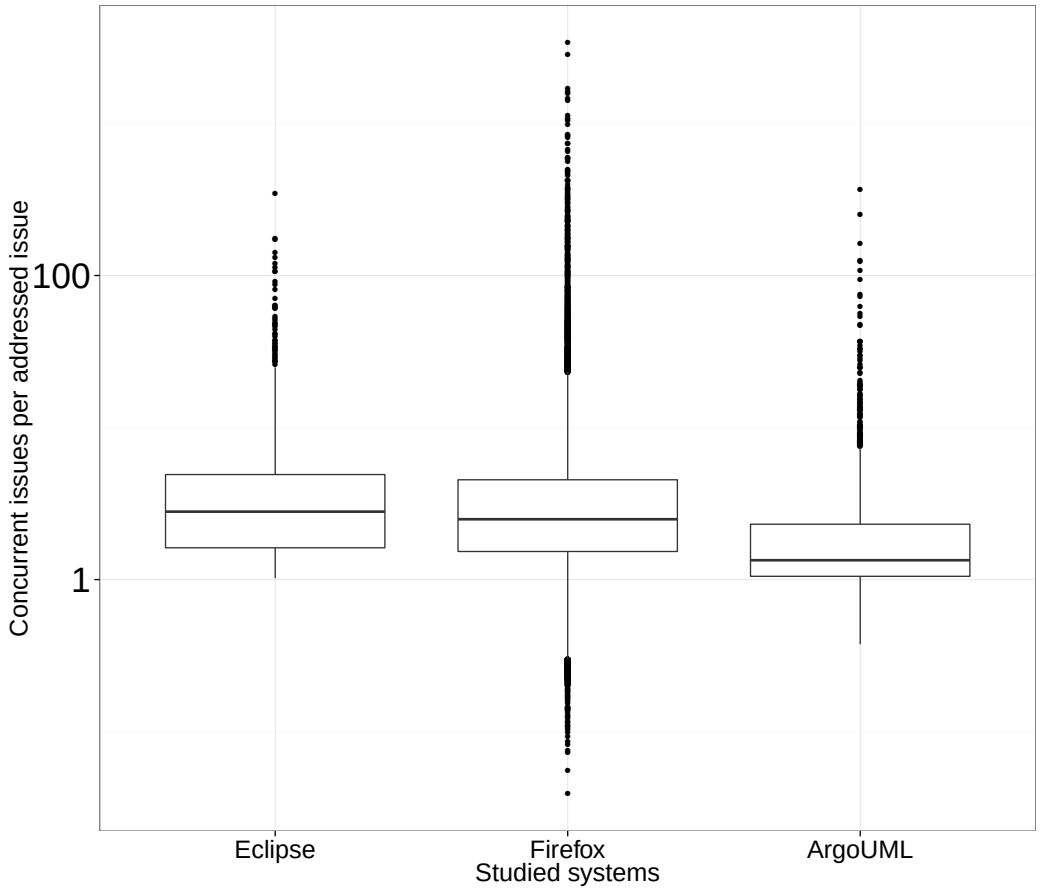


***Figure 21* – Backlog of issues per addressed issue of the current release cycle.** The median number of concurrent fixes per addressed issue for the Eclipse, Firefox, and ArgoUML projects are 3, 2, and 1, respectively.

3.5 Exploratory Data Analysis

3.5.1 Backlog of Issues per Addressed Issue

Since we observe that the integration workload in terms of the number of backlog of issues is an influential attribute in all of the studied projects, we also investigate the competition that is due to issues that are waiting for integration per addressed issue in the release cycle. Figure 21 shows the distributions of the number of competing issues for an addressed issue of a given release cycle. For each addressed issue, a median of three, two, and one other issues are competing for integration in the Eclipse, Firefox, and ArgoUML projects, respectively. It is interesting to note that the distribution of the Firefox project is equivalent to the Eclipse project one, even though the Firefox releases are more frequent. This might suggest an intense period of activity in the Firefox release cycles (high rates of integration and fixing activity).

3.5.2 Practical Suggestions

In our study, we observe that attributes such as: *fixing time per resolver*, *backlog of issues*, *resolver integration speed*, and *queue position* have a considerable impact on the studied types of delivery delay. As such, we suggest that our investigated attributes could be used as a starting point in project management tools to track the integration time of addressed issues. For example, a tool that could automatically track the *backlog of issues* by using the ITS, could raise warnings when the backlog for integration crosses a project-specific threshold. Such a warning could lead to early integration sessions before the official release deadline, and prevent log jams in the integration queue.

Our work suggests that the integration stage is also a bottleneck that has to be managed in a software project. Tracking data and developing tools to reduce delivery delay should also be the target of the practice and research.

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 explanatory and response variables.

The main threat in this regard is the representativeness of the data. Although the Firefox and Eclipse projects 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 notes for version 2.0 but only appears in the release note for version 3.0. Such human errors may introduce noise in our datasets. To explore how correct the release notes are, we draw a random sample of 120 Firefox addressed issues, each one listed in the release notes of versions 17 to 27. We verify the corresponding *tag* that such issues were integrated into in the BETA channel, *i.e.*, the most stable channel of the Firefox

project that lead to the RELEASE channel.[29] Indeed, 94% ($\frac{113}{120}$) addressed issues were integrated into the corresponding tag that lead to the release for which the release notes have listed such issues. This sample can be found on the supplemental material web page.[30]

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

In addition, the way that we segment the response variable of our explanatory models is also subject to bias. For the delivery delay in terms of releases (Definition 1), 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 at least one MAD above the median as a threshold to split the response variable of the prolonged delivery delay (Definition 3) into two categories. A different threshold to split the response variable may yield different results.

Moreover, the attributes that we considered in our explanatory models are not exhaustive. We choose a starting set of attribute families that can be easily computed through publicly available data sources such as ITSs and VCSs. The addition of other attributes would likely improve model performance. For instance, one could study testing or code review effort that was invested on an addressed issue. Nonetheless, our random forest models performed well compared to random guessing and Zero-R models with the current set of attributes and response variable segmentation. With respect to our linear regression models, we base our observations using models that obtain 39% to 65% of variability explained. Although higher R^2 values are usually targeted in research, we provide a sound starting point of regression models for studying delivery delay phenomena—especially in a field that involves human intervention, such as software engineering.

Finally, the main limitation of our statistical models (*i.e.*, random forests and linear regressions) is that we cannot claim a causal relationship between our explanatory variables (*i.e.*, the studied attributes) and delivery delay. Instead, our conclusions are based on associations that are drawn from the average behavior of our studied projects' data.

[29] <https://hg.mozilla.org/releases/mozilla-beta/tags>
[30] <http://sailhome.cs.queensu.ca/replication/integration_delay/>

3.6.3 External Validity

External threats are concerned with our ability to generalize our results. In our work, we investigated only 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 projects. Replication of this work in a large set of projects is required in order to reach more general conclusions.

3.7 Related Work

In this chapter, we present our studies regarding the general delivery delay of addressed issues. Hence, we outline related work about possible delays that can be related to 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 studied attributes, 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 delivery delay of issues that are already addressed rather than the probability to accept a particular patch. Differently from Choetkiertikul *et al.* (MORAKOT et al., 2015a; MORAKOT et al., 2015b), we study the attributes that may induce addressed issues to be prevented from delivery rather than the risk of postponing an upcoming release.

3.8 Conclusions

Once an issue is addressed, what users and code contributors care most about is when the software is going to reflect such an addressed issue, *i.e.*, when such an addressed issue is delivered. However, we observed that the integration of several addressed issues was prevented for a considerable amount of time. In this context, it is

not clear why certain addressed issues take longer to be integrated than others. We performed an empirical study of 20,995 issues from the ArgoUML, Eclipse and Firefox projects. In our study, we:

- despite being addressed well before an upcoming release, 34% to 60% of the addressed issues are not integrated in more than one release in the ArgoUML and Eclipse projects. Furthermore, 98% of the Firefox project issues had their integration delayed by at least one release.

- train random forest models to model the integration time of an addressed issue. Our models obtain a median AUC values between 0.62 to 0.96. Our models outperform baseline random and Zero-R models.

- compute variable importance to understand which attributes are the most important in our random forest models to study delivery delay. Heuristics that estimate the effort that teams invest in fixing issues are the most influential in our models to study delivery delay in terms of number of releases.

- find that, surprisingly, *priority* and *severity* have little impact on our exploratory models for delivery delay. Indeed, 36% to 97% of priority P1 addressed issues were delayed by at least one release.

- find that a shorter delivery delay is associated with fixes that are performed during more controlled stages of a given release cycle.

- observe that the time at which issues are addressed and the resolvers of the issues have great impact on estimating the delivery delay of an addressed issue. Our explanatory models obtain R^2 values between 0.39 to 0.65.

- verify that our models that identify addressed issues that have a prolonged delivery delay outperform random guessing and Zero-R models, obtaining AUC values of 0.82 to 0.96.

- find that the time at which an issue is addressed (queue position), the integration workload (in terms of the backlog of addressed issues), and the heuristics that

estimate the effort that teams invest in fixing issues (fixing time per resolver), are the most influential attributes for issues that have a prolonged delivery delay.

Our work provides 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 have the largest impact on delivery delay. Therefore, our findings highlight the importance of future research and tooling that can 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 Do Rapid Releases Reduce Delivery Delay?

An earlier version of Study 2 appear in the proceedings of the International Conference on Mining Software Repositories (MSR'16) (COSTA et al., 2016).

4.1 Introduction

Within the context of constantly evolving requirements (e.g., in *agile* development), approaches like eXtreme Programming (XP) and Scrum have arisen to foster faster software delivery (BECK, 2000).[31] Those methodologies claim to better embrace a constantly evolving requirements context by shortening release cycles. Indeed, modern release cycles are on the order of days or weeks rather than months or years (BASKERVILLE; PRIES-HEJE, 2004). Such rapid releasing enables faster user feedback and a smoother roadmap for user adoption.

The allure of delivering new features faster has led many large software projects to shift from a more traditional release cycle (*e.g.*, 12-18 months to ship a major release), to shorter release cycles (*e.g.*, weeks). For example, Google Chrome, Mozilla Firefox, and Facebook teams have each adopted shorter release cycles (ADAMS; MCINTOSH, 2016). In this chapter, we use the terms *rapid releases* to describe releases that are shipped using release cycles of weeks or days, and *traditional releases* to describe releases that are shipped using release cycles of months or years.

Prior research has investigated the impact of adopting rapid releases (MÄNTYLÄ et al., 2014; SOUZA; CHAVEZ; BITTENCOURT, 2014; SOUZA; CHAVEZ; BITTENCOURT, 2015; BAYSAL; DAVIS; GODFREY, 2011; KHOMH et al., 2012). For example, Khomh *et al.* (KHOMH et al., 2012) found that bugs that are related to crash reports tend to be fixed more quickly in the rapid Firefox releases than the traditional ones. Mäntylä *et al.* (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.

[31] <http://www.scrumguides.org/>

To the best of our knowledge, little prior research has empirically studied the impact that a shift from a traditional to a rapid release cycle has on the speed of delivering addressed issues. Such an investigation is important to empirically check if adopting a rapid release cycle really does lead to the quicker delivery of addressed issues. In Chapter 3, we study the delay that happens before the delivery of an addressed issue. We found that 98% of the addressed issues in the rapid releases of Firefox were prevented from delivery in at least one release. Such delayed deliveries hint that even though rapid releases are consistently shipped every 6 weeks, they may not be delivering addressed issues as quickly as its proponents purport.

Hence, in this chapter, we compare traditional and rapid release cycles with respect to delivery delay. We perform a quantitative analysis of 72,114 issue reports from the Firefox project (34,673 for traditional releases and 37,441 for rapid releases). These issue reports refer to bugs, enhancements, and new features (ANTONIOL et al., 2008). We address the following RQs:

- ***RQ1: Are addressed issues delivered more quickly in rapid releases?*** Interestingly, we find that although issues are addressed more quickly in rapid releases, they tend to require a longer time to be delivered to users.

- ***RQ2: Why can traditional releases deliver addressed issues more quickly?*** We find that minor-traditional releases (*i.e.*, releases of smaller scope that are shipped after a major version of the software) are a key reason as to why addressed issues tend to be delivered more quickly in traditional releases. In addition, we find that the length of the release cycles are roughly the same between traditional and rapid releases when considering both minor and major releases, with medians of 40 and 42 days, respectively.

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

Chapter Organization. The remainder of this chapter is organized as follows. In Section 4.2, we describe the design of Studies 2. In Section 4.3, we present the obtained results. In Section 4.4, we analyze potential confounding factors. In Section 4.5, we suggest practical guidelines based on our study. Section 4.6 discloses the threats to the

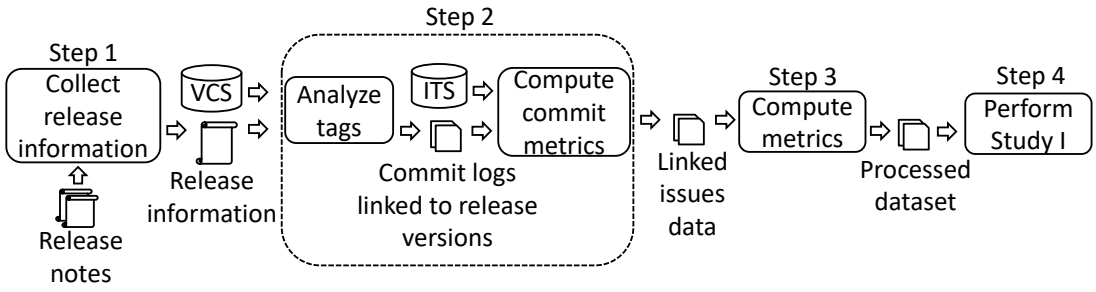


Figure 22 – Overview of the process to construct the dataset that is used in our Study 2.

validity of our study, while we discuss the related work in Section 4.7. Finally, we draw conclusions in Section 4.8.

4.2 Methodology

In Study 2, we set out to comparatively analyze the delivery delay of addressed issues that were shipped in traditional releases versus the ones that were shipped in rapid releases. In this section, we provide information about the subject projects, data collection process, and how we perform the analyses of our study.

4.2.1 Subjects

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

4.2.2 Data Collection

Figure 22 shows an overview of our data collection approach. Each step of the process is described below.

Step 1: Collect release information. We collect the date and version number of each Firefox release (minor and major releases of each release strategy) using the Firefox

Table 11 – The studied traditional and rapid Firefox releases.

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

release history wiki.[32] Table 11 shows: *(i)*the range of versions of releases that we investigate, *(ii)* the investigated time period of each release strategy, and (iii) the number of major and minor studied releases in each release strategy.

Step 2: Link issues to releases. Once we collect the release information, we use the *tags* within the VCS to link issue IDs to releases. First, we analyze the tags that are recorded within the VCS. Since Firefox migrated from CVS to Mercurial during release 3.5, we collect the tags of releases 1.0 to 3.0 from CVS, while we collect the tags of releases 3.5 to 27 from Mercurial.[33,34] By analyzing the tags, we extract the commit logs within each tag. The extracted commit logs are linked to their respective tags. We then parse the commit logs to collect the issue IDs that are being 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 investigated releases should have at least five digits (2,559 issues were discarded).
2. Commit logs that follow the pattern: "Bug <ID> - reftest" or "Bug <ID> - JavaScript Tests", which refer to tests and not bug fixes (269 issues were discarded).
3. Any potential ID that is the name of a file, *e.g.*, "159334.js" (607 issues were discarded).

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

Since the commit logs are linked to VCS tags, we are also able to link the issue IDs found within these commit logs to the releases that correspond to those tags. For example, since we find the fix for issue 529404 in the commit log of tag 3.7a1, we link this issue ID to that release. We also merge together the data of development releases like 3.7a1 into the nearest minor or major release. For example, release 3.7a1 would be merged with release 4.0, since it is the next user-intended release after 3.7a1. In the case that a particular issue is found in the commit logs of multiple releases, we consider that particular issue to pertain to the earliest release that contains the last

[32] <https://en.wikipedia.org/wiki/Firefox_release_history>
[33] <http://cvsbook.red-bean.com/cvsbook.html>
[34] <https://mercurial.selenic.com/>

fix attempt (commit log), since that release is the first one to contain the complete fix for the issue. Finally, we collect the issue report information of each remaining issue (*e.g.*, opening date, fix date, severity, priority, and description) using the ITS. Moreover, since the minor-rapid releases are *off-cycle releases*, in which 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).[35] We add the manually collected addressed issues from ESR releases within the rapid releases data, since they also represent data from a rapid release strategy.

Steps 3 and 4: Compute metrics and perform analyses. We use the data from Step 2 to compute the metrics that we use in our analyses. We select these metrics (which are described in the research approach for Study 3) because we suspect that they share a relationship with delivery delay.

4.3 Results

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

4.3.1 RQ1: Are addressed issues delivered more quickly in rapid releases?

RQ1: Motivation

Since there is a lack of empirical evidence to indicate that rapid release cycles deliver addressed issues more quickly than traditional release cycles, we compare the delivery delay of addressed issues in traditional releases against the delivery delay in rapid releases in RQ1.

RQ1: Approach

Figure 23 shows a simplified life cycle of an issue, which includes the triaging phase (*t1*), the fixing phase (*t2*), and the integration phase (*t3*). We consider the last RESOLVED-FIXED status as the moment at which a particular issue was addressed (the

[35] <https://www.mozilla.org/en-US/firefox/releases/>

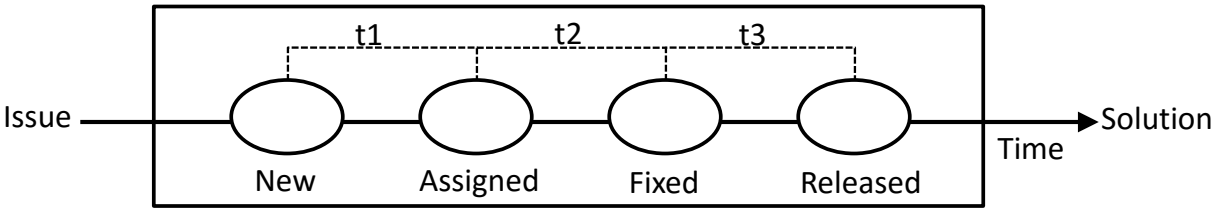


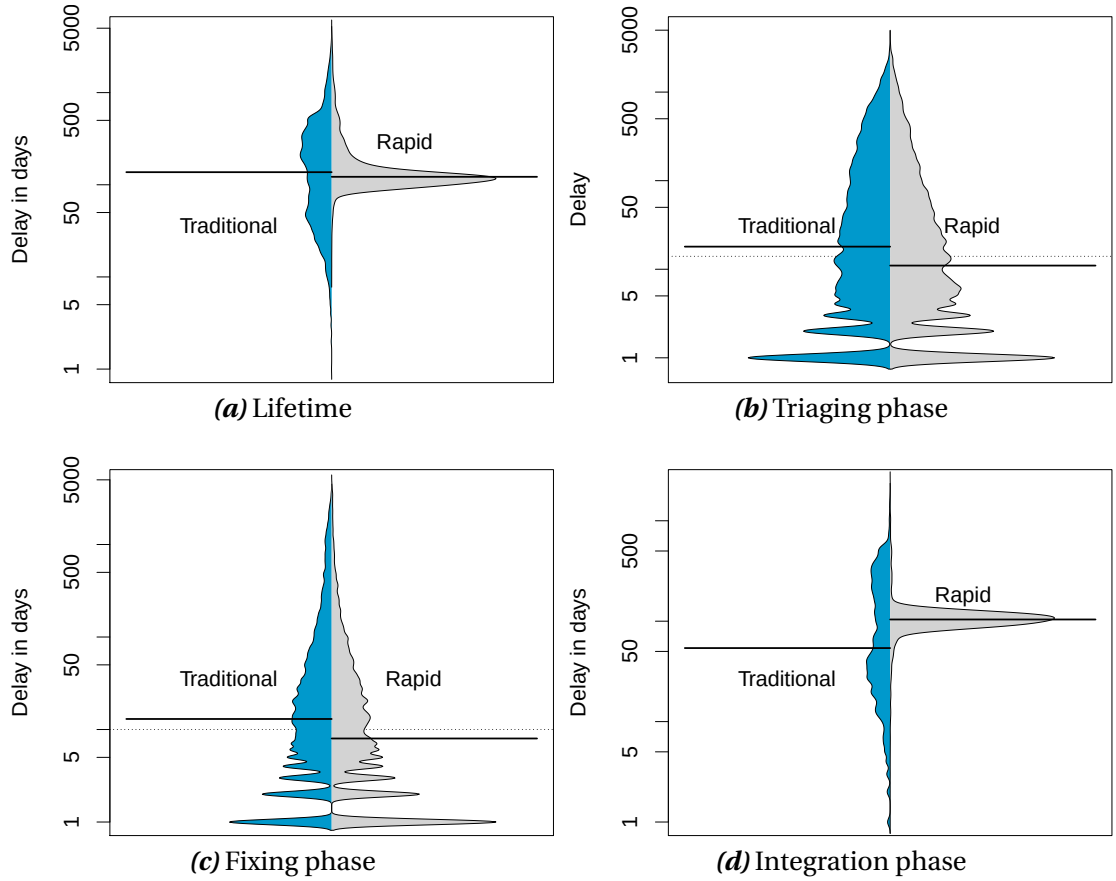
Figure 23 – A simplified life cycle of an issue.

fixed state in Figure 23). The *lifetime* of an issue is composed of all three phases (from *new* to *released*). For RQ1, we first observe the lifetime of the issues of traditional and rapid releases. Next, we look at the time span of the *triaging*, *fixing*, and *integration* phases within the lifetime of an issue.

We use beanplots (KAMPSTRA et al., 2008) to compare the distributions of our data. The vertical curves of beanplots summarize and compare the distributions of different datasets (see Figure 24a). The higher the frequency of data within a particular value, the thicker the bean is plotted at that particular value on the y axis. We also use Mann-Whitney-Wilcoxon (MWW) tests (WILKS, 2011) and Cliff's delta effect-size measures (CLIFF, 1993). MWW tests are non-parametric tests of the *null hypothesis* that two distributions come from the same population ($\alpha = 0.05$). On the other hand, Cliff's delta is a non-parametric effect-size measure to verify the difference in magnitude of one distribution compared to another distribution. The higher the value of the Cliff's delta, the greater the difference of values between distributions. For instance, if we obtain a significant p value but a small Cliff's delta, this means that although two distributions do not come from the same population their difference is not that large. A positive Cliff's delta indicates how much larger the values of the first distribution are, while a negative Cliff's delta indicates the inverse. Finally, we use the *Median Absolute Deviation* (MAD) (HOWELL, 2005; LEYS et al., 2013) as a measure of the variation of our distributions. The MAD is the median of the *absolute deviations* from one distribution's median. The higher the MAD, the greater is the variation of a distribution with respect to its median.

RQ1: Results

Observation 1—There is no significant difference between traditional and rapid releases regarding issue lifetime. Figure 24a shows the distributions of the lifetime of the issues in traditional and rapid releases. We observe a $p < 1.03e^{-14}$ but a $negligible$ difference between the distributions (*delta* $= 0.03$). We also observe that traditional releases have a greater MAD (154 days) than rapid releases (29 days), which indicates that rapid releases are more consistent with respect to the lifetime of the issues. Our results indicate that the difference in the issues' lifetime between traditional and rapid



(a) Lifetime ***(b)*** Triaging phase

(c) Fixing phase ***(d)*** Integration phase

Figure 24 – Time spans of the phases involved in the lifetime of an issue.

releases is not as obvious as one might expect. We then look at the triaging, fixing, and integration time spans to better understand the differences between traditional and rapid releases.

Observation 2—Addressed issues are triaged and fixed more quickly in rapid releases, but tend to wait for a longer time before being delivered. Figures 24b , 24c, and 24d show the triaging, fixing, and integration time spans, respectively. We observe that addressed issues take a median time of 54 days to be integrated into traditional releases, while taking 104 days (median) to be integrated into rapid releases. We observe a $p < 2.2e^{-16}$ with a $small$ effect-size ($delta = -0.25$).

Regarding fixing time span, an issue takes 6 days (median) to be fixed in rapid releases, and 9 days (median) in traditional releases. These results are statistically significant $p < 2.2e^{-16}$, but there is only a $negligible$ difference between distributions ($delta = 0.13$).

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 $p < 2.2e^{-16}$ but $negligible$ difference (*delta* $= 0.11$) between traditional and rapid releases regarding triaging time. The median triaging time for rapid and traditional releases are 11 and 18 days, respectively.

When we consider both pre-integration phases together (triaging $t1$ plus fixing $t2$ in Figure 23), we observe that an issue takes 11 days (median) to triage and address in rapid releases, while it takes 19 days (median) in traditional releases. We observe a $p < 2.2e^{-16}$ with a $small$ effect-size ($delta = 0.15$). Our results suggest that even though issues have shorter pre-integration phases in rapid releases, they remain "on the shelf" for a longer time on average.

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

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

4.3.2 RQ2: Why can traditional releases deliver addressed issues more quickly?

RQ2: Motivation

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

RQ2: Approach

In RQ2, we group traditional and rapid releases into major and minor releases and study their delivery delays. As in RQ1, we also use beanplots (KAMPSTRA et al., 2008), MWW tests (WILKS, 2011), and Cliff's delta effect-size measures (CLIFF, 1993) to perform our comparisons.

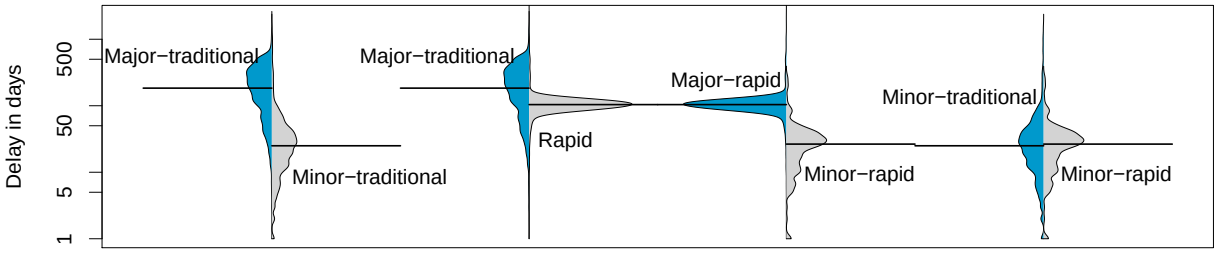


Figure 25 – Distributions of delivery delay of addressed issues grouped by minor and major releases.

RQ2: Results

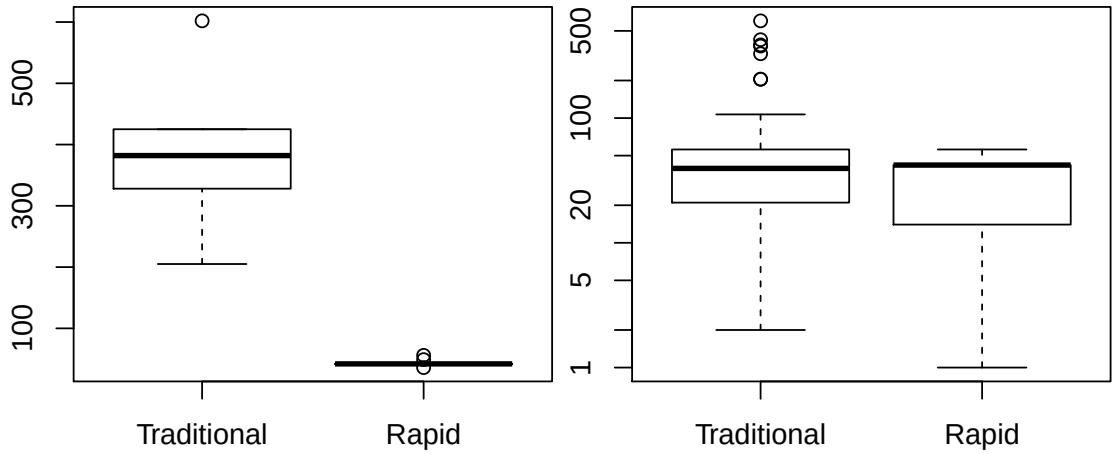
Observation 3—Minor-traditional releases tend to have less delivery delay than major/minor-rapid releases. Figure 25 shows the distributions of delivery delay grouped by (1) *major-traditional vs. minor-traditional*, (2) *major-traditional vs. rapid*, (3) *major-rapid vs. minor-rapid*, and (4) *minor-traditional vs. minor-rapid*. In the comparison of *major-traditional vs. minor-traditional*, we observe that minor-traditional releases are mainly associated with shorter delivery delay. Furthermore, in the comparison *major-traditional vs. rapid*, rapid releases deliver addressed issues more quickly than major-traditional releases on average ($p < 2.2e^{-16}$ with a $medium$ effect-size, *i.e.*, $delta = 0.40$).

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

Minor-traditional releases have the lowest delivery delay (median of 25 days). This is likely because they are more focused on a particular set of issues that, once addressed, should be released immediately. For example, the release history documentation of Firefox shows that minor releases are usually related to stability and security issues.[1]

Observation 4—When considering both minor and major releases, the time span between traditional and rapid releases are roughly the same. Since we observe that delivery delay is shorter on average in traditional releases, we also investigate the length of the release cycles to better understand our previous results (see Observation 2). Figure 26a shows that, at first glance, one may speculate that rapid releases should deliver addressed issues more quickly because releases are produced more frequently.

[1] <https://www.mozilla.org/en-US/firefox/releases/>



(a) Only Major ***(b)*** Major and Minor

Figure 26 – Release frequency (in days). The outliers in figure (b) represent the major-traditional releases.

However, if we consider both major and minor releases—as shown in Figure 26b—we observe that both release strategies deliver releases at roughly the same rate on average (median of 40 and 42 days for traditional and rapid releases, respectively).

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

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

RQ3: Motivation

In RQ1 and RQ2, we study the differences between rapid and traditional releases with respect to delivery delay. We find that although issues tend to be addressed more quickly in rapid releases, they tend to wait longer to be delivered. We also find that the use of minor releases is a key reason as to why traditional releases may deliver addressed issues more quickly. In RQ3, we investigate what are the characteristics of each release strategy that are associated with delivery delays. This important investigation sheds light on what may generate delivery delays in each release strategy, so that projects are aware of the characteristics of rapid releases versus traditional releases before choosing to adopt one of these release strategies.

Table 12 – Metrics that are used in our explanatory models (Reporter, Resolver, and Issue families).

Family	Metrics	Value	Definition (d)|Rationale (r)
Reporter	Experience	Numeric	**d:** the number of previously delivered 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/her reports and the solution to his/her reports might be delivered more quickly (SHIHAB et al., 2010).
	Reporter integration	Numeric	**d:** The median in days of the previously delivered addressed issues that were reported by a particular reporter.
			r: If a particular reporter usually reports issues that are delivered quickly, his/her future reported issues might be delivered quickly as well.
Resolver	Experience	Numeric	**d:** the number of previously delivered addressed issues that were addressed by the resolver of a particular addressed issue. We consider the collaborator that changed the status of an issue to RESOLVED-FIXED as the resolver of that issue.
			r: The greater the experience of the resolver, the greater the likelihood that his/her code will be delivered faster (SHIHAB et al., 2010).
	Resolver integration	Numeric	**d:** The median in days of the previously delivered addressed issues that were addressed by a particular resolver.
			r: If a particular resolver usually address issues that are delivered quickly, his/her future addressed issues might be delivered quickly as well.
Issue	Stack trace attached	Boolean	**d:** We verify if the issue report has a stack trace attached in its description.
			r: A stack trace attached may provide useful information regarding the cause of the issue, which may quicken the integration of the 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 delivered 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 priority levels (*e.g.*, P1) might be delivered faster than other issues.
			r: Higher priority issues will likely be delivered 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.

RQ3: Approach

For RQ3, we build explanatory models (*i.e.*, logistic regression models) for the traditional and rapid releases data using the metrics that are presented in Tables 12, 13,

Table 13 – Metrics that are used in our explanatory models (Project family).

Family	Metrics	Value	Definition (d)|Rationale (r)
Project	Queue rank	Numeric	**d:** A rank number that represents the moment at which 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 a rank of 1, while the last addressed issue has a rank of 500.
			r: An issue with a high *queue rank* is a recently addressed issue. An addressed issue might be delivered faster/slower depending of its rank.
	Cycle queue rank	Numeric	**d:** A rank number that represents the moment at which 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 one has a rank of 300.
			r: An issue with a high *cycle queue rank* is a recently addressed issue compared to the others of the same release cycle. An issue addressed close to the upcoming release might be delivered 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 delivered 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 delivered faster.

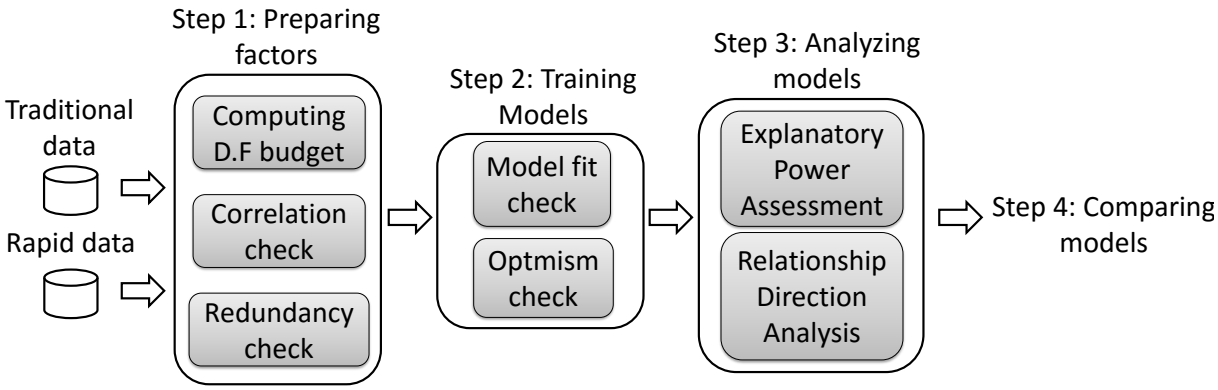


Figure 27 – Overview of the process that we use to build our explanatory models.

and 14. We model our response variable Y as $Y = 1$ for addressed issues that are delayed, *i.e.*, had their delivery prevented in at least one release(COSTA et al., 2014) and $Y = 0$ otherwise. Hence, our models are intended to explain why a given addressed issue has a delayed delivery (*i.e.*, $Y = 1$).

We follow the guidelines of Harrell Jr. (HARRELL, 2001) for building explanatory regression models. Figure 27 provides an overview of the process that we use to build

Table 14 – Metrics that are used in our explanatory models (Process family).

Family	Metrics	Value	Definition (d)|Rationale (r)
Process	Number of Impacted Files	Numeric	**d:** The number of files that are linked to an issue report.
			r: A delivery 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 date when the issue was triaged 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 delivered faster.
	Number of activities	Numeric	**d:** An activity is an entry in the issue's history.
			r: A high number of activities might indicate that much work was required to address the issue, which may impact the integration of the issue into a release (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 delivery 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 an issue may be delivered faster.
	Number of tosses	Numeric	**d:** The number of times that the assignee has changed.
			r: Changes in the issue assignee might indicate that more than one developer have worked on the issue. Such issues may be more difficult to integrate, since different expertise from different developers might be required (JEONG; KIM; ZIMMERMANN, 2009; JIANG; ADAMS; GERMAN, 2013).

our models. First, we estimate the budget of degrees of freedom that we can spend on our models while having a low risk of overfitting (*i.e.*, producing a model that is too specific to the training data to be useful when applied to other unseen data). Second, we check for metrics that are highly correlated using Spearman rank correlation tests (ρ) and we perform a redundancy analysis to remove any redundant metrics before building our explanatory models.

We then assess the fit of our models using the ROC area and the Brier score. The ROC area is used to evaluate the degree of discrimination that is achieved by a model. The ROC values range between 0 (worst) and 1 (best). An area greater than 0.5 indicates

that the explanatory model outperforms naïve random guessing models. The Brier score is used to evaluate the accuracy of probabilistic predictions. This score measures the mean squared difference between the probability of delay assigned by our models for a particular issue I and the actual outcome of I (*i.e.*, whether I is actually delayed or not). Hence, the lower the Brier score, the more accurate the probabilities that are produced by a model.

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

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

We also plot nomograms (IASONOS et al., 2008; HARRELL, 2001) to evaluate the impact of the metrics in our models. Nomograms are user-friendly charts that visually represent explanatory models. For instance, Figure 29 shows the nomogram of the model that we fit for the rapid release data. The higher the number of points that are assigned to an explanatory metric on the x axis (*e.g.*, 100 points are assigned to *comments* in rapid releases), the larger the effect of that metric in the explanatory model. We compare which metrics are more important in both traditional and rapid releases in order to better understand the differences between these release strategies.

RQ3: Results

Observation 5—Our models achieve a Brier score of 0.05-0.16 and ROC areas of 0.81-0.83. The models that we fit to traditional releases achieve a Brier score of 0.16 and an ROC area of 0.83, while the models that we fit to the rapid release data achieve a Brier score of 0.05 and an ROC area of 0.81. Our models outperform naïve approaches

Table 15 – Overview of the regression model fits. The χ^2 of each metric is shown as the proportion in relation to the total χ^2 of the model.

		Traditional releases	Rapid releases
# of instances		$34,673$	$37,441$
Wald χ^2		$4,964$	$2,705$
Budgeted Degrees of Freedom		1033	149
Degrees of Freedom Spent		26	25
Reporter experience	D.F.	1	1
	χ^2	2^{***}	2^{***}
Reporter integration	D.F.	1	1
	χ^2	5^{***}	4^{***}
Resolver Experience	D.F.	1	
	χ^2	1^{***}	$\oslash$
Resolver integration	D.F.	1	1
	χ^2	2^{***}	5^{***}
Fix time	D.F.	1	1
	χ^2	2^{***}	8^{***}
Severity	D.F.	6	6
	χ^2	1^{***}	1^{***}
Priority	D.F.	5	5
	χ^2	1^{***}	≈ 0
Size of description	D.F.	1	1
	χ^2	≈ 0	1^{***}
Stack trace attached	D.F.	1	1
	χ^2	≈ 0	≈ 0
Number of files	D.F.	1	1
	χ^2	1^{***}	1^{***}
Number of comments	D.F.	1	1
	χ^2	$\approx 0^{*}$	31^{***}
Number of tossing	D.F.	1	1
	χ^2	$\approx 0^{***}$	≈ 0
Number of activities	D.F.	1	1
	χ^2	1^{***}	3^{***}
Interval of comments	D.F.	$\oslash$	$\oslash$
	χ^2		
Code churn	D.F.	1	1
	χ^2	≈ 0	≈ 0
Queue position	D.F.	1	1
	χ^2	17^{***}	2^{***}
Queue rank	D.F.	1	1
	χ^2	56^{***}	14^{***}
Cycle queue rank	D.F.	1	1
	χ^2	10^{***}	28^{***}
Cycle queue position	D.F.	$\oplus$	$\oslash$
	χ^2		

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

such as random guessing and ZeroR—our ZeroR models achieve ROC areas of 0.5 and Brier scores of 0.06 and 0.45 for rapid and traditional releases, respectively. Moreover, the bootstrap-calculated optimism is less than 0.01 for both the ROC areas and Brier scores of our models. This result shows that our regression models are stable enough to perform the statistical inferences that follow.

Observation 6—Traditional releases prioritize the delivery of backlog issues, while

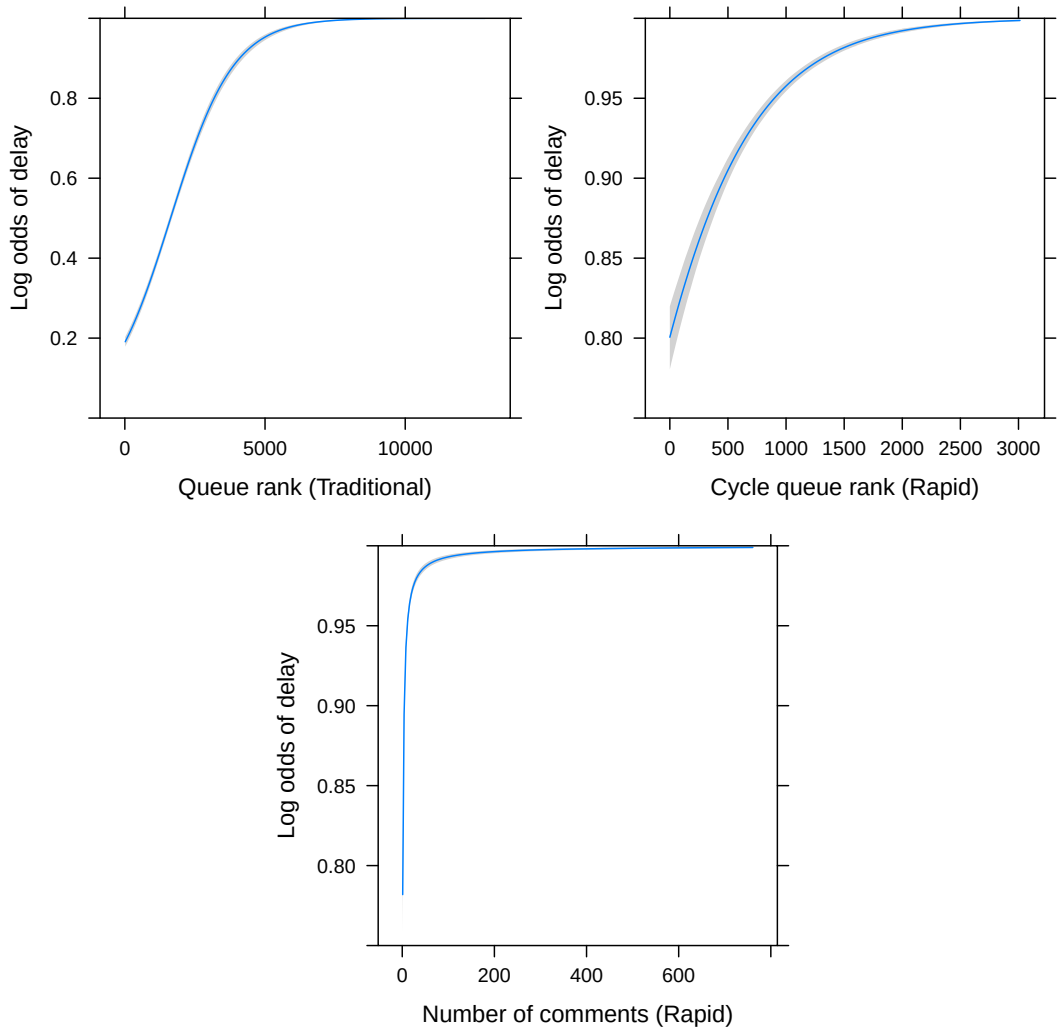


Figure 28 – The relationship between metrics and delivery delay. The blue line shows the values of our model fit, whereas the grey area shows the 95% confidence interval based on models fit to 1,000 bootstrap samples. The parentheses indicate the release strategy to which the metric is related.

rapid releases prioritize the delivery of issues of the current release cycle. Table 15 shows the explanatory power (χ^2) of each metric that we use in our models. The *queue rank* metric is the most important metric in the models that we fit to the traditional release data. Queue rank measures the moment when an issue is addressed in the backlog of the project (see Table 13). Figure 28a shows the relationship that queue rank shares with delivery delay. Our models reveal that the addressed issues in traditional releases have a higher likelihood of being delayed if they are addressed later when compared to other issues in the backlog of the project.

On the other hand, *cycle queue rank* is the second-most important metric in the models that we fit to the rapid release data. Cycle queue rank is the moment when an issue is addressed in a given release cycle. Figure 28b shows the relationship that cycle queue rank shares with delivery delay. Our models reveal that the addressed issues in

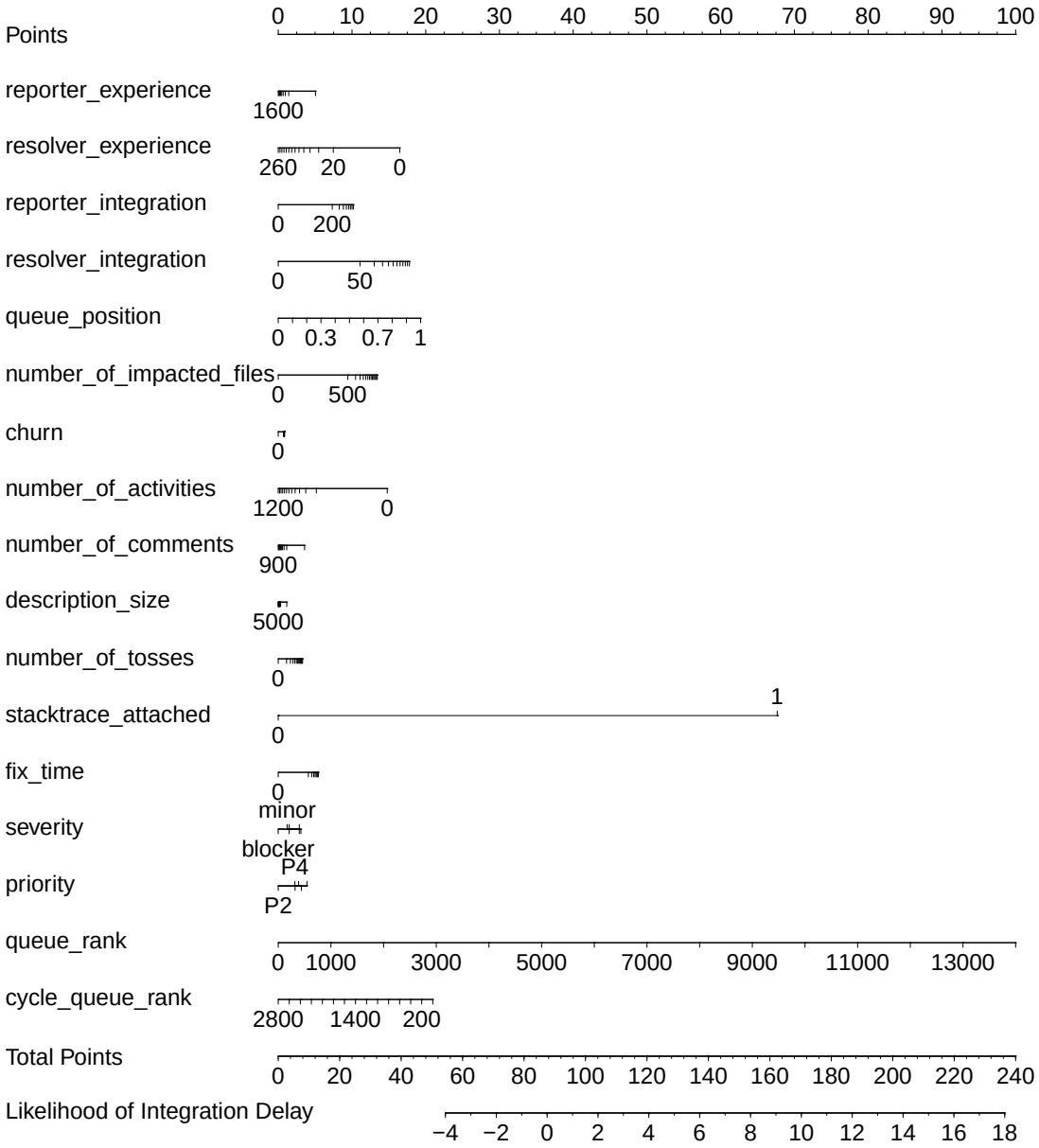


Figure 29 – Nomogram of our explanatory models for the traditional release cycle.

rapid releases have a higher likelihood of being delayed if they were addressed later than other addressed issues in the *current release cycle*. Interestingly, we observe that the most important metric in our rapid release models is the *number of comments*. Figure 28c shows the relationship that the *number of comments* shares with delivery delay. We observe that the greater the number of comments of an addressed issue, the greater the likelihood of delivery delay. This result corroborates the intuition that a lengthy discussion might be indicative of a complex issue, which may be more likely to be delayed.

Moreover, Figures 29 and 30 show the estimated effect of our metrics using nomograms (IASONOS et al., 2008). Indeed, our nomograms reiterate the large impact of *number of comments* (100 points) and *cycle queue rank* (84 points) in rapid releases,

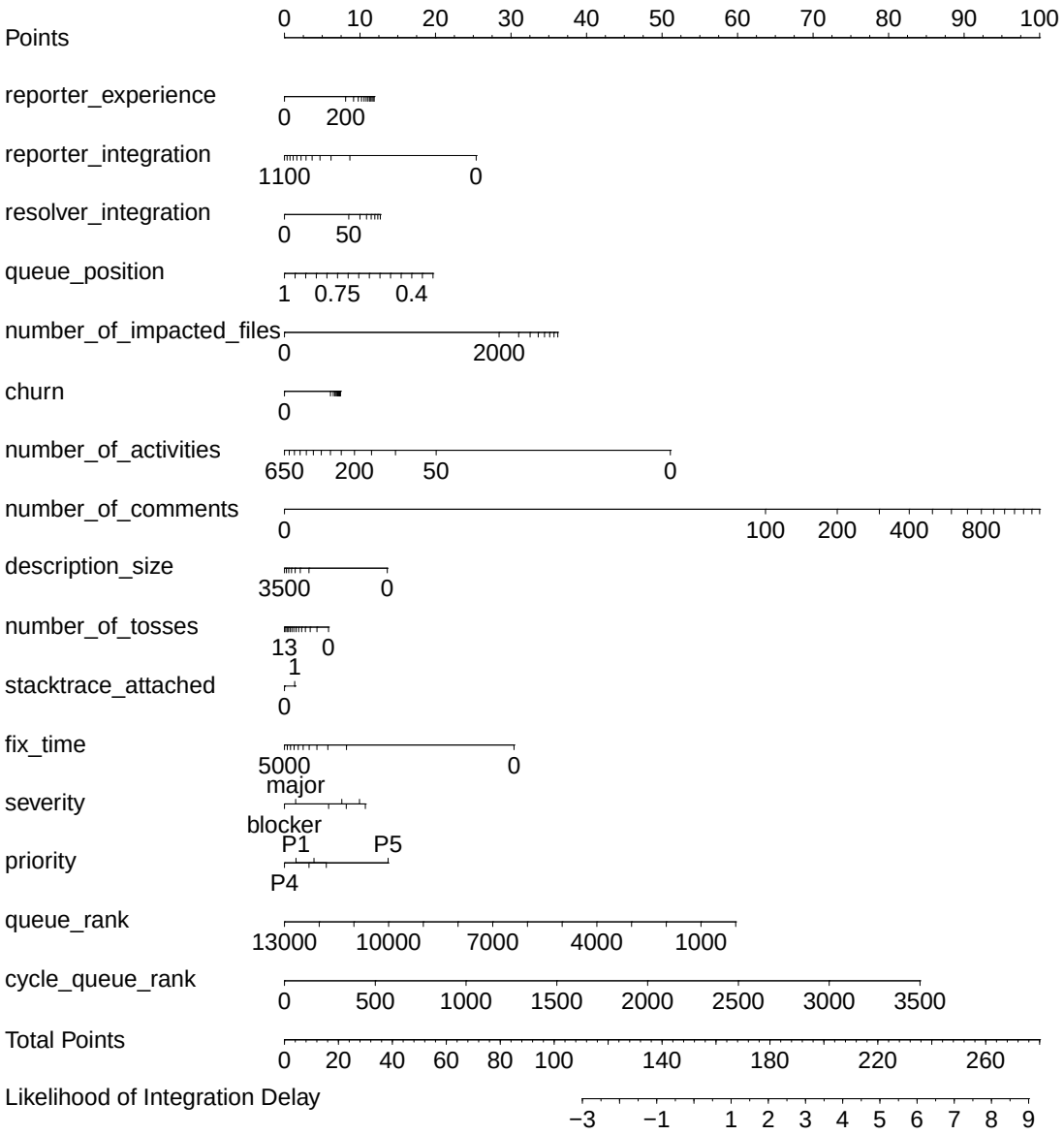
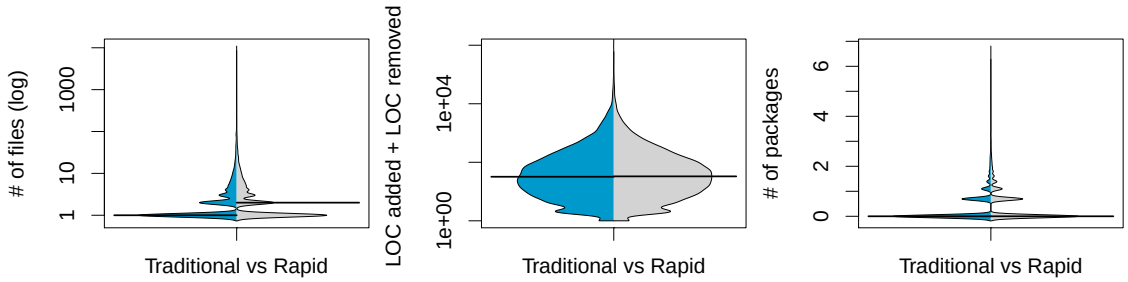


Figure 30 – Nomogram of our explanatory models for the rapid release cycle.

and the large impact of *queue rank* (100 points) in traditional releases. We also observe that *stack trace attached* has a large impact on traditional releases (68 points) despite not being a significant contributor to the fit of our models (*cf.* Table 15). The large impact shown in our nomogram for *stack trace attached* is due to the skewness of our data—only 5 instances within the traditional release data have the *stack trace attached* set to true. Thus, *stack trace attached* cannot significantly contribute to the overall fit of our models.

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



(a) Files ***(b)*** LOC ***(c)*** Packages

Figure 31 – Size of the addressed issues in the traditional and rapid release data.

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

4.4 Analysis of Potential Confounding Factors

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

Observation 8—The delivery delay of addressed issues is unlikely to be related to the size of an issue. One may suspect that the difference in delivery delay between release strategies may be due to the *size of an issue*. We use the *number of files*, *LOC*, and *number of packages* that were involved in the fix of an issue to measure the *size of an issue*. Figure 31 shows the distributions of the metrics that measure the *size of an issue*. We observe that the difference between distributions of *LOC* is statistically insignificant ($p = 0.86$). As for the *number of files* and the *number of packages*, although we observe significant differences (p values of 0.014 and $< 2.2e^{-16}$, respectively), effect-sizes are $negligible$ ($delta = -0.05$ and $delta = -0.07$, respectively).

Observation 9—The difference between traditional and rapid releases is unlikely to be related to the differences between enhancements and bug fixes. We also investigate if the observed difference in the delivery delay between traditional and rapid releases is related to the type of addressed issues. For example, rapid releases could be delivering more enhancements, which likely require additional integration time in order to ensure

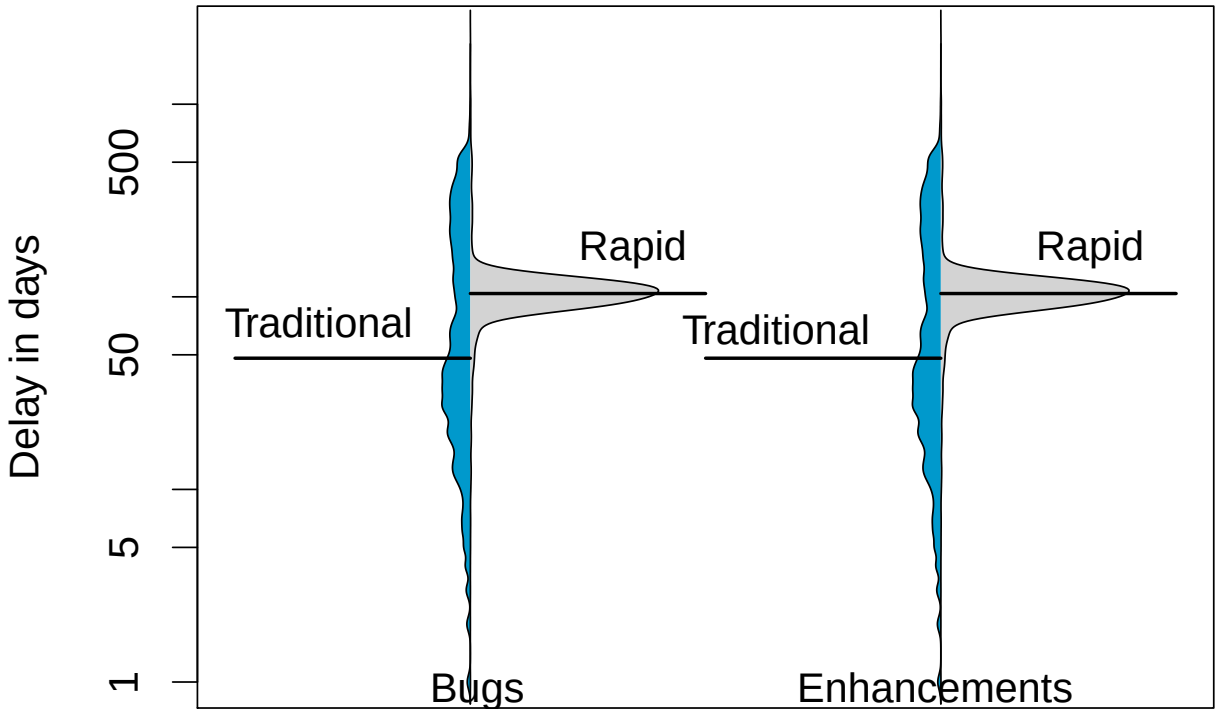


Figure 32 – We group the addressed issues into "bugs" and "enhancements" by using the *severity* field. However, the difference in the delivery delay between release strategies is unlikely to be related with the type of the issue.

that the new content is of sufficient quality. Figure 32 shows the distributions of delays among release strategies grouped by bug fixes and enhancements. We observe no clear distinction between delivery delay and the type of addressed issues that is being delivered.

4.5 Practical Suggestions

In this section, we outline suggestions for practitioners and researchers based on the results of our empirical study.

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

Consistency of delivering addressed issues. Our empirical study suggests that rapid releases can improve the consistency of the time to deliver addressed issues (see Observation 2). A more consistent delivery of addressed issues can be an advantage for the software organization, since end users would have a better understanding as to when issues will be addressed and delivered.

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

4.6 Threats to Validity

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

Internal Validity. Internal threats to validity are concerned with the ability to draw conclusions from the relation between the independent and dependent variables. The way that we link issue IDs to releases may not represent the total addressed issues per release. For example, although Firefox developers record issue IDs in commit logs, we do not know how many of the addressed issues were not recorded in the VCS. Techniques that improve the quality of the link between issue reports and commit logs could prove useful for future work.

In Section 4.4, we compare the delivery delay between rapid and traditional releases by grouping the issues as bug fixes or enhancements. We use the *severity* field of the issue reports to perform this grouping. We are aware that the severity field is noisy (HERRAIZ et al., 2008; TIAN et al., 2015) (*i.e.*, many values represent the same level of importance). Still, the *enhancement* severity is one of the significantly different values of severity according to previous research (HERRAIZ et al., 2008). We also use the number of files, packages, and the LOC to approximate the *size* of an issue. Although these are widely used metrics to measure the size of a change, we are aware that this might not represent the true complexity of the fix of an issue.

External Validity. External threats are concerned with our ability to generalize our results. We study Firefox releases, since the Firefox project shifted from a traditional release cycle to a rapid release cycle. Although we control for variations using the same studied project in different time periods, we are not able to generalize our conclusions to other projects that adopt a traditional/rapid release cycle. In order to mitigate the external threat, we also perform a qualitative study of the Firefox, ArgoUML, and Eclipse projects (see Chapter 5). By adding two other projects in our qualitative analysis, and analyzing new sources of data (our participants), we are able to gain insights from other subjects and better understand why delivery delay occurs. Still, we cannot claim that our results are generalizable to other software projects that are not studied in this work. Hence, replication of this work using other projects is required in order to reach more general conclusions.

4.7 Related Work

In this section, we situate our study with respect to prior work on the impact of adopting rapid release cycles.

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; CHAVEZ; BITTENCOURT, 2015) found that backout of commits increased when rapid releases were adopted. However, they note that such results may be due to changes in the development process rather than the rapid release cycle—the backout culture was not widely adopted during the traditional Firefox releases. We also investigate the shift from traditional releases to rapid releases in this paper. However, we analyze delivery delay rather than quality and quality assurance activities.

It is not clear yet if rapid releases lead to a faster rate of bugs fixes. Baysal *et al.* (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 that are associated with crash reports are fixed faster in rapid Firefox releases when compared to Firefox traditional releases. However, fewer bugs are fixed in rapid releases, proportionally. Our study corroborates that issues are addressed more quickly in rapid release cycles, but tend to wait longer to be delivered to the end users.

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

4.8 Conclusions

In this chapter, we perform a study about the impact that rapid release cycles have on delivery delays. In our study, we analyze a total of 72,114 issue reports of 111 traditional releases and 73 rapid releases of the Firefox project. We obtain the following results:

- Although issues tend to be addressed more quickly in the rapid release cycle, addressed issues tend to be integrated into consumer-visible releases more quickly in the traditional release cycle. However, a rapid release cycle may improve the consistency of the delivery rate of addressed issues (see Observation 2).
- We observe that the faster delivery of addressed issues in the traditional releases is partly due to minor-traditional releases. One suggestion for practitioners is that more effort should be invested in accommodating minor releases to issues that are urgent without compromising the quality of the other releases that are being shipped (see Observation 3).
- The triaging time of issues is not significantly different among the traditional and rapid releases (see Observation 2).
- The total time that is spent from the issue report date to its integration into a release is not significantly different between traditional and rapid releases (see Observation 1).
- In traditional releases, addressed issues are less likely to be delayed if they are addressed recently in the backlog. On the other hand, in rapid releases, addressed issues are less likely to be delayed if they are addressed recently in the current release cycle (see Observation 6).

This study is the first to empirically check whether rapid releases ship addressed issues more quickly than traditional releases. Our findings suggest that there is no silver

bullet to deliver addressed issues more quickly. Instead, rapid releases may increase the consistency of the delivery rate of addressed issues to end users.

5 Why do Delivery Delays Occur?

5.1 Introduction

In our prior studies (Chapter 3 and 4), we quantitatively investigate the delivery delay of addressed issues. We perform several statistical analyses based on the data that is publicly available on the ITSs and VCSs of our subject projects. However, to reach deeper knowledge as to why delivery delays occur, we survey 37 participants from the ArgoUML, Firefox, and Eclipse projects about the delivery delay of addressed issues. We also perform follow up interviews with four participants to get deeper insights about the responses that we receive. In Study 3, we qualitatively investigate the delivery delay of addressed issues to *(i)* reach additional insights that could not be possible by only performing quantitative analysis and *(ii)* verify to which extent our participants agree with our findings from the quantitative studies. More specifically, we address the following research questions:

- ***RQ4: What are developers' perceptions as to why delivery delays occur?*** The perceived reasons for the delivery delay of addressed issues are related to decision making, team collaboration, and risk management activities. Moreover, delivery delay will likely lead to user/developer frustration according to our participants.

- ***RQ5: What are developers' perceptions of shifting to a rapid release cycle?*** The allure of delivering addressed issues more quickly to users is the most recurrent motivator of switching to a rapid release cycle. Moreover, the allure of improving the flexibility and quality of addressed issues is another advantage that are perceived by our participants.

- ***RQ6: To what extent do developers agree with our quantitative findings about delivery delay?*** The dependency of addressed issues on other projects and team workload are the main perceived explanations of our findings about delivery delay in general. Integration rush and increased time spent on polishing addressed issues (during rapid releases) emerge as main explanations as to why traditional releases may achieve shorter delivery delays.

Chapter Organization. The remainder of this chapter is organized as follows. In Section 5.2, we describe the design of our study. In Section 5.3, we present the obtained

results. In Section 5.4, we disclose the threats to the validity of our study, while we discuss the related work in Section 5.5. Finally, we draw conclusions in Section 5.6.

5.2 Methodology

In this study, we qualitatively analyze the delivery delay phenomena by surveying and interviewing the team members of our subject projects. In this section, we describe the subject projects how we collect and analyze our data.

5.2.1 Subjects

We analyze the Firefox, ArgoUML, and Eclipse (JDT) projects. We naturally choose these projects, since this qualitative study is intended to complement our prior quantitative analyses that we performed in those projects. We provide a brief description of each subject project below (we have already provided a detailed description in Section 3.2.1).

ArgoUML is an open source UML modeling tool. ArgoUML provides support for all of the UML 1.4 diagrams. At the time that we perform this study, ArgoUML was downloaded 80,000 times worldwide.[36] ArgoUML uses the IssueZilla ITS to record its issue reports.[37]

Eclipse is a popular *Integrated Development Environment* (IDE) that is famous for its support for the Java programming language.[38] We study the *Java Development Tools* (JDT) project of the Eclipse Foundation.[39] The JDT project provides the Java perspective for the Eclipse IDE, which includes a number of views, editors, wizards, and builders.

ArgoUML and Eclipse (JDT) adopt a traditional release cycle when compared to the Firefox project. For instance, the median duration of release cycles that we study for the ArgoUML and Eclipse (JDT) projects are 180 and 112 days, respectively (see Chapter 3). While we are able to study the perceived impact of the shift between release strategies when surveying the participants of the Firefox project, we study the opinion of the ArgoUML and Eclipse participants about how that impact would be on their projects.

[36] <http://argouml.tigris.org>
[37] <http://argouml.tigris.org/project_bugs.html>
[38] <https://eclipse.org/>
[39] <https://projects.eclipse.org/projects/eclipse.jdt>

***Table 16* – Survey questions (excerpt).** Each horizontal line indicates a page break.

1. For how long have you been developing software? *(dropdown)* **2.** For how long have you worked in the (Firefox/ArgoUML/Eclipse) project? *(dropdown)* **3.** How would you describe your roles in the software development of the Firefox/ArgoUML/Eclipse project? (e.g., developer, tester, release manager, etc.) *(text box)* **4.** In your opinion, what motivates a development team to shift from a traditional release cycle (e.g., a release every 9 to 18 months) to a rapid release cycle (e.g., a release every 6 weeks)? *(text box)* **5.** In this survey, we consider that an issue is completed when it is implemented and tested, i.e., it is ready to be delivered. Do you remember an issue that the development team completed work on, but was not delivered to end users through the next possible release? Can you tell us what caused the delivery delay of this issue in your opinion? *(text box)* **6.** In your experience, how common are the cases in which completed issues (issues that are implemented and tested) are omitted from the next possible release? **7.** Who decides when a completed issue is delivered into an official release in your team? *(text box)* **8.** In your opinion, when is the delivery of a completed issue to the end user considered to be delayed in your project? *(text box)*
9. In your opinion, is it frustrating to users when a completed issue skips one or more releases? Why? *(text box)* **10.** Is it frustrating for the team members when a completed issue skips one or more releases? Why? *(text box)*
11. Assuming that an issue is completed today (implementation and testing are completed), what reasons can you think of for the issue not to be delivered to end users in the next release? *(text box)*
12. What can team members do to avoid the delivery delay of completed issues? *(text box)* **13.** To what extent do you agree that the characteristics listed in the table below are related to the delivery delay of a completed issue? - The reporter of an addressed issue, the resolver of an addressed issue, the priority level, the severity level, number of comments, number of modified files, number of lines of code, the time at which an issue was addressed during the release cycle. *(5-point Likert scale for each option)*
14-Firefox. Have you worked in both traditional and rapid release cycles of the Firefox project? *(yes/no)* **15-Firefox.** In your opinion, how much impact does a rapid release cycle have on the time to deliver completed issues for end users? *(text box)* **16-Firefox.** Did your project evaluate the shift to rapid release cycles? If so, how? *(text box)* **14-Others.** Do you have experience working on a rapid release cycle in any other project? *(yes/no)* **15-Others.** In your opinion, what would be the impact of shifting to a rapid release cycle (e.g., a release every 6 weeks rather than a release every 9 to 18 months) on the delay to deliver completed issues, in your project? *(text box)* **16-Others.** If your project had shifted from a traditional to a rapid release cycle, how would you evaluate if this shift benefited your project? *(text box)*

5.2.2 Data collection

To collect the data to perform our qualitative study, we design a web-based survey that was sent to 780 participants of the Firefox, Eclipse (JDT), and ArgoUML projects. We sent our survey to 513 Firefox, 184 Eclipse (JDT), and 83 ArgoUML participants. We gather developer e-mails from the respective developer mailing list archives of the subject projects. We consider e-mail addresses from messages that were sent in the past 4 years. To encourage participation, we provided gift cards to a random subset of the respondents who answered all of the questions of our surveys.

Our survey is based on the two major *themes* that are investigated in this thesis. The first *theme* is about delivery delay in general, while the second *theme* is focused

Table 17 – Participant range per subject project.

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

on the impact of switching to a rapid release cycle on the delivery delay (see Figure 1). In Table 16, we highlight a subset of the questions of our survey. Each horizontal line represents a page break in the survey. Our complete surveys are available in Appendices A, B, and C. The first three questions collect demographic information. Questions #5-13 belong to the general delivery delay *theme*, while questions #4, #14-16 belong to the impact of switching to a rapid release cycle *theme*. We placed one question of the second *theme* early in the survey to mitigate bias in the responses about the motivation to switch to a rapid release cycle. Finally, questions #14-16 are different for the Firefox project, since the other projects did not shift from a traditional to a rapid release cycle.

In total, we receive 37 responses (5% response rate), of which 25 responses come from Firefox participants, 9 from Eclipse participants, and 3 from ArgoUML participants. We also conduct follow-up interviews with four of the Firefox participants to gather deeper insights into their responses. Our interviews are semi-structured and our goal was to clarify the responses of our survey and collect more details about specific cases of delivery delays for addressed issues.

5.2.3 Research Approach

Given the exploratory nature of our qualitative analysis, we use methods from *Grounded Theory* (CHARMAZ, 2014). The author of this thesis and co-author #1 independently conduct three sessions of open coding of the responses to open-ended questions (one session for each RQ). In the following, the codes that were generated are shared and merged into a new set of codes. The co-author #2 reviews the set of codes and adds additional entries to the final set of codes. At the end of the process, we achieve 175 unique codes. Finally, we used axial coding to find higher level conceptual themes to answer our RQs.

When reporting the results of RQ4-RQ6, we indicate in superscript the number of participants that mentioned a particular code that emerged during the qualitative analysis. These numbers do not necessarily indicate the importance of a given code, since they were coded based on the received responses rather than scored by participants. Also, we mention quotes from the interviews when necessary to provide more detail about the results and to allow for traceability through links to raw data. Finally,

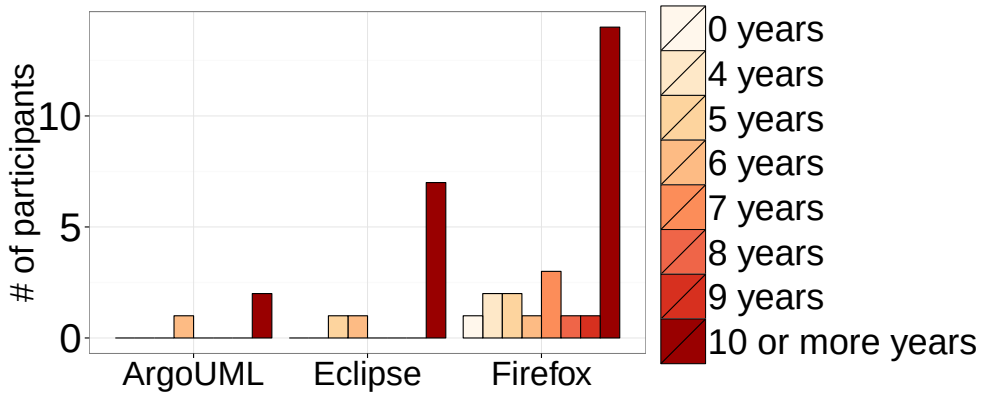


Figure 33 – Software development experience of the participants.

Table 17 shows the IDs of the participants that we use while reporting results.

Finally, we also perform quantitative analyses of the responses to Likert-scale questions. First, we check if the factors that are listed in *question #13* are significantly different using the ranks (responses) that are assigned to each factor. We use a Kruskal Wallis test (KRUSKAL; WALLIS, 1952) to check if there is a statistically significant difference between the ranks assigned to the factors. The Kruskal Wallis test is the non-parametric equivalent of the ANOVA test (FISHER, 1925) to check if there are statistically significant differences when comparing three or more distributions. Since Kruskal Wallis does not indicate which factor has statistically different values with respect to others, we use the Dunn test (DUNN, 1964) to perform specific comparisons. For example, the Dunn test indicates if the ranks that are assigned to the *number of comments* metric are statistically different when compared to the ones that are assigned to the *number of modified files* metric. We use the Bonferroni correction (DUNN, 1961) on the obtained p values to account for the multiple comparisons that we perform between each of the factors that are listed in *question #13*.

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

5.2.4 Exploratory Analysis

We present a exploratory analysis of the data that we collect from the responses of the participants. Figure 33 shows the experience of the participants. We collect this data from *question #1*. The options range from “0 years” to “10 or more years”. We observe that 62% ($\frac{23}{37}$) of the participants have “10 or more years” of software development

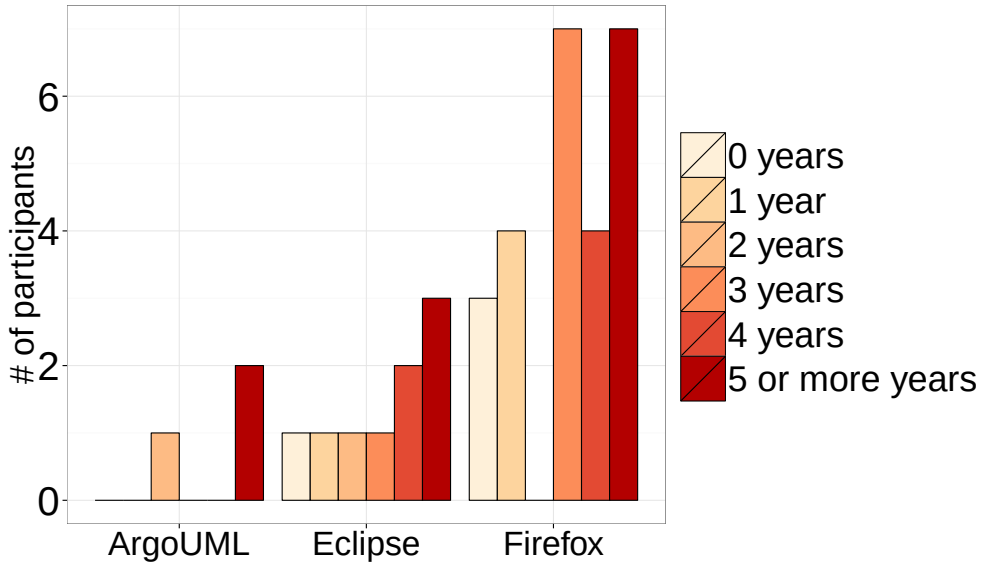


Figure 34 – Development experience of the participants in the respective project.

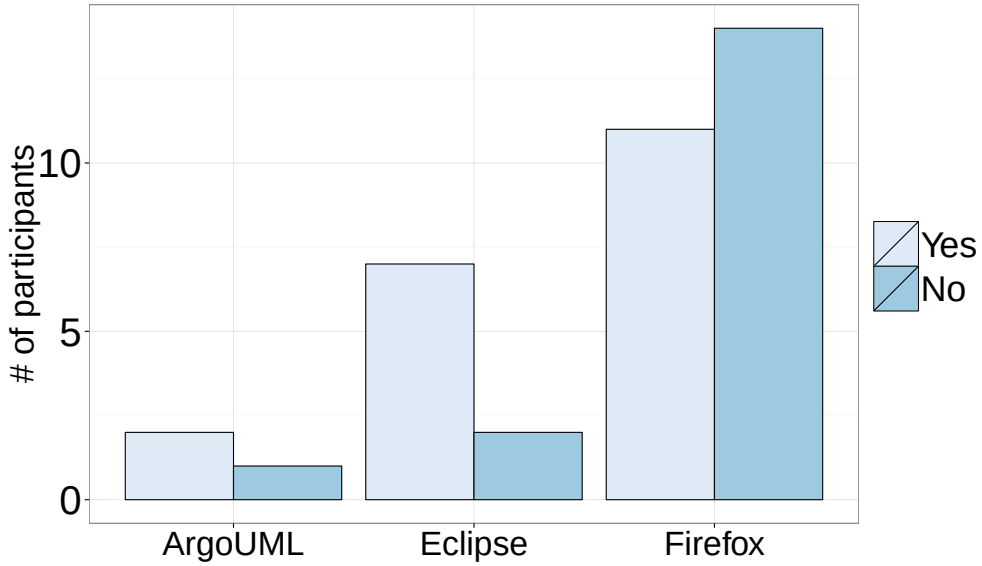


Figure 35 – Experience of the participants with respect to rapid release cycles.

experience. Furthermore, Figure 34 shows the experience of the participants related to the specific project that they are representing. We collect this data from *question #2* and the options range from "0 years" to "5 or more years". 51% ($\frac{19}{37}$) of the participants have 4 or more years of experience. Moreover, Figure 35 shows how many participants have experience in working on rapid release cycles (*question #14*). We note that 57% ($\frac{21}{37}$) of the participants have some experience with rapid release cycles. Figure 36 shows the team roles that the participants classified themselves as (*question #3*). The majority of the participants consider themselves as "developers" and "testers". Since one participant can occupy several roles, the numbers that are shown in Figure 36 represent the frequency that a role was cited rather than the number of participants. Finally, we observe that the majority of the participants perceive delivery delay as an unusual

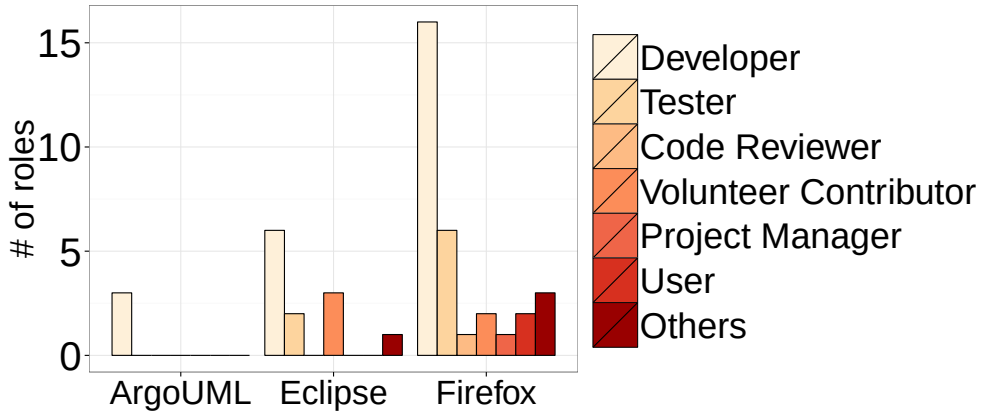


Figure 36 – An overview of the roles of the participants. One participant may have more than one role.

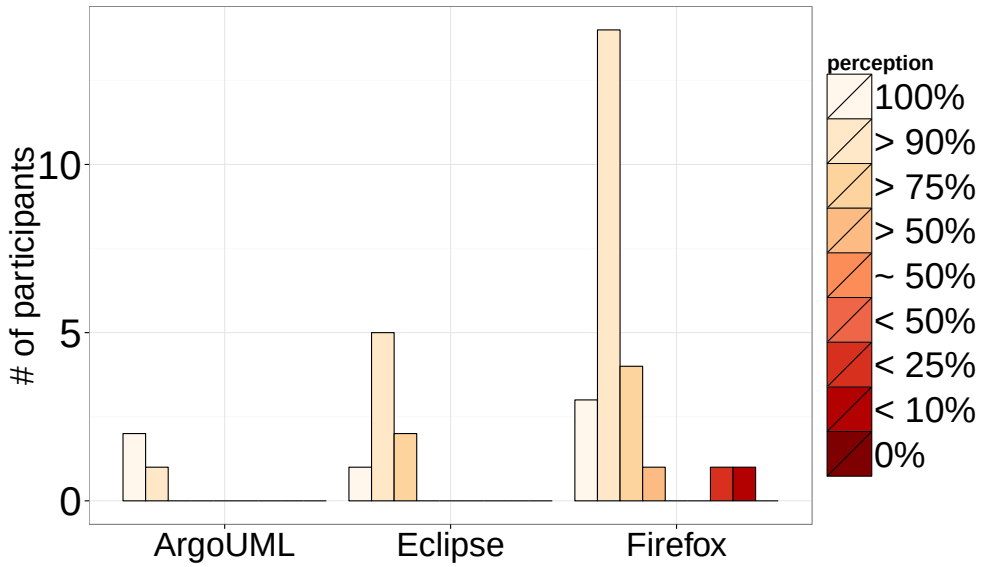


Figure 37 – Participants' perception on how frequent is delivery delay. The data is grouped by proportions of how many addressed issues are included in the next possible release. This data refers to the responses to *question #6*.

event rather than typical (see Figure 37). For instance, 14 of the Firefox participants think that 90% of the issues are included in the next possible release.

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

5.3 Results

We present the motivation, approach and obtained results for each investigated RQ below.

5.3.1 RQ4: What are developers' perceptions as to why integration delays occur?

RQ4: Motivation

To the best of our knowledge, there is no prior work that qualitatively studies delivery delay. Qualitative studies are important to detect phenomena that are difficult to uncover quantitatively. Our goal in this RQ is to better understand *why* delivery delays happen. This investigation is a starting point to reveal new ways of mitigating delivery delays.

RQ4: Results

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

Theme 1—Development activities. The number of tests that should be executed was a recurrent theme among participants. For instance, several participants stated that *additional testing*$^{(12)}$ should be executed in order to avoid delivery delay. *P17* states that the lack of *"actual user testing beyond what QA can provide"* can lead to delivery delay. Additionally, according to *F15*, *"the most common reason is that testing was incomplete"* and according to *P19*, delivery delay may happen because *"testing has been too narrow"*. Finally, *E32* voices concerns about integration testing: *"No integration tests has been done."* Such observations bring us back to a core software engineering problem of when is testing sufficient? (BELLER; GOUSIOS; ZAIDMAN, 2015; ALGHAMDI et al., 2016).

Other recurrent themes that emerged during our qualitative analysis are *workload*$^{(7)}$ and *code review*.$^{(7)}$ For example, *E30* states that *"As the delayed completed issues stack up, they are harder to integrate (the codebase is constantly changing, merge issues might emerge)."* Interestingly, our statistical models in our prior work (see Chapter 3) indicate *workload*$^{(7)}$ as a metric that shares a strong relationship with delivery delay. As for *code review*,$^{(7)}$ the *"Unavailability of the lead/reviewer/[Project Management Committee] (PMC)"* is a reason of delivery delay that is pointed out by *E26*, while *F08* argues that a *"prompt code reviews [may] help"* to avoid delivery

delays (MCINTOSH et al., 2016).

Theme 2—Decision making. Decision making refers to the activities that are not directly related to software construction, but can influence the speed at which software is shipped. For example, how early a codebase should be "frozen"? Which issues should be prioritized? The *timing*(9) and *prioritization*(9) are the recurrent themes in our survey responses. For instance, two of the participants stated that issues can be delayed because they are addressed *"too late in the release cycle"* (*E28*) or because they were addressed in a *"long release cycle."* Also, *F12*'s opinion about how to avoid delivery delay is to *"test [addressed issues] early using real users (e.g., on the pre-release channels)."* Regarding *prioritization,*(9) *E28* argues that team members should *"try to complete most important things early in the release cycle"* to avoid delivery delay. Additionally, *F07* points out how re-prioritization of issues is important: *"[...] prioritizing and re-prioritizing tasks to be sure you are building things on time [...]."*

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

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

Theme 5—Collaboration with other teams. Delivery delay may also occur due to the

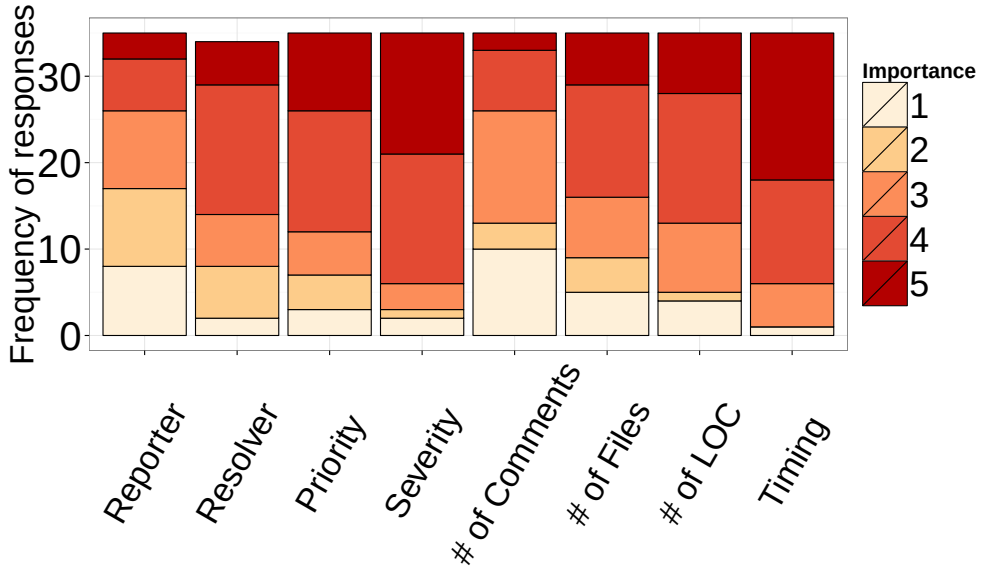


Figure 38 – Frequency of ranks per factor.

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

Observation 7—The time at which an issue is addressed during a release cycle and the issue severity are the factors that receive the highest ratings of importance. In *question #13* of our survey, we ask participants to rate the degree to which a factor is related to delivery delay. The factors that we list are: the reporter, the resolver, the priority, the severity, the number of comments, the number of modified files, the number of modified LOC, and the time at which an issue was addressed during a release cycle. The responses to *question #13* are based on a 5-points-Likert scale, *i.e.*, participants rate factors using ranks from 1 (strongly disagree) to 5 (strongly agree).

In Figure 38, we show the frequency of each rank per factor, while we show the average rating of each factor in Table 18. We observe that the factors that receive the highest ranks are *severity* and *timing*. This result is in agreement with our regression models that are presented in RQ3, in which *cycle queue rank* is one of the most influential variables (see Observation 6). Indeed, during the interview, *P06* further explains that if an issue that is risky is addressed in the end of a release cycle, such an issue is

Table 18 – Rating of factors related to delivery delay. The highest ratings are in bold.

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

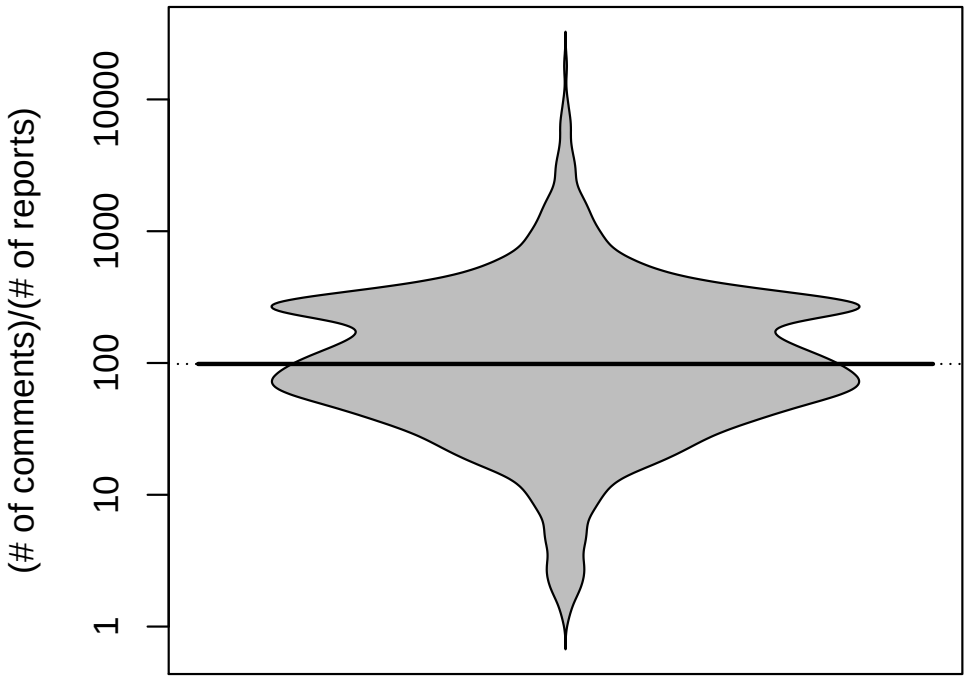


Figure 39 – Distribution of number of comments normalized by the number of reported issues.

likely to be delayed to the next cycle, so that it can receive additional testing.

On the other hand, the factors with the lowest ranks are *reporter*, and *# of comments*. We also asked our interviewees about these lower ratings. One of our interviewees explained that the reporter of an issue might influence delivery delay only in cases in which the reporter is also a Firefox employee. In these cases, the reporter will address the issue her/himself, which can speed up the shipping process.[1] As for the *# of comments*, another interviewee clarified that there are several passionate people on bugs that can inflate the number of comments even if the issue is easy to ship. For each reporter, we normalize the number of his/her comments by the number of his/her reported issues. We plot the distribution of the normalized number of comments in Figure 39. The median number of comments per reported issue is 98. Indeed, we ob-

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

Table 19 – $P\text{-}values$ of the comparisons between factors. Values in bold are < 0.05.

Factor x Factor	Reporter	Resolver	Priority	Severity
Reporter	—	$1.2e^{-1}$	$\mathbf{1.3}e^{-2}$	$\mathbf{1.4}e^{-5}$
Resolver	$1.2e^{-1}$	—	1	$1.2e^{-1}$
Priority	$\mathbf{1.3}e^{-2}$	$\mathbf{1.3}e^{-2}$	—	$4.8e^{-1}$
Severity	$\mathbf{1.4}e^{-5}$	$1.2e^{-1}$	$4.7e^{-1}$	—
# of Comments	$4.8e^{-1}$	$1.2e^{-1}$	$\mathbf{1.4}e^{-2}$	$\mathbf{1.6}e^{-5}$
# of Files	$1.8e^{-1}$	$7.9e^{-1}$	1	$6.6e^{-2}$
# of LOC	$\mathbf{3.0}e^{-2}$	1	1	$2.8e^{-1}$
Timing	$\mathbf{8.0}e^{-7}$	$\mathbf{3.2}e^{-2}$	$1.8e^{-1}$	1

Factor x Factor	# of Comments	# of Files	# of LOC	Timing
Reporter	$4.8e^{-1}$	$1.8e^{-1}$	$\mathbf{3.0}e^{-2}$	$\mathbf{8.1}e^{-7}$
Resolver	$1.2e^{-1}$	$7.9e^{-1}$	1	$\mathbf{3.2}e^{-2}$
Priority	$\mathbf{1.4}e^{-2}$	1	1	$1.8e^{-1}$
Severity	$\mathbf{1.6}e^{-5}$	$6.6e^{-2}$	$2.8e^{-1}$	1
# of Comments	—	$1.7e^{-1}$	$\mathbf{3.3}e^{-2}$	$\mathbf{9.6}e^{-7}$
# of Files	$1.7e^{-1}$	—	1	$\mathbf{1.4}e^{-2}$
# of LOC	$3.3e^{-2}$	1	—	$1.1e^{-1}$
Timing	$\mathbf{9.6}e^{-7}$	$\mathbf{1.4}e^{-2}$	$1.1e^{-1}$	—

serve reporters with a great number of comments (*e.g.*, 500 to 10,000 comments) per reported issue. This result suggest that the perception of our interviewee is likely to be true.

A Kruskal Wallis test indicates that the difference in ratings between metrics are statistically significant ($p = 0.01507$). Table 19 shows the Bonferroni corrected $p\text{-}values$ of the Dunn tests. We observe that the *timing* factor has significant larger response values than all the other factors except the *severity*, *priority*, and *LOC* factors ($p < 0.05$).

We also use Spearman's ρ to correlate the rating of the factors with (i) general experience (*question #1*) and (ii) project experience (*question #2*). The only statistically significant correlation that we observe is between the *timing* factor and general experience. We achieve a negative correlation of -0.36 ($p = 0.03235$). This result suggests that less experienced participants tend to report that the time at which an issue is addressed during a release cycle plays a more important role in delivery delay. One of our interviewees explains this observation by stating that *"when an issue is addressed early in the release cycle, it should have more time to be tested before integration,"* which can be helpful for fixes from less experienced resolvers. Finally, we also correlate the responses to *question #6* with general and project experience. However, no significant correlations were found.

Our survey participants report that the delivery of addressed issues may be delayed due to reasons that are related to the development activities, decision making, team collaboration, or risk. Moreover, delivery delay likely lead to user/developer frustration.

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

RQ5: Motivation

In this research question, we intend to complement our quantitative findings about the comparison between traditional and rapid release cycles regarding delivery delays. This investigation is important to gain deeper explanations as to why addressed issues may be delivered more quickly in traditional releases. Additionally, we intend to understand what are the reasons for the perceived success of adopting a rapid release cycle. This is also important to help project leaders with their decision of adopting a rapid release cycle rather than a traditional one.

RQ5: Results

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

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

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

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

within that time frame."

Theme 7— Delivery. The most recurrent perceived advantage of rapid release cycles is the *"faster delivery"*(33) of new functionalities. When asked about the motivation to use rapid release cycles, *F05* mentions *"increasing speed of getting new features to users,"* while *F06* mentions a similar statement: *"getting new features to users sooner."* Interestingly, not all participants that mentioned the time to deliver new functionalities report that rapid releases always reduce such time. For *F22*, rapid releases *"reduce the time to deliver issues to end users in some cases, and lengthen them in others."* More specifically, *F24* says that *"Low priority issues (new features) take less time to be delivered, whereas high priority ones (important bugs) take more time."*

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

Theme 8—Development activities. We do not observe a specific theme that is recurrent with respect to development activities. Instead, we observe a broad range of themes that are cited by the participants. Among such themes, we observe *quality*,(3) *more functionalities*,(2) *better motivation*,(2) and *better prototyping*.(2) Quality should be a measure of success of using rapid release cycles. According to *E26*, *"quality of delivered code should remain the same or improve"* after switching to rapid releases. Another way to measure the success of a rapid release cycle is the *number of functionalities*(2) that are completed. *E34* states the following: *"I would see if more issues were completely fixed"* as a measure of success.

Moreover, rapid releases may also impact team members' motivation. For instance, *F06*'s opinion about why to switch to rapid release cycles is *"the need to motivate the community via more frequent collaboration."* Finally, rapid release cycles may also improve prototyping activities. For instance, *P27* argues that, by adopting rapid releases, a development team can *"fix bugs quickly [and] prototype features, having results in few months."*

The allure of delivering addressed issues more quickly to users is the most recurrent motivator of switching to a rapid release cycle. Moreover, the allure of improving management flexibility and quality of addressed issues are other advantages that are perceived by our participants with respect to switching to rapid release cycles.

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

RQ6: Motivation

The main motivation for this research question is to solicit feedback about our quantitative findings. More specifically, we aim to understand to what extent our quantitative findings agree with the participants' perception of integration delays.

RQ6: Results

In this research question, we investigate how our participants feel about the data that we collect during our prior quantitative studies (Studies 1 and 2). This research question is divided into two subsections—one for each *theme* that is investigated in this thesis—*(i)* reasons for delivery delay in general and *(ii)* the impact of rapid release cycles on delivery delay (Figure 1).

Theme 9—delivery delay in general. In this analysis, we present the data that we collect in our prior studies (Chapter 3) and investigate if this data resonates with participants' experience. We provide the methodology of our data-related questions to participants through a web page that is mentioned in our surveys (see Appendix D).

Figure 40 shows the chart that we presented to participants. For example, 89% of the addressed issues skip two Firefox stable releases before being shipped to users. The most recurrent themes among the responses of participants to explain this data are: *team workload*$^{(5)}$ and *dependency*.$^{(2)}$ Among the responses that are related to *team workload*,$^{(5)}$ *E27* explains that *"committers are too busy,"* while *E26* argues that there might be *"delay[s] in review[s] when the issues [are] completed,"* which can generate delivery delay. Regarding *dependency*,$^{(2)}$ *E32*'s opinion is that delivery delay may happen due to *"the strong connection to other Eclipse projects which makes integration more costly (time consuming)."* Furthermore, two of our interviewees (*F11* and *F23*) provide us with examples of why addressed issues may be delayed due to dependency problems. For example, *F23* explains during the interview that delivery delay can happen when there are *"dependencies between projects and one of them gets done, but the other implementation takes a longer while."* Another example, provided by *F23* is when *"you release a bug fix but then you realize: Hey! These users are not able to use these websites anymore because web servers implement the spec in a wrong way or do some really weird things that are not expected."*

Additionally, we ask participants from the Eclipse and ArgoUML projects about

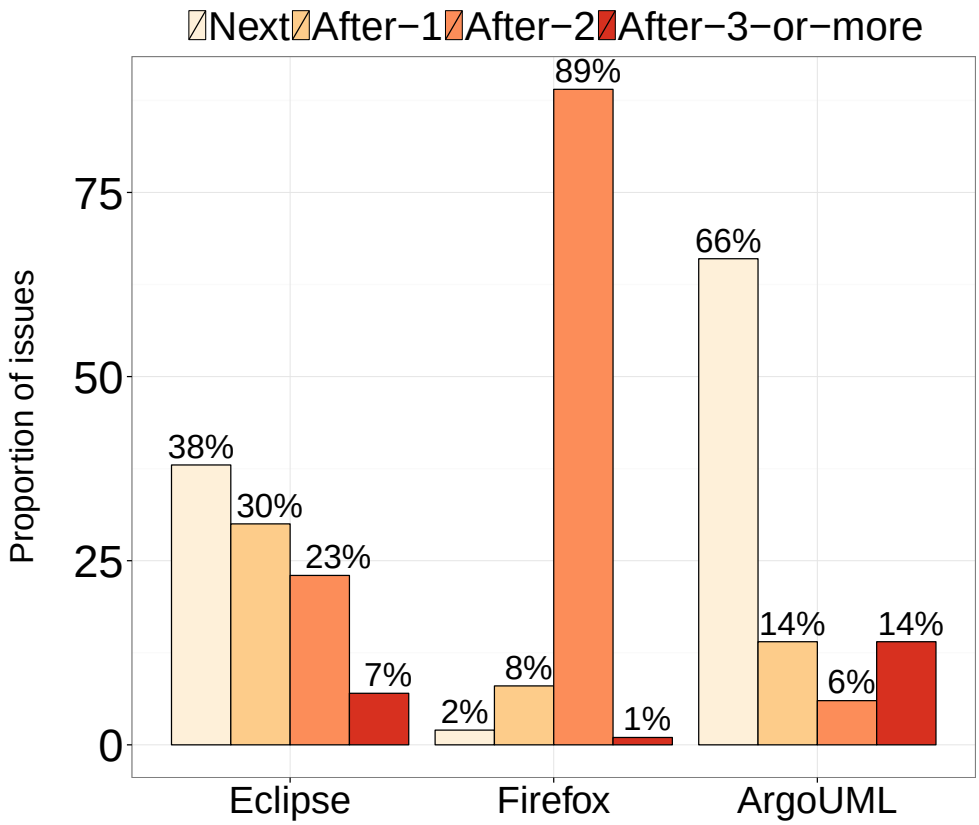


Figure 40 – Proportion of addressed issues that have their integration delayed by a given number of releases. For example, 89% of the addressed issues skip two Firefox stable releases before being shipped to users (this chart was already presented in Chapter 3.

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

Theme 10—Impact of switching to a rapid release cycle. We present the data that is shown in Figure 24d to the participants of the Firefox project. Figure 24d compares the delivery delay between traditional and rapid release cycles. We then ask if this result resonates with the participants' experience. More details about how we show this data to participants can be found in Appendix E. From the 14 responses that we received for this question, 5 participants explicitly disagree with our analysis, while 6 participants explicitly agree with it.

We interviewed two participants that disagree with the results (*F06* and *F09*). After providing extra explanation about our methodology and asking them to elaborate

on their responses, we could better understand their reasons. *F06* clarifies: *"I'm not surprised that there are things in that bucket"* (the short delays due to minor traditional releases), instead *"I'm surprised that there are many of them."* In addition, *F09* declared *"I misunderstood [your] question, but now it [(the data)] makes sense."* With respect to the remaining participants that disagree with our results, they inform us that the data does not resonate with their experience. For instance, *F21* provides the following opinion *"this does not resonate with my experience. I find the traditional model is much much slower than rapid release to get fixes in users hands."*

From the set of participants that agree with our results, two of them explain that the behaviour that is presented by the traditional release data is due to the *integration rush*(2) that happens prior to shipping. *F15*'s opinion is that *"since missing a release cycle isn't a big deal, more features are kept from being released until they're properly polished instead of being rushed at the end of a long release cycle." F22* also provides us with a reasonable explanation when stating that our result *"makes perfect sense as issues will, unless fast-tracked or held back, be released a set quantum of time after they are completed. This is dominated by the timing of the release schedule, not by the timing of the discovery or fix."*

The dependency of addressed issues on other projects and team workload are major perceived reasons to explain our findings about delivery delays. Moreover, participants are divided when explaining why traditional releases may have shorter delivery delays. Nevertheless, the fact that in rapid releases an integration rush is no longer needed and that additional time can be spent on polishing addressed issues emerge as main explanations as to why traditional releases can have shorter delivery delays.

5.4 Threats to Validity

Internal Validity. In our qualitative analysis, we had few participants (37). However, such an analysis is important to (i) gain insights from additional projects (ArgoUML and Eclipse) and (ii) better understand *why* delivery delay happens. Moreover, we choose methods from Grounded Theory to perform our qualitative analysis. Although the coding process is performed by two authors independently and reviewed by a third author, we cannot claim that we reach all the perspectives that are possible from our questions.

External Validity. External threats are concerned with our ability to generalize

our results. We survey participants from three different projects (Firefox, ArgoUML, and Eclipse) and perform four follow-up interviews to gain deeper insights about the responses that we obtain from our surveys. Still, we cannot claim that our results are generalizable to other software projects that are not studied in this work. Hence, replication of this work using other projects is required in order to reach more general conclusions.

5.5 Related Work

Since this work is the first to qualitatively investigate the delivery delay of addressed issues (*i.e.*, by surveying and interviewing developers of our subject projects), we survey related research that used a methodology similar to ours to investigate other topics.

Treude *et al.* (TREUDE; FILHO; KULESZA, 2015) studied what information developers expect in a summary of development activities. The authors found that *unexpected events* are as important as *expected events* in a summary of what a developer did. Coelho *et al.* (COELHO et al., 2016) performed a survey based study to investigate the perspectives of developers concerning exception handling bug hazards in Android apps. The authors found that 64% of the participants recognized that they handle exceptions most of the time during apps development. Pinto *et al.* (PINTO; KULESZA; TREUDE, 2015) investigated whether developers were aware about the performance issues that their tool was able to automatically find. Pinto *et al.* found that the minority of participants were aware of the performance degradation cases that their tool was able to automatically find. Shimagaki *et al.* (SHIMAGAKI et al., 2016) qualitatively studied the impact that modern code review practices have on the quality of a proprietary system that is developed by Sony Mobile. The authors surveyed 93 stakeholders, of which 15 attended to follow-up interviews and 25 senior engineers answered follow-up surveys. Differently from prior work, our study focus on the qualitative investigation of delivery delays of addressed issues.

5.6 Conclusions

In our qualitative study, we survey 37 participants from the Firefox, ArgoUML, and Eclipse projects. We make the following observations:

- The perceived reasons for delivery delay of addressed issues are primarily related to activities such as development, decision making, team collaboration, and risk management (see Themes 1, 2, 3, and 5).

- The dependency of issues on other projects and team workload are the main perceived reasons to explain our data about delivery delay in general (see Theme 1).
- The allure of delivering addressed issues more quickly to users is the most recurrent motivator for switching to a rapid release cycle (see Theme 7). In addition, the allure of improving management flexibility and quality of addressed issues are other advantages that are perceived by our participants (see Themes 6 and 8).
- Integration rush and the increased time that is spent on polishing addressed issues (during rapid releases) emerge as one of the main explanations as to why traditional releases may achieve shorter delivery delays (see Theme 10).

Today the Firefox project is often used in support of proposals for moving to a rapid release cycle throughout many development organizations worldwide. Yet there has never been any studies that extensively explored the benefits and challenges (regarding delivery delays) of rapid release cycles on any project (till our work). In this regard, our quantitative and qualitative observations may serve any organization that is interested in adopting a rapid release cycle. For instance, even though the allure of delivering addressed issues more quickly is the most recurrent motivator to adopt rapid releases (Theme 7), we observe that this often is not achieved (Observation 2). In summary, our study provides real observations and offers a wider context of the dis/advantages of adopting a rapid release strategy.

6 Conclusions

Issue Tracking Systems have 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 may still suffer delivery delays that software 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 delays after being addressed. To perform our studies, we leverage data that is recorded in *Version Control Systems* and *Issue Tracking Systems* and we survey team members of our subject projects. We answer the questions that guided our studies below:

- ***Study 1—How frequent is delivery delay?*** Delivery delays are frequent in our subject projects (*e.g.*, 34% to 98% of addressed issues were delayed by at least one release) (Chapter 3).
- ***Study 2—Do rapid releases reduce delivery delays?*** Rapid releases are not silver bullets to reduce the delivery delay of addressed issues. Instead, rapid releases may improve the consistency of the delivery rate of addressed issues (Chapter 4).
- ***Study 3—Why do delivery delays occur?*** Limited testing capacity and lack of code review are reasons as to why delivery delays occur. Also, the allure of delivering addressed issues more quickly is the main motivator to adopt a rapid

release cycle although such a quicker delivery it is not always achieved according to our empirical observations (Chapter5).

Our main findings suggest that delivery delays that are incurred by addressed issues introduce a non-negligible bottleneck in the software development process. Organizations should invest on improving their process (*e.g.*, testing or code review processes) to mitigate such delivery delays and provide a more efficient delivery of addressed issues. Future research in Software Engineering should not only consider the required time to develop a new software functionality when assessing new tools or practices, but also whether such new tools or practices will likely help the delivery of new software functionalities.

6.1.1 Future Work

The studies that are performed in this thesis pave a way for several future work possibilities. We outline some venues for future work below.

Replication. Future work could replicate the studies that are performed in this thesis using other large scale software projects. Such replication studies are important to reach more generalizable conclusions about delivery delays.

Software Quality. To study the trade-off between delivery delays and software quality is important. For example, one could empirically investigate whether rapid releases deliver software issues with a higher quality despite of the increased delivery delay of addressed issues.

Tooling. Several tools could be developed to improve the practice of Software Engineering. For instance, *Issue Tracking Systems* could allow the tagging of addressed issues that are going to be delayed. The development team could also fill in the reasons why an addressed issue is being delayed. Such type of features can be used for documentation purposes as well as for letting users and contributors more aware as to why an addressed issue is being delayed.

Prediction. A considerable effort has been invested in the prediction of software bugs. Bugs are already known to be harmful for software systems, so that their prediction is of great importance to avoid unwanted costs. Another possibility of future work is the accurate prediction of delivery delays, which could allow for a better planing and risk mitigation in software projects.

Bibliography

ADAMS, B.; MCINTOSH, S. Modern Release Engineering in a Nutshell: Why Researchers should Care. In: *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. [S.l.: s.n.], 2016. p. 78–90. Cited on page 72.

ALGHAMDI, H. M. et al. An automated approach for recommending when to stop performance tests. In: IEEE. *Proceedings of the International Conference on Software Maintenance and Evolution*. [S.l.], 2016. p. toAppear. Cited on page 103.

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. Cited 4 times on pages 13, 20, 41, and 42.

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 (CASCON)*. [S.l.: s.n.], 2008. p. 23–37. Cited 3 times on pages 12, 18, and 73.

ANVIK, J.; HIEW, L.; MURPHY, G. C. Coping with an open bug repository. In: *Proceedings of the 2005 OOPSLA Workshop on Eclipse Technology eXchange*. [S.l.: s.n.], 2005. (eclipse '05), p. 35–39. Cited on page 13.

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. Cited 3 times on pages 12, 18, and 19.

BASKERVILLE, R.; PRIES-HEJE, J. Short cycle time systems development. In: *Information Systems Journal*. [S.l.: s.n.], 2004. v. 14, n. 3, p. 237–264. Cited on page 72.

BAYSAL, O.; DAVIS, I.; GODFREY, M. W. A tale of two browsers. In: ACM. *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*. [S.l.], 2011. p. 238–241. Cited 3 times on pages 72, 93, and 94.

BECK, K. Extreme programming explained: embrace change. In: . [S.l.]: Addison-Wesley Professional, 2000. Cited on page 72.

BELLER, M.; GOUSIOS, G.; ZAIDMAN, A. How (much) do developers test? In: IEEE. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. [S.l.], 2015. v. 2, p. 559–562. Cited on page 103.

BHATTACHARYA, P.; NEAMTIU, I. Bug-fix time prediction models: Can we do better? In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. [S.l.: s.n.], 2011. (MSR '11), p. 207–210. Cited on page 20.

BREIMAN, L. Random forests. In: *Machine Learning*. [S.l.: s.n.], 2001. (Springer Journal no. 10994), p. 5–32. Cited on page 44.

BROOKS, F. P. *The mythical man-month*. [S.l.]: Addison-Wesley Reading, MA, 1975. v. 1995. Cited on page 44.

CHARMAZ, K. *Constructing grounded theory*. [S.l.]: Sage, 2014. Cited on page 99.

CLIFF, N. Dominance statistics: Ordinal analyses to answer ordinal questions. In: *Psychological Bulletin*. [S.l.: s.n.], 1993. v. 114, n. 3, p. 494–509. Cited 3 times on pages 38, 77, and 79.

COELHO, R. et al. Exception handling bug hazards in android. *Empirical Software Engineering*, Springer, p. 1–41, 2016. Cited on page 113.

COSTA, D. A. d. et al. An empirical study of delays in the integration of addressed issues. In: *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. [S.l.: s.n.], 2014. p. 281–290. Cited 5 times on pages 24, 43, 44, 45, and 83.

COSTA, D. A. da et al. The impact of switching to a rapid release cycle on the integration delay of addressed issues: an empirical study of the mozilla firefox project. In: ACM. *Proceedings of the 13th International Workshop on Mining Software Repositories*. [S.l.], 2016. p. 374–385. Cited on page 72.

DELONE, W. H.; MCLEAN, E. R. The delone and mclean model of information systems success: a ten-year update. *Journal of management information systems*, v. 19, n. 4, p. 9–30, 2003. Cited on page 12.

DUNN, O. J. Multiple comparisons among means. *Journal of the American Statistical Association*, Taylor & Francis Group, v. 56, n. 293, p. 52–64, 1961. Cited 2 times on pages 38 and 100.

DUNN, O. J. Multiple comparisons using rank sums. *Technometrics*, Taylor & Francis, v. 6, n. 3, p. 241–252, 1964. Cited 2 times on pages 38 and 100.

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 2 times on pages 48 and 85.

FISHER, R. A. *Statistical methods for research workers*. [S.l.]: Genesis Publishing Pvt Ltd, 1925. Cited on page 100.

FREEDMAN, D. A. *Statistical models: theory and practice*. [S.l.]: cambridge university press, 2009. Cited on page 47.

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*. [S.l.: s.n.], 2010. (RSSE '10), p. 52–56. Cited 6 times on pages 13, 20, 41, 42, 45, and 84.

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*. [S.l.: s.n.], 2010. (ICSE '10), p. 495–504. Cited on page 20.

HANLEY, J. A.; MCNEIL, B. J. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology*, v. 143, n. 1, p. 29–36, 1982. Cited on page 46.

HARRELL, F. E. *Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis*. [S.l.]: Springer, 2001. Cited 4 times on pages 4, 47, 83, and 85.

HERBSLEB, J. D.; MOCKUS, A. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on software engineering*, IEEE, v. 29, n. 6, p. 481–494, 2003. Cited on page 12.

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*. [S.l.: s.n.], 2008. (MSR '08), p. 145–148. Cited 4 times on pages 20, 45, 53, and 92.

HOOIMEIJER, P.; WEIMER, W. Modeling bug report quality. In: *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*. [S.l.: s.n.], 2007. (ASE '07), p. 34–43. Cited on page 19.

HOWELL, D. C. Median absolute deviation. In: *Encyclopedia of Statistics in Behavioral Science*. [S.l.: s.n.], 2005. Cited 2 times on pages 22 and 77.

IASONOS, A. et al. How to build and interpret a nomogram for cancer prognosis. In: *Journal of Clinical Oncology*. [S.l.]: American Society of Clinical Oncology, 2008. v. 26, n. 8, p. 1364–1370. Cited 2 times on pages 85 and 88.

JEONG, G.; KIM, S.; ZIMMERMANN, T. Improving bug triage with bug tossing graphs. In: *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.: s.n.], 2009. p. 111–120. Cited 2 times on pages 45 and 84.

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*. [S.l.: s.n.], 2013. (MSR '13), p. 101–110. Cited 7 times on pages 12, 13, 14, 21, 45, 69, and 84.

KAMPSTRA, P. et al. Beanplot: A boxplot alternative for visual comparison of distributions. In: *Journal of Statistical Software*. [S.l.: s.n.], 2008. v. 28, n. 1, p. 1–9. Cited 2 times on pages 77 and 79.

KHOMH, F. et al. Do faster releases improve software quality? an empirical case study of mozilla firefox. In: IEEE. *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR)*. [S.l.], 2012. p. 179–188. Cited 4 times on pages 72, 78, 93, and 94.

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*. [S.l.: s.n.], 2006. (MSR '06), p. 173–174. Cited 4 times on pages 13, 20, 41, and 42.

KRUSKAL, W. H.; WALLIS, W. A. Use of ranks in one-criterion variance analysis. *Journal of the American statistical Association*, Taylor & Francis, v. 47, n. 260, p. 583–621, 1952. Cited 2 times on pages 38 and 100.

LEYS, C. et al. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. In: *Experimental Social Psychology Journal*. [S.l.: s.n.], 2013. v. 49, p. 764–766. Cited 2 times on pages 22 and 77.

MÄNTYLÄ, M. V. et al. On rapid releases and software testing: a case study and a semi-systematic literature review. In: *Journal of Empirical Software Engineering*. [S.l.]: Springer, 2014. p. 1–42. Cited 2 times on pages 72 and 93.

MANTYLA, M. V. et al. On rapid releases and software testing. In: *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. [S.l.: s.n.], 2013. p. 20–29. Cited on page 22.

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*. [S.l.: s.n.], 2011. (Promise '11), p. 11:1–11:8. Cited 4 times on pages 13, 20, 41, and 42.

MCINTOSH, S. et al. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Software Engineering*, v. 21, n. 5, p. 2146–2189, 2016. Cited on page 104.

MOCKUS, A.; FIELDING, R. T.; HERBSLEB, J. D. Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, v. 11, n. 3, p. 309–346, 2002. Cited on page 53.

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 3 times on pages 13, 21, and 69.

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 3 times on pages 13, 21, and 69.

NAGAPPAN, N.; BALL, T. Use of relative code churn measures to predict system defect density. In: *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. [S.l.: s.n.], 2005. p. 284–292. Cited 2 times on pages 45 and 84.

OLKIN, G. C. S. F. I. Springer texts in statistics. 2002. Cited on page 47.

PANJER, L. D. Predicting eclipse bug lifetimes. In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. [S.l.: s.n.], 2007. (MSR '07), p. 29–. Cited 3 times on pages 20, 43, and 82.

PINTO, F.; KULESZA, U.; TREUDE, C. Automating the performance deviation analysis for multiple system releases: An evolutionary study. In: IEEE. *Source Code Analysis and Manipulation (SCAM), 2015 IEEE 15th International Working Conference on*. [S.l.], 2015. p. 201–210. Cited on page 113.

RAHMAN, M. T.; RIGBY, P. C. Release stabilization on linux and chrome. *IEEE Software*, v. 32, n. 2, 2015. Cited on page 34.

ROMANO, J. et al. Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? In: *Annual meeting of the Florida Association of Institutional Research*. [S.l.: s.n.], 2006. Cited on page 38.

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 19.

SARLE, W. The varclus procedure. *SAS/STAT User's Guide,*, 1990. Cited on page 48.

SCHROTER, A.; BETTENBURG, N.; PREMRAJ, R. Do stack traces help developers fix bugs? In: *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. [S.l.: s.n.], 2010. p. 118–121. Cited 2 times on pages 43 and 82.

SHIHAB, E. et al. Predicting re-opened bugs: A case study on the eclipse project. In: IEEE. *Proceedings of 17th Working Conference on Reverse Engineering (WCRE)*. [S.l.], 2010. p. 249–258. Cited on page 82.

SHIMAGAKI, J. et al. A study of the quality-impacting practices of modern code review at sony mobile. In: ACM. *Proceedings of the 38th International Conference on Software Engineering Companion*. [S.l.], 2016. p. 212–221. Cited on page 113.

SOUZA, R.; CHAVEZ, C.; BITTENCOURT, R. Rapid releases and patch backouts: A software analytics approach. In: *IEEE Software Journal*. [S.l.]: IEEE, 2015. v. 32, n. 2, p. 89–96. Cited 2 times on pages 72 and 93.

SOUZA, R.; CHAVEZ, C.; BITTENCOURT, R. A. Do rapid releases affect bug reopening? a case study of firefox. In: IEEE. *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*. [S.l.], 2014. p. 31–40. Cited 2 times on pages 72 and 93.

SPEARMAN, C. The proof and measurement of association between two things. *The American journal of psychology*, JSTOR, v. 15, n. 1, p. 72–101, 1904. Cited on page 100.

STEEL, R. G.; JAMES, H. *Principles and procedures of statistics: with special reference to the biological sciences*. [S.l.], 1960. Cited on page 48.

SUBRAMANIAM, C.; SEN, R.; NELSON, M. L. Determinants of open source software project success: A longitudinal study. In: *Journal of Decision Support Systems*. [S.l.]: Elsevier, 2009. v. 46, n. 2, p. 576–585. Cited on page 12.

TIAN, Y. et al. On the unreliability of bug severity data. *Empirical Software Engineering*, Springer, p. 1–26, 2015. Cited 2 times on pages 53 and 92.

TREUDE, C.; FILHO, F. F.; KULESZA, U. Summarizing and measuring development activity. In: ACM. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. [S.l.], 2015. p. 625–636. Cited on page 113.

WEIB, C. et al. How long will it take to fix this bug? In: *Proceedings of the Fourth International Workshop on Mining Software Repositories*. [S.l.: s.n.], 2007. (MSR '07), p. 1–. Cited 4 times on pages 13, 20, 41, and 42.

WILKS, D. S. Statistical methods in the atmospheric sciences. In: . [S.l.]: Academic press, 2011. v. 100. Cited 2 times on pages 77 and 79.

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. Cited 2 times on pages 20 and 21.

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*. [S.l.: s.n.], 2013. (ICSE '13), p. 1042–1051. Cited 4 times on pages 13, 20, 41, and 42.

Appendix

APPENDIX A – Firefox Survey

Understanding the Delivery Delay of Completed Issues

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

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

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

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

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

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

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

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

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

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

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

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

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

Delivery delay in your project

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

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

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

Reasons related to delivery delay

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

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

Reasons related to delivery delay

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

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

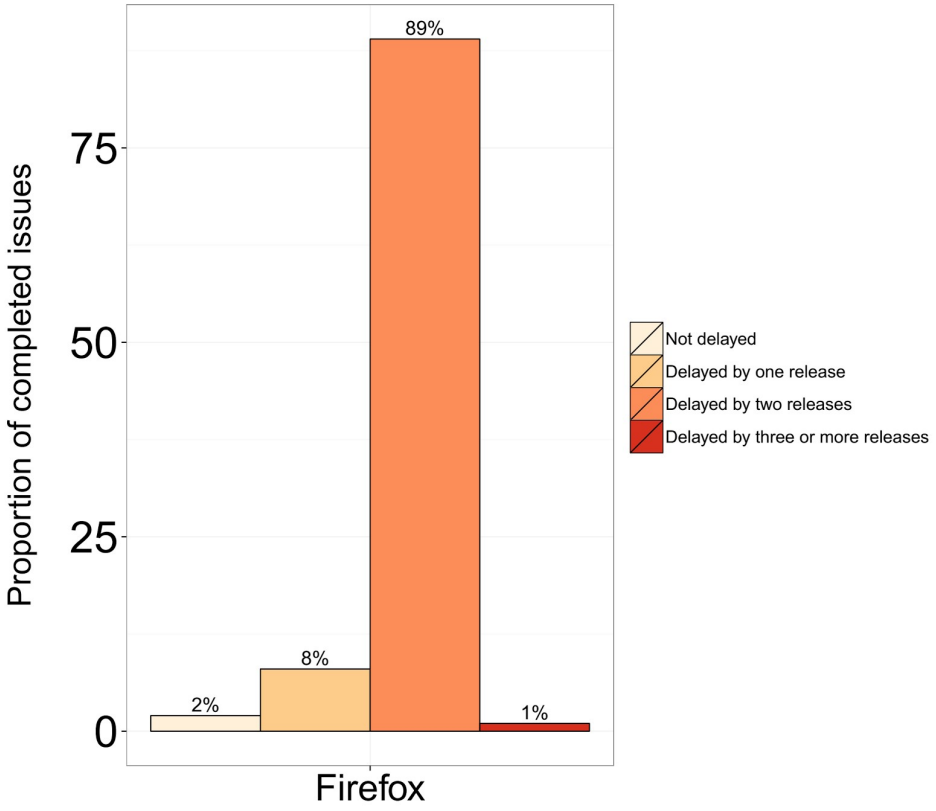
	Strongly agree	Agree	Neither agree nor disagree	Disagree	Strongly disagree
The reporter of a completed issue					
The resolver of a completed issue					
The priority value of a completed issue					
The severity value of a completed issue					
Number of comments recorded in a completed issue					
Number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

Feedback on the results about delivery delay

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

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

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



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

Shifting to a rapid release cycle

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

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

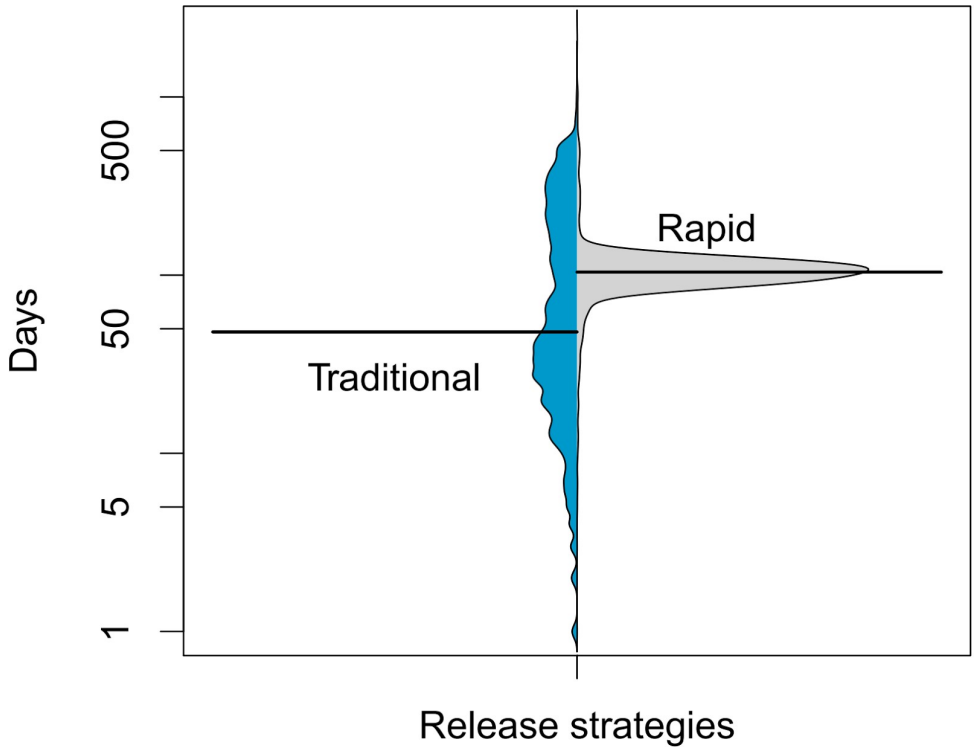
    - ◯ Yes
    - ◯ No

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

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

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

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



Ending our questionnaire

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

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

- Yes
- No

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

- Yes
- No

24. **Do you have further comments for us?**

APPENDIX B – ArgoUML Survey

Understanding The Delivery Delay of Completed Issues

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

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

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

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

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

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

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

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

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

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

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

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

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

Delivery delay in your project

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

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

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

Reasons related to delivery delay

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

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

Reasons related to delivery delay

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

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

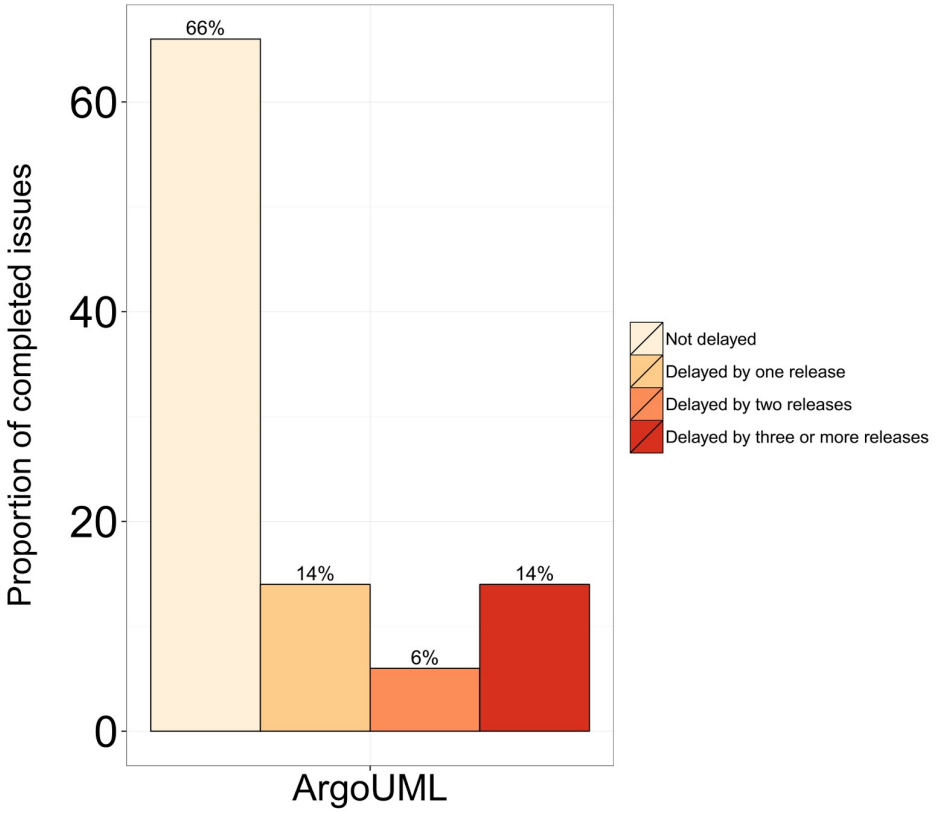
	Strongly agree	Agree	Neither agree nor disagree	Disagree	Strongly disagree
The reporter of a completed issue					
The resolver of a completed issue					
The priority value of a completed issue					
The severity value of a completed issue					
Number of comments recorded in a completed issue					
Number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

Feedback on the results about delivery delay

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

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

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



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

Shifting to a rapid release cycle

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

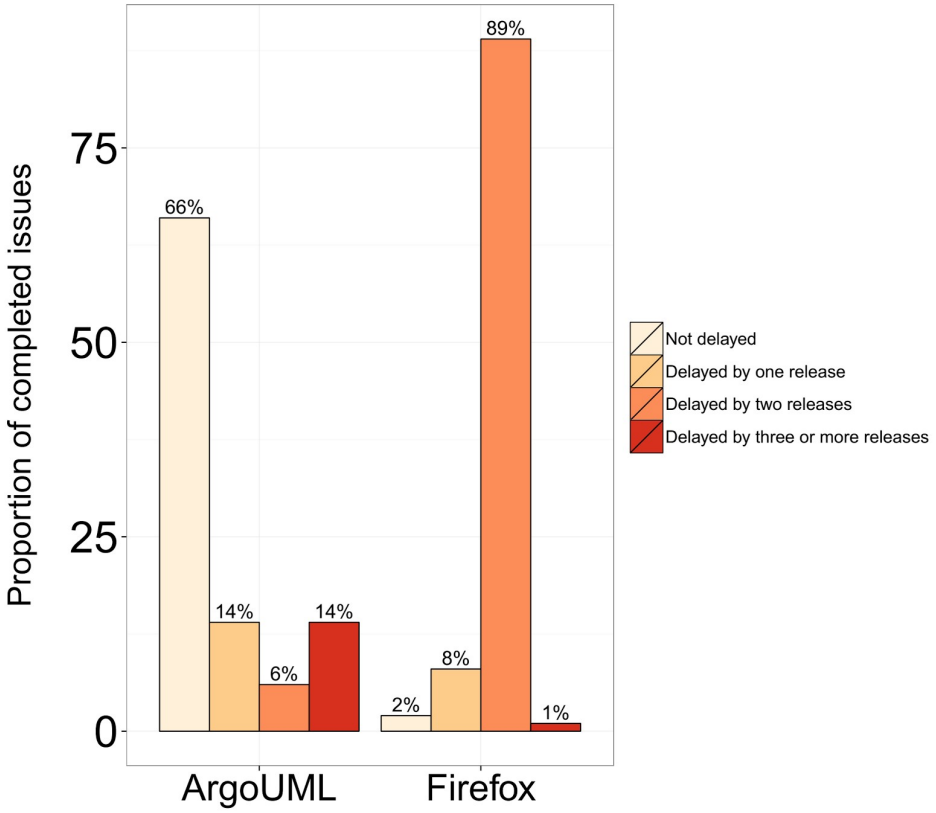
17. **Do you have experience working on a rapid release cycle in any other project?**
    Mark only one oval.

    - ◯ Yes
    - ◯ No

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

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

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



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

Ending our questionnaire

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

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

- Yes
- No

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

- Yes
- No

24. **Do you have further comments for us?**

APPENDIX C – Eclipse Survey

Understanding The Delivery Delay of Completed Issues

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

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

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

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

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

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

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

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

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

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

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

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

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

Delivery delay in your project

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

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

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

Reasons related to delivery delay

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

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

Reasons related to delivery delay

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

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

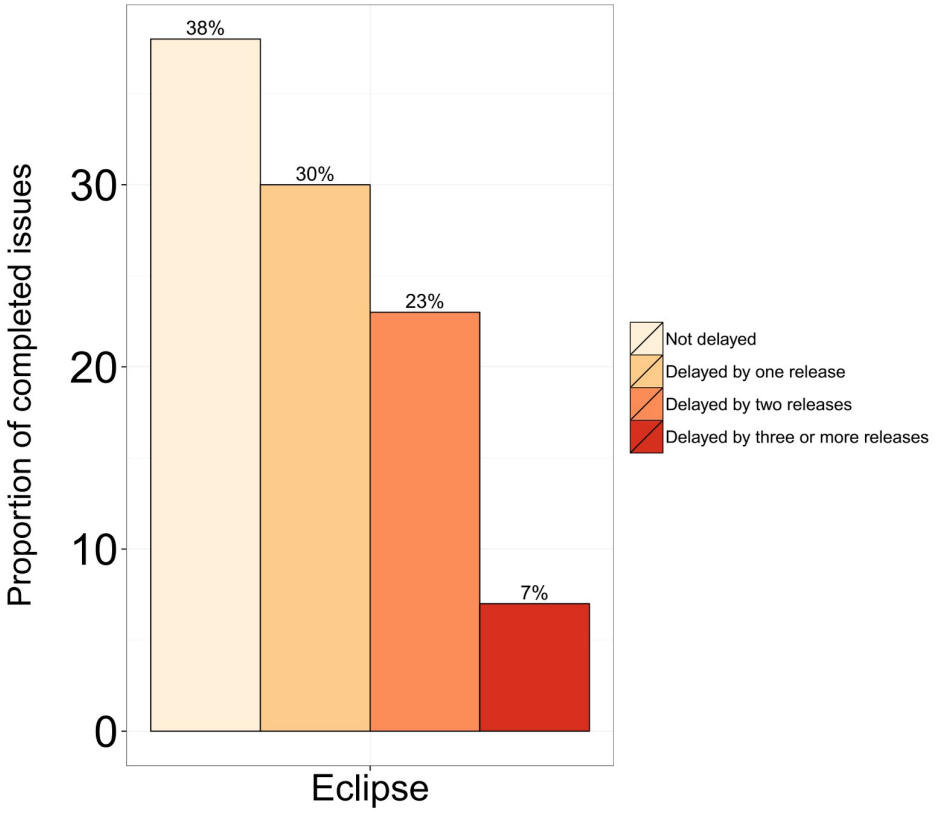
	Strongly agree	Agree	Neither agree nor disagree	Disagree	Strongly disagree
The reporter of a completed issue					
The resolver of a completed issue					
The priority value of a completed issue					
The severity value of a completed issue					
Number of comments recorded in a completed issue					
Number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

Feedback on the results about delivery delay

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

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

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



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

Shifting to a rapid release cycle

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

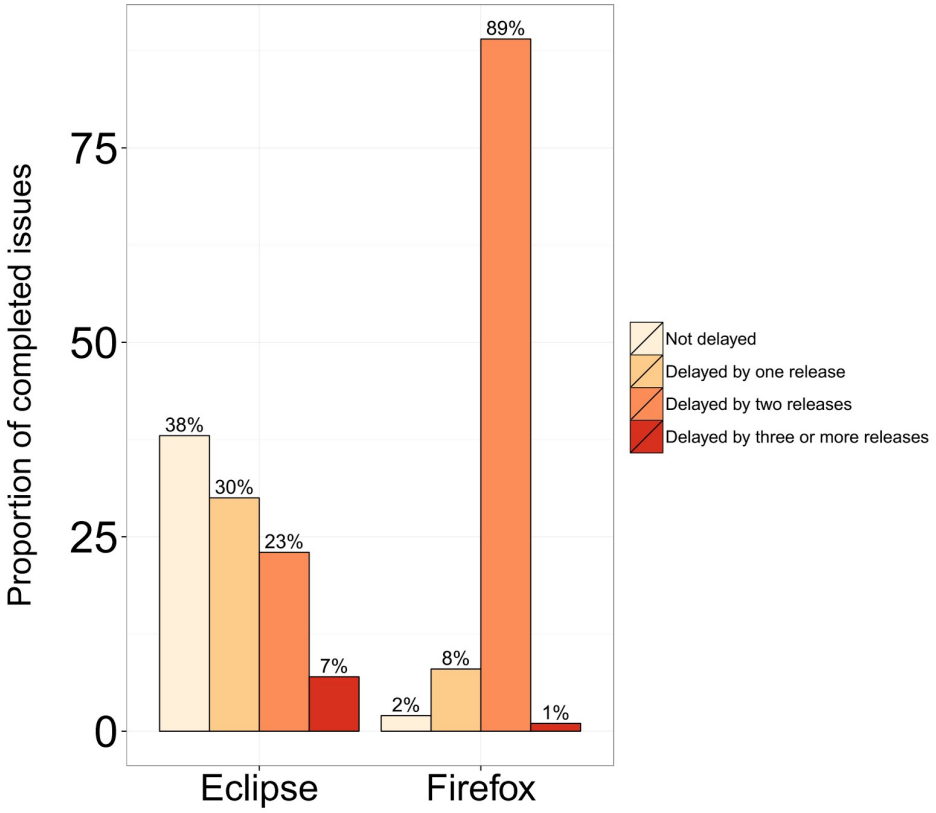
17. **Do you have experience working on a rapid release cycle in any other project?**
    Mark only one oval.

    - Yes
    - No

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

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

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



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

Ending our questionnaire

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

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

- Yes
- No

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

- Yes
- No

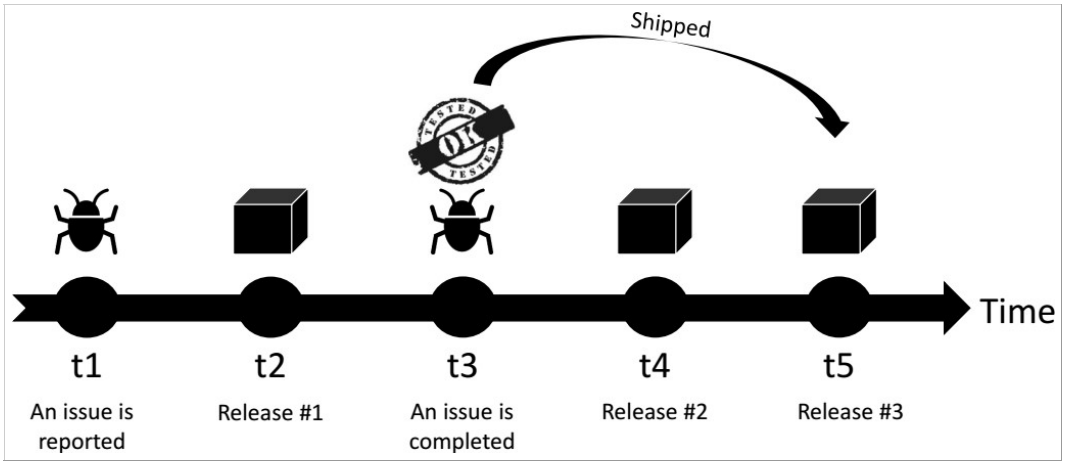
24. **Do you have further comments for us?**

APPENDIX D – Methodology Web Page I

How do we compute the delivery delay of completed issues?

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

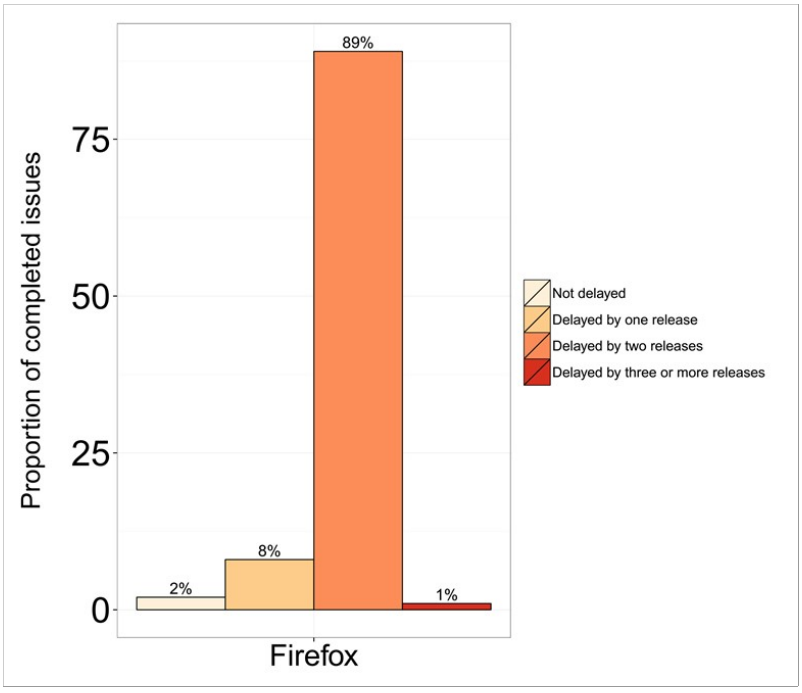
Delivery delay measures how long it takes for a system functionality (i.e., an issue) to be delivered to the end user from the time at which the issue was completed (i.e., implemented and tested).



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

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

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

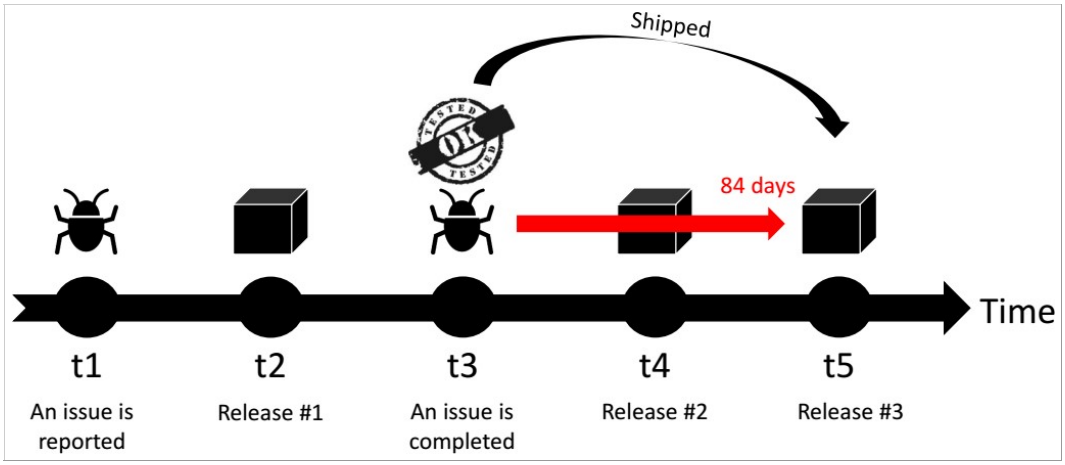


The figure shows that 89% of the Firefox completed issues miss 2 official releases before being shipped to end users.

APPENDIX E – Methodology Web Page II

How do we compare rapid vs traditional releases?

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



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

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

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

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