Understanding the impact of release policies on software development processes

Zeinab Abou Khalil

▶ **To cite this version:**

Zeinab Abou Khalil. Understanding the impact of release policies on software development processes. Programming Languages [cs.PL]. Université de Lille; Université de Mons, 2021. English. NNT : 2021LILUI011 . tel-03366534v2

HAL Id: tel-03366534
https://theses.hal.science/tel-03366534v2

Submitted on 5 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Doctoral Thesis

UNDERSTANDING THE IMPACT OF RELEASE POLICIES ON SOFTWARE DEVELOPMENT PROCESSES

Zeinab Abou Khalil

Thesis committee:

Prof. Tom Mens *University of Mons, Belgium*	Co-director
Prof. Laurence Duchien *University of Lille, France*	Co-director
Prof. Andy Zaidman *Delft University of Technology, The Netherlands*	Reviewer
Prof. Jean-Rémy Falleri *University of Bordeaux, France*	Reviewer
Prof. Véronique Bruyere *University of Mons, Belgium*	President
Dr. Eleni Constantinou *Eindhoven University of Technology, The Netherlands*	Examiner
Dr. Clément Quinton *University of Lille, France*	Invited

A dissertation submitted in fulfilment of the requirements of the degree of PhD of Science at the

Ecole doctorale Sciences Pour l'Ingénieur, Université de Lille
&
Faculté de Sciences, Université de Mons

26 February, 2021

Thèse de Doctorat

COMPRENDRE L'IMPACT DES POLITIQUES DE PUBLICATION SUR LES PROCESSUS DE DÉVELOPPEMENT LOGICIEL

Zeinab Abou Khalil

Comnité de thèse:

Prof. Tom Mens *Université de Mons, Belgique*	Directeur
Prof. Laurence Duchien *Université de Lille, France*	Directeur
Prof. Andy Zaidman *Université de technologie de Delft, Pays-Bas*	Rapporteur
Prof. Jean-Rémy Falleri *Université de Bordeaux, France*	Rapporteur
Prof. Véronique Bruyere *Université de Mons, Belgique*	Président
Dr. Eleni Constantinou *Université de technologie d'Eindhoven, Pays-Bas*	Examinateur
Dr. Clément Quinton *Université de Lille, France*	Invité

Thèse de doctorat pour obtenir le titre de
Docteur en Sciences
de

l'Ecole doctorale Sciences Pour l'Ingénieur, Université de Lille
et
la Faculté de Sciences, Université de Mons

26 Février, 2021

Dedicated to my family.

Acknowledgements

I would like to express my gratitude to all the people that have contributed to the realization of this thesis.

Foremost, I would like to thank my two supervisors, Tom Mens and Laurence Duchien, for offering me the opportunity to carry out this doctoral work. Thank you for the many advices and discussions that helped me achieve this work. Thank you Tom, a second time, for your moral support that helped me overcome the PhD challenges, and I am very grateful for your dedication towards helping me make the most of my PhD. I would also like to thank Eleni Constantinou for her wisdom, friendship, understanding, and teaching me how to be a successful researcher. Besides my supervisors, I would like to thank all the jury members who took the time to read and comment my thesis. I would also like to appreciate the feedback and support I received from Professor Alexander Serebrenik.

I am also grateful to all the researchers that worked with me during this thesis. Thanks to Clément Quinton for his help and support when we work together. Thanks to Foutse Khomh for the collaboration and for welcoming me as a visiting researcher.

Moreover, I would like to thank all of my friends that their companionship and unconditional support made this journey exciting and delightful. Finally, I must express my profound gratitude to my family. Without their unconditional love and support, I never would have been able to get this far. I have always been far away from home, but their care is always by my side.

Thanks to the SNCB for all the go and backs between Lille and Mons.

Abstract

The advent of delivering new features faster has led many software projects to change their development processes towards more rapid release models where releases are shipped using release cycles of weeks or days. The adoption of rapid release practices has significantly reduced the amount of stabilization time, the time it takes for a software product's failure rate to reach close to the steady-state, available for new features. This forces organizations to change their development process and tools to release to the public, in a timely manner and with good quality. Rapid releases are claimed to offer a reduced time-to-market and faster user feedback; end-users benefit of faster access to functionality improvements and security updates and improve turnaround time for fixing bad bugs. Despite these benefits, previous research has shown that rapid releases often come at the expense of reduced software reliability. Despite the increasing adoption of rapid releases in open-source and commercial software, the effects of this practice on the software development process are not well understood.

The goal of this thesis is to provide a deeper understanding of how rapid releases impact different parts of the open-source software development process. We present empirical evidence about the short and long-term impact of rapid releases on the bug handling and testing process in open source organizations; and the plan and tools needed for successful adoption of rapid releases. This thesis presents an empirical case study of rapid releases in Eclipse and Mozilla Firefox projects. We follow a mixed-methods approach where we analyze software repositories, containing different types of data such as source code, testing data and software issues; and we conduct a survey with Eclipse developers. This helps understand the evolution and changes of the software development process, the plans and practices needed for successful adoption of rapid releases, and identifying several future research directions calling for further investigation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The goal of this thesis is to provide a deeper understanding of how rapid releases impact different parts of the open-source software development process. This chapter introduces the thesis context and concepts necessary to develop the background knowledge that will help the reader to understand the remainder of this thesis. In Section 1.1, we present the thesis context. We stated our thesis statement in Section 1.2. In Section 1.3, we summarize the thesis contribution. Finally, Section 1.4 presents the how the dissertation is organized.

1.1 Thesis Context

In large collaborative software projects with many features and requirements, and involving hundreds of people working on the same massive piece of code, development teams formalize what should be done (good practices) or not (bad practices) and generally define the plan of the software development process, what features to include in the software product and when to release a new version of the software product [10]. In this context, the software development process is defined as the way in which the software product is developed, from its definition to its delivery and maintenance. Software development processes aim to help project teams in organizing and structuring software projects. This process usually includes a set of methods, tools, and practices.
It is estimated that 80% of the software development effort is spent on software maintenance [100]. In the light of the sheer size and complexity of today's software systems, it is no wonder that software maintenance, especially bug handling activity, has become a challenging task [141]. The bug handling process is a part of software development and maintenance process. An efficient bug handling process, a process to correct software bugs, is critical for the success of software projects. In the last decades, while society has been increasingly relying on software, the cost of software maintenance, compared to the global cost of software, has grown [30]. Software maintenance refers to modifying and updating a software product after its delivery to correct faults and improve performance. In recent years, software maintenance has become more challenging due to the increasing number of bugs in large-scale and complex software programs [141]. Maintenance refers to an activity that occurs at any time after the implementation of the new development project, while the software evolution is defined as examining the dynamic behaviour of systems, i.e., how they change [49]. It is important to study the software evolution to better understand the software development process [86]. Software evolution analysis deals with software changes, their causes, and their effects. Such analysis uses all types of software repositories to perform historical analysis. Such data involve the release history with all the source code and the change information, bug report data, and data extracted from the running system. In particular, the analysis of release, source code and issue data have gained in importance as they leverage valuable information to analyze the evolution of software [33]. Software Organizations improve the quality, speed, and efficiency of building or updating software using the release management process. It is the process of planning, scheduling, and managing a software build through the stages of developing, testing, deploying and supporting the release [39]. Release management usuallystarts at the first stage of the development cycle, where managers layout a release policy, a document that defines the scope, principles, and end goals for the release management process. Release managers establish release plans based on the release policy. These plans are broad guidelines for delivering releases. Every major release provides a significant amount of new or modified functionalities compared to the previous release and bug fixes. Due to today's competitive business world, software organizations must deliver new features and bug fixes fast to gain and sustain the satisfaction of users [13]. As a result, many large software projects have switched from a more traditional release cycle (e.g., 12–18 months to deliver a major release), to shorter release cycles (e.g., weeks) [27]. Large projects such as Google Chrome, Mozilla Firefox, and Facebook have adopted shorter release cycles [4]. For example, Firefox adopted a shorter release cycle to speed up the delivery of its new features [116]. This has been partly done to compete with Google Chrome's rapid release model. In this dissertation, we focus on the phenomenon of rapid releases which

refer to releases that occur in release cycles that last weeks or days. Rapid release cycles claim various benefits to both companies and end-users. They offer a reduced time-to-market and faster user feedback [16, 79]; end-users benefit because they have faster access to functionality improvements and security updates [80] and improve turnaround time for fixing bad bugs [16]. Despite these benefits, research has shown that rapid releases often come at the expense of reduced software reliability [28, 63] and important repercussions on software quality [80]. Adopting rapid release policies can impact different parts of the software development process, such as the testing and maintenance parts.

1.2 Thesis Statement

Despite that rapid releases are increasingly being adopted in open-source and commercial software, it is not well studied what the effects are of such practice [69]. It is essential to understand its effects on the software process as it forces organizations to change their development process and tools to release their software, in a timely manner and with a good quality [22]. This thesis' overall objective is to empirically understand the short and long-term impact of rapid releases on different parts of the development process of open source software (OSS). Such understanding can provide valuable insights to help OSS organizations to consider what preparation and adjustment to their release management process may be necessary when switching to shorter release cycles. Such insights can also open up research directions that can help open source foundation and commercial organizations; and practitioners on how to plan and adopt rapid releases and how to migrate to a rapid release model properly. It can also bring a better understanding and generalization of release practices. We will, therefore study the following thesis statement in this dissertation:

> By empirically studying rapid releases in large and mature open-source software projects, we provide valuable insights in the advantages and disadvantages of the adoption of rapid releases, and the release management plans and tools needed for successful adoption.

To validate the aforementioned thesis statement, we will rely on a mixed-methods research approach. Mixed methods research is a methodology that consists of collecting, analyzing, and integrating quantitative (e.g., case studies) and qualitative (e.g., surveys and interviews) research [120]. Mixed methods incorporate the strengths of both qualitative and quantitative research methods to overcome the weaknesses and limitations of the particular methods [29, 107]. Usually, the results of the qualitative study are used to support the findings of quantitative study [34]. More specifically, this thesis uses case studies and surveys as two complementary methods to quantitatively and qualitatively assess the usefulness of our research. As part of our case study, we analyzed different types of various kinds of development artifacts such as bug-tracking repositories, testing data commit and release histories; we conducted a survey with Eclipse developers to support our analysis. In each of Chapter 4 and 5, we provide more detail on our case study plan and the data collection process.

1.3 Contribution

The empirical results that will be presented and discussed in this dissertation show the impact of adopting rapid releases, providing software organizations and practitioners with valuable insights on the advantages and disadvantages of rapid releases

and actionable information that help them to make informed decisions on whether and when to safely switch to a rapid release model. Various software repositories, containing different types of data, can help in studying and understanding the evolution and changes of the software development process. Among the available data sources, we decide to focus on source code, testing data and software issues. Source code plays a key role in software maintenance activities. Software issues and testing provide valuable information about software quality. The main contributions of this dissertation are:

- Quantitative analysis: We empirically examine the short-term and long-term impact of the rapid releases on the software maintenance process, specifically the bug handling process (Chapter 4).
- Quantitative analysis: We study the impact of rapid releases on some common software development practices of large open source projects Eclipse and Mozilla Firefox (Chapter 4 & 5).
- Qualitative analysis: We carry out a survey among software maintainers in order to assess the impact of rapid releases on software maintenance (Chapter 4).
- For all case studies, we provide guidelines for future rapid releases adopters on how and whether to switch to a rapid release model (Chapter 4 & 5).

1.4 Dissertation Structure

The dissertation is organized as follows:
After this introduction, chapter 2 presents the terminology and concepts that develop the background required to understand the remainder of the dissertation. We present background information on the software development process and release management and the bug handling process. Chapter 3 reviews the existing literature on rapid releases and, in particular, bug fixing process in rapid releases.
Chapter 4 presents an empirical study of the bug handling activity in the Eclipse core projects where we study the evolution of the bug handling process and the impact of the transition to a rapid release cycle. The content of this chapter is mainly based on our previous publications in the International Conference on Software Maintenance and Evolution 2019 (ICSME) proceedings [2], and in the Journal of Systems and Software 2020 (JSS) [3].
Chapter 5 presents our study on revisiting the impact of rapid releases on the Mozilla Firefox project. We study the impact of switching to more rapid releases on different activities in the software development process, such as the bug handling process and testing activity.
Finally, Chapter 6 concludes this dissertation by summarizing our contributions, limitations and discussing future work.

Chapter 2

Background

This chapter introduces the terminology and concepts necessary to develop the background knowledge that will help the reader to understand the remainder of this thesis. In Section 2.1, we present techniques, tools and practices that are relevant to understand the development process of many software projects, especially the Eclipse and Mozilla Firefox projects. We introduced concepts related to release management and release models in Section 2.2. In Section 2.3, we introduce bug reporting, the bug handling process and explain the Bugzilla tracking system. Finally, Section 2.4 summarizes the chapter.

2.1 Software Development Process

A software development process defines the methods and procedures that organizations and individuals have to follow to create software products and services [98]. Ultimately, the purpose of a software process is to provide a roadmap for the development of high-quality software products that meet the needs of their stakeholders within a balanced schedule and budget [137]. It focuses on the creation and maintenance of tasks rather than the output or the end product. Thus, a typical software process should aim to solve the potential and future problems of software development in terms of planning and budgeting. A broad definition of the software development process includes development, deployment and maintenance of a software product, which should include a set of policies, organizational structures (e.g., task definitions), human activities, technologies, and procedures [43]. Among the many software development approaches proposed in the literature, the most well-known ones are the traditional waterfall model and more recent agile approaches [92].
The four basic software development process activities of specification, development, validation and evolution are organized differently in different development processes [122] (see Fig. 2.1). In the waterfall model, they are sequentially organized, while in evolutionary development, such as agile methods, they are interleaved. The way in which these activities are carried out depends on the type of software, individuals and organizational structures involved.

1. **Software specification** or requirements management is the process of understanding and defining the functional and non-functional requirements required for the system and determining the operational and developmental constraints of the system. The requirements engineering process produces a software requirements document that corresponds to the system specifications.
2. **Software design and implementation** is the process by which a system specification is converted into an executable system by system design. A software design is a description of the architecture of the software to be implemented, the data that is part of the system, the interfaces between the system components and, sometimes, the algorithms used.
3. **Software validation** aims to demonstrate that a system meets its specifications and customer expectations. It involves checking the processes at every stage of the software process. Most validation costs are incurred post-implementation when system operation is tested.
4. **Software evolution**, in particular software maintenance, is the term used in software engineering to refer to the process of developing software initially, then it is repeated updating for various reasons. The goal of software evolution would be to implement potential major changes to the system. The existing system is never complete and keeps evolving [75]. As it evolves, it becomes more and more complex. The primary goals of software evolution are to ensure system reliability and flexibility.

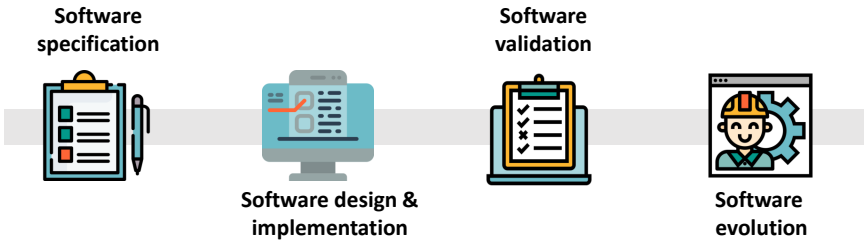


FIGURE 2.1: Software development activities.

We present a few definitions, which help readers understand different artifacts and practices in the software development process:

- **Source code:** computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translators [56].
- **Issue:** uniquely identifiable entry in an issue-tracking system that describes a problem or an enhancement [56].
- **Software feature:** software characteristic specified or implied by requirements documentation [56].
- **Commit:** the smallest increment a developer contributes to the code base of a project, and it is recorded into the version control system [67].
- **Patch:** a modification to a source code to resolve functionality issues, improve security and add new features [105].
- **Channel:** an approach to distributing products and services from the original supplier to the end-user organization [56].
- **Stabilization:** is the process of getting a release branch into a releasable state; that is, of deciding which changes will be in the release, which will not, and shaping the branch content accordingly [42].
- **Feature freeze:** no new functionality can be added [56].

Building and maintaining a high-quality software system is a complex task that requires the cooperation of many members in different software teams. Different individuals have different roles within the project. Individuals may have more than one role (e.g., an individual can be both a reviewer and a committer), but the roles are distinct. Below we describe some roles in developing a software product:

- **Stakeholder:** an individual or organization having a right, share, claim, or interest in a system or in its possession of characteristics that meet their needs and expectations [56].
- **Developer:** an individual that performs development activities (including requirements analysis, design, writing source code and testing) during the software development process [56].
- **Committer:** a developer with commit privileges [56].
- **Maintainer:** an individual that performs maintenance activities [56].
- **Reviewer:** an individual that performs reviewing activities to find and resolve issues in the software [56].
- **Code sheriff:** an individual that aids developers to easily, quickly, and seamlessly land their code in the proper location(s) and ensure that the code does not break the automated tests [118].

- **Contributor:** an individual that contributes code, documentation, tests, ideas to software project [56].
- **End-users:** the personnel who use the system for their specific business purpose [56].

During the development process, software teams will generate a large number of artifacts, such as source code files, documentation, reported errors, technical discussions, etc. Software development teams rely on software tools to manage these artifacts and keep track of their complete change history. Several types of collaborative tools have been developed and widely used by software developers [82]. Example of software tools that are relevant in our work are:

Version Control Systems (VCS) manage the code base of a software system and store every version of the source code. These tools permit developers to work simultaneously on the same code files. They also provide conflict resolution mechanisms when there are overlaps between the changes of developers. Commit is the smallest unit of modification recorded into the VCS. Each commit has a unique identifier, the information of the developer who submitted the commit, the date of submission, the commit message, the list of the affected files, the patch, etc. Examples of version control systems are SVN[1] and Mercurial[2].

Issue Tracking Systems (ITS) allow the recording and tracking of software issues reported by end-users and developers. Also, they enable developers to collaboratively fix the reported issues and keep track of their progress and status. **Issue reports** include feature requests, software bugs, enhancements and refactoring actions. When an issue report represents a software bug, we refer to it as a **bug report**. In the remaining of this thesis, we also refer to issue tracking systems **bug tracking systems**. A bug tracking system is a software tool that keeps track of reported software bugs in software development projects. Examples of bug tracking systems are the open source system like Bugzilla[3] and the commercial system Mantis[4].

2.2 Release Management

Release Management is a fundamental approach for successful open source projects. Release management includes the planning, implementation and monitoring of software development processes. **Release Managers** are individuals who manage the software by administrating the release process, proposing release schedules, and ensuring that new releases progress timely [126]. A **release** is typically a named set of software components and supporting documentation that is subject to change management and is upgraded, maintained and tested as a whole. A release has a **release policy** that defines the methods of how the release will be built, configured, and installed into the production environment [71]. In general, the release management process involves the following steps [15]:

- **Requirements Gathering and Planning:** When a new release is prepared, requirements and improvements needed to fix the current product are gathered.

[1] https://subversion.apache.org/
[2] https://hg.mozilla.org/
[3] https://www.bugzilla.org/
[4] https://www.mantisbt.org/

The plan breaks the release into stages, sets up the overall workflow, and outlines who is responsible for each task.

- **Release Building:** The building process of the new release is started when the release requirements are identified and planned. It consists of the main activities used in the software development process.

- **Acceptance Testing:** The system release build is subjected to a comprehensive testing procedure when it is ready. The program is reviewed to verify if it works correctly and lives up to the requirements and dependencies.

- **Release Preparation:** The verified release is packaged and prepared as a final product to be delivered after checking if it meets the minimum standards of acceptance and business requirements as detailed in the release plan.

- **Release Deployment:** The release is deployed to the customer.

The system requirements evolve over time, so a release plan has to be defined that allows for changes and the gradual introduction of requirements. Release planning steers the project by defining milestones for when new releases have to be delivered. This forces the community to integrate their finalized code into the new releases. We present different types of releases, release channels and strategies to manage the release process in Section 2.2.1, Section 2.2.2 and Section 2.2.3, respectively.

2.2.1 Software Release Basics

In this section, we present the different release types delivered during the software development process. A **build** is an executable code that is handed over to the tester to test the functionality of the developed part of the project. **Version** is an instance of the software product that contains substantial and significant enhancements or other substantial change in functionality or performance compared to the previous version [45]. Whereas a **release** is a version of a software system that is tested and fully functional and has been made available to users for testing or public consumption [56]. **Milestones** are releases that include specific sets of functions and are released as soon as the functionality is complete [132]. **Release engineering** involves building fast and reliable pipelines to transform source code into viable products. Release engineering focuses on the purely technical aspects of the software release. In a nutshell, release engineering deals with the technical aspects of getting a release from development to production [112]. During software development, different types of releases are delivered [106]: There are two main types of release: **internal/external testing releases** and **external product releases**. For instance, the development organization may release a product internally to the testing group or may release a product to the user community for beta testing:

1. **Internal and external testing releases:** The software is tested to ensure that it works as intended before being available for public users. To this end, most organizations use alpha versions, beta versions and release candidates [35]:

   (a) ***Alphas*** are used for internal acceptance testing where developers generally test the fix/change that has been produced.

   (b) ***Betas*** can be considered as a form of external User Acceptance Testing, which is performed by users and communities. It starts when the software is declared to be "feature complete", but may contain some known or unknown issues, such as performance or crashing issues.

(c) A ***Release candidate*** is a version of the software that is considered to be as the final software release. In this phase, no features are developed or enhanced; only issue fixes are allowed.

2. **External product releases:** Most organizations follow three levels of product releases (semantic versioning): **major releases**, **minor releases**, and **patches** [102]. Major and minor releases are designed to be external product releases that are persistent and supported for a period of time.

   (a) A ***major release*** represents the official version of a software product.

   (b) A ***minor release*** is off-schedule and represents the same basic product but with enhanced features, performance, or quality.

   (c) A ***patch release*** corresponds to a developmental configuration that is intended to be transient. It is produced to fix specific bugs or to add minor functionality.

   Other types of external product releases are:

   (d) A ***service release*** (SR) contains bug fixes and may contain a small number of new features or new functionality.

   (e) An ***update*** release is a minor service release and contains only bug or security fixes.

2.2.2 Release Channels

Many modern software projects support pipelined software development which involves teams working in parallel on different release channels (e.g., Chrome and Android Studio) [19, 104]. They use channels to slowly roll out updates to users, starting with daily channel builds (least stable), all the way up to the stable channel releases. The pre-release channels are intended for users who want to experience the latest features and improvements [5]. This gives developers time to test their code on upcoming versions before they are delivered to end-users. The early channels are not recommended for typical end-users because they can have stability issues. Example of the common types of release channels are presented below:

- ***Daily channel:*** This channel is cutting edge as it receives daily updates of new features as soon as they are built. This channel is vulnerable to issues and errors since features are added even before being tested.

- ***Dev channel:*** This channel's releases are selected from older canary releases that have been tested for a while. Like the canary channel, this channel is used to test and show people the newest features as soon as possible. It is still unstable and subject to bugs. Updates are typically released weekly or monthly.

- ***Beta channel:*** If a user is interested in using the new features, with minimal risk, the beta channel is the place to be. Beta channel releases usually contain all the features that a team has decided to add, but it is still expected to have some bugs and performance issues.

- ***Release (or: Stable) channel:*** This channel's releases have been fully tested and are the most reliable. Most users use this channel for production use. Stable releases follow the official release schedule for major version releases.

2.2.3 Release Management Strategies

There are different strategies to manage the release process. Release strategies can be classified as **feature-based**, **time-based** or a **hybrid** of both [88]. Feature-based releases are delivered when a set of predefined goals has been achieved (e.g., when a set of features has been implemented or a set of reported bugs have been fixed). Time-based releases are delivered according to fixed time intervals, with whatever features are ready when the release date comes. A common type of time-based release strategy is the use of **regular cycles** where releases are delivered at regular intervals.
The **release cycle** consists of all the stages from the start of the development of a new version of a product until its release to users, including the release of minor versions containing bug fixes or minor enhancements. When the release strategy is selected, the release cycle time is to be chosen. The **release cycle time** is defined as the time between the start of the development of a software product and its actual release [90]. Michlmayr et al. [90] identified five factors that affect the choice of release interval: regularity and predictability; nature of the project and its users; commercial factors; cost and effort; and network effects.

	Traditional Releases	**Rapid Releases**
Release criteria	feature-based	time-based
Time between releases	a year or more	weeks or few months
New features	next major release	next rapid release
Supported versions	many	one or two
Bug fixes	bug fix release	next rapid release

TABLE 2.1: Comparison between traditional and rapid releases.

A release model is a simplified description of a release process, i.e., of the activities carried out during the release cycle [123]. Table 2.1 highlights the aspects of two release models, traditional releases and rapid releases. Traditional releases follow a feature-based release strategy where every major release is delivered about one year or more after the previous release. Rapid releases follow a time-based release strategy where the time is reduced to a few weeks or months. There is a stabilization period in both release models, which occurs just before the release, it is called **feature freeze**. During this period, all work on adding new features is suspended, moving the effort to test and fix bugs to improve quality and stability. Under traditional releases, new features are implemented for the next release only. However, bug fixes are backported to older releases to guarantee that the old version users will get bug fixes in a timely manner [123]. Under rapid releases, it is usual to support only the latest version since releases are more frequent. The rapid release model allows the organization to deliver new features to users earlier. On the other hand, traditional releases may be appropriate when stability is more important than new features.
The ability to release rapidly was the second most mentioned benefit of agile software development, and it was practiced by 2 out 3 of respondents [16]. It was perceived that rapid releases enable easier progress tracking, easier monitoring of the quality of the software, more rapid feedback, a reduction of turnaround time and hence easier bug fixing. Agile software development highlights the importance of rapid releases, as one of its principles states "Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale" [14]. Rapid releases are claimed to offer a reduced time-to-market and faster user feedback [63]; end-users benefit from faster access to functionality improvements and security updates [80]; and improve turnaround time for fixing bad bugs [16]. Shorter time-to-market

allows projects that adopt rapid releases can compete better with other projects and respond faster to customers' changing needs [68]. Michlmayr and Fitzgerald [89] found that a reduced release cycle time can attract collaborators where the majority are volunteers motivated by the challenges associated with shorter release activities. However, Kerzazi and Khomh [60] stated that a release process needs to be automated and optimized in order to support rapid release cycles. Therefore, rapid release cycles are often adopted through modern release engineering practices such as Continuous Integration (CI), Continuous DElivery (CDE) and Continuous Deployment (CD) as they are an enabler for rapid releases:

Continuous Integration (CI) is a development practice that requires developers to integrate code into a shared repository several times a day. This practice aims to test the code frequently throughout the development process so that possible issues are identified and corrected early.

Continuous Delivery (CDE) is an extension of CI where developed code is continuously delivered as soon as it is considered ready for being shipped. It involves continuous integration, automation testing and deployment of automation processes, which allow rapid and reliable development and delivery of software with the least manual overhead.

Continuous Deployment (CD) is the next logical step after continuous delivery. It is the process of deploying the code directly to the production stage as soon as it is developed. In CD, all changes that pass the automated tests are automatically deployed to the production stage.

Organizations improve the quality, speed, and efficiency of building or updating software by focusing on release management. Release management is planning, scheduling, and managing a software build through the stages of developing, testing, deploying and supporting the release. This section presents the definition of release management, different types of releases, release channels, and strategies to manage the release process.

2.3 Bug Handling

Software development teams use specific processes to support their daily activities such as coding, testing, bug fixing, issue tracking, code reviewing and continuous integration. For each of these activities, dedicated tools and processes are being used that may vary from one development community to another, as well as from one period to another. A typical example of this is the well-known bug handling activity that is an essential activity to ensure software quality. It is estimated that 80% of the software development effort is spent on software maintenance [100]. In recent years, software maintenance has become more challenging due to the increasing number of bugs in large-scale and complex software programs [141]. There are four types of maintenance: corrective, adaptive, perfective, and preventive. Bug fixing is a corrective maintenance action. A well-structured bug management process makes life easy for software practitioners to deal better with software bugs. In Section 2.3.1, we present a few definitions. In Section 2.3.2, we define what a bug report. In Section 2.3.3, we discuss different phases in the life cycle of a software bug. In Section 2.3.4, we present the Bugzilla tracking system. In our analysis, in Chapter 4 and 5, we relied on data retrieved from Bugzilla.

2.3.1 Definitions

In this section, we present a few definitions, which will help readers understand the following sections:

Defect: An imperfection or deficiency in a work product where that work product does not conform to its specifications and needs to be repaired or replaced [56].

Error: The difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition [56].

Fault: A manifestation of an error in the software. An incorrect step, process, or data definition in a computer program [56].

Failure: The inability of a system to perform a required function or its inability to perform within previously specified limits; an externally visible deviation from the system's specification [56].

The use of these terms differs with respect to each organization or software project. Therefore, in this dissertation, we use the common term "bug" as an umbrella word because of its intuitiveness and widespread usage, with the following definition:

Bug: A deviation between the expected behavior of program execution and what actually happened [91].

2.3.2 Bug Report

A bug report contains various fields to describe what is relevant to a specific bug. This information is important for getting a rough idea of what the bug is about. However, not all the fields are compulsory. A description of some fields of a bug report are summarised in Table 2.2.

Field	Description
ID	The bug ID.
Description	Descriptive information that helps in the debugging process such as the reported error message, the steps to reproduce the error, etc.
Product	The product which is affected by the bug
Reporter	The person who reported the bug.
Creation date	The date and time of bug creation.
Assignee	The person in charge of resolving the bug.
Version	The version of the software the bug was found in (e.g. 3.2, 4.5).
Severity	The estimated bug impact as perceived by the bug reporter (e.g., enhancement, trivial, minor, normal, major, critical, blocker).
Status	It defines exactly what state the bug is in (e.g., new, fixed and closed).
Resolution	The latest resolution status of a resolved bug (e.g., fixed, duplicate).

TABLE 2.2: Typical fields that can be found in bug report.

Bug reports are filed on the bug tracking system. Fig. 2.2 shows an example of Eclipse project bug report on Bugzilla. The pre-defined fields of a bug report provide a variety of categorical data about the report. Field values such as the report identifier number, creation date, and reporter, are set when the report is created. Other values, such as the product, component, version, priority, and severity are filled out by the reporter when the bug is created, but the values might change over the lifetime of the report. Other fields may also change over time, such as the person to whom the bug is assigned, or the current status of the report. Also, the contact information of the people involved in the activity of the bug is included. These variable fields

represent the different ways that a bug report can be categorized in the development process. The free-form text contains the report title, a complete bug summary, and additional comments. Typically, the bug description contains a detailed description of the bug's effects and any necessary details for the bug to be replicated by a developer. The additional comments include discussions on potential approaches to fixing the bug and references to other bugs that include or appear to be duplicate reports of additional details about the bug. Reporters and developers can provide attachments to reports to provide non-textual additional information. The activity log provides a historical record of how the report has changed over time, for example, when the report has been reassigned, or when its priority has been changed.

FIGURE 2.2: Example of bug report on Bugzilla tracking tool (bug ID=510161).

2.3.3 Bug Handling Process

Software bugs are introduced in software during the different phases of the software development process. The goal of bug handling process is to determine these bugs in the software and to fix them. The bug handling process is shown in Figure 2.3. During the this process, a bug undergoes three phases to be fixed:

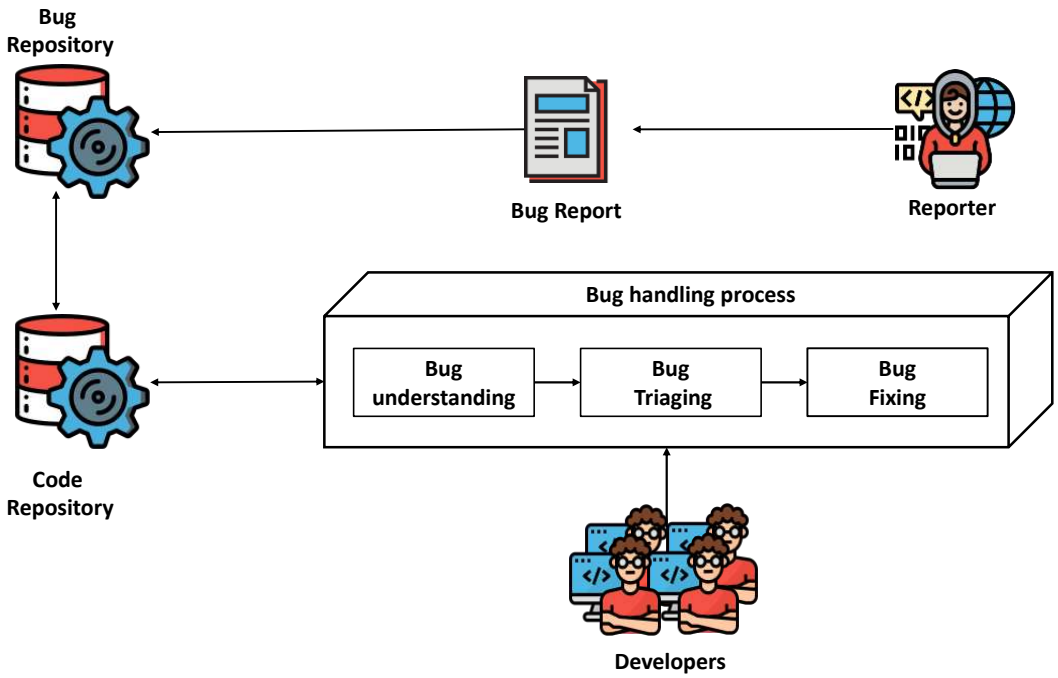


FIGURE 2.3: Bug Handling Process.

1. **Bug understanding:** The purpose of this task is to understand the content of a given bug report to describe the important contents, filter the duplicates and predict the characteristics (e.g., priority, severity and status) of reported bugs. The challenge occurs when a huge amount of bug reports is submitted every day. It can lengthen the fixing time and affect the quality of this task execution.

2. **Bug triaging:** It is concerned with the assignment of bugs to the appropriate developers for fixing. The developer responsible for this activity is called triager. Triagers are supposed to assign the appropriate developers to fix the given bugs and then mark the status of the corresponding bug report to 'Assigned'. Triagers might assign improper developers to do bug fixing tasks if there was no good understanding of the bug or poor knowledge of developer skills. This will lead to the bug re-assignment(s) (i.e., bug tossing). Several automatic bug triaging approaches have been proposed to recommend the best developers for fixing a bug [7, 8, 119, 136].

3. **Bug fixing:** The assigned developer is responsible for fixing the reported bugs. The developer needs to find the source code files where the bug is and develop or update code as a part of the bug-fixing process. Manual bug localization is time-consuming and expensive. Thus, methods for automatically locating bugs from bug reports are highly desirable. Therefore, researchers have developed automatic tools to locate the given bugs [65, 111, 133, 134].

To facilitate this bug handling process, developers rely on dedicated collaborative bug tracking tools. Our empirical analyses in Chapters 4 and 5 are based on data extracted from Bugzilla. In the following section, we introduce Bugzilla and its lifecycle.

2.3.4 Bugzilla Issue Tracking System

Bugzilla is one of the most well-known open-source bug tracking tools. Bugzilla tends to follow a dedicated process to support bug fixing. Our case studies in Chapter 4 and 5 use Bugzilla as their bug tracking system. Figure 2.4 shows the bug life cycle supported by Bugzilla. A bug's life starts with the `UNCONFIRMED` status. Each bug has a unique numeric identifier assigned to it, called the bug ID, after reporting. Each bug has associated a set of attributes such as severity, priority, version, when it was discovered, who reported it and so on. Once the new bug is confirmed as a real problem its status changes to `NEW`. The status of a bug becomes `ASSIGNED` when a developer takes charge of fixing the bug. Once a bug resolution is carried out, the bug status changes into `RESOLVED`. The *Resolution* takes values such as `FIXED`, `INVALID`, `WONTFIX`, `WORKSFORME`, `DUPLICATE` and `MOVED`. Bug resolutions are subject to the scrutiny of a Quality Assurance (QA) team. `VERIFIED` bugs are bugs that have an approved resolution. Eventually, a bug reaches the `CLOSED` status. As shown in Figure 2.4, loops may exist and a bug's life can be much more complex. For example, a bug can be `REOPENED` because it needs more work.

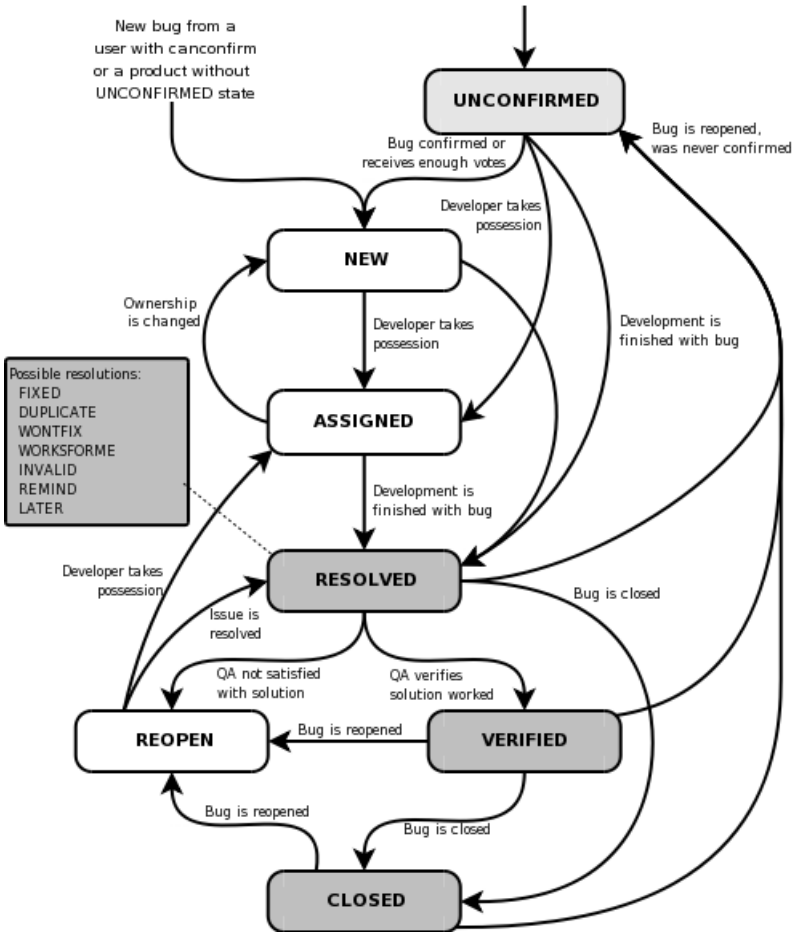


FIGURE 2.4: Example of the Bugzilla 3.6 bug life cycle.

Each bug report is accompanied by a *history* containing all events that occurred during the bug's life cycle (e.g., a reporter created the bug, the bug was assigned to someone, the status and resolution type of a bug were changed, etc.). Table 2.3 provides a fictitious example of what a bug report history on Bugzilla typically looks like. The first column shows the email of the person *Who* performed the change, the

second column *When* this change occurred, the third column *What* was changed, and the last two columns show the old value that was *Removed* and the new value that was *Added* to replace the old value. For instance, the first modification was performed by Donald at 9.23 AM EDT on the 4th of October 2016. Donald changed the value of the *Assignee* field to daisy@domain.org and the status value from `NEW` to `ASSIGNED`.

Who	**When**	**What**	**Removed**	**Added**
donald@domain.org	2016-10-04	cc		dagobert@domain.org
	09:23:28	Assignee		daisy@domain.org
		Status	`NEW`	`ASSIGNED`
daisy@domain.org	2016-10-06	Version	4.5	4.5.1
	11:02:01	Attachment #111253	0	1
daisy@domain.org	2016-10-25	Severity	normal	critical
	11:05:34	Status	`ASSIGNED`	`RESOLVED`
		Resolution	—	`FIXED`

Table 2.3: Fictitious example of bug report history in Bugzilla.

2.4 Summary

In this chapter, we have introduced the terminologies and preliminary concepts that help in forming the foundation to understand the thesis. At the beginning of the chapter, we have presented the software development process and different terminologies of software engineering. Then, we have presented different strategies of release management. Finally, we have introduced the anatomy of the bug report and its life cycle.

Chapter 3

Related Work

Due to today's competitive business world, software organizations must deliver new features and bug fixes fast to gain and sustain the satisfaction of users [13]. In light of this, many large software projects switch from a more traditional release cycle to rapid release cycles. This thesis's aims to empirically understand the impact of rapid releases on different parts of the software development process.
In this chapter, we present a non-exhaustive summary of the related research to this thesis. The chapter is divided into two sections. In the first section, we discuss some closely related research focusing on different activities of the bug handling process. The second section addresses the work on rapid releases and the bug handling process in relation to rapid releases.

3.1 Bug Handling Process

3.1.1 Bug triaging

Saha et al. [110] extracted code change metrics, such as the number of changed files, to identify the reasons for delays in bug fixes, and to improve the overall bug fixing process in four Eclipse Core projects: JDT, CDT, Plug-in Development Environment (PDE), and Platform. Their results showed that a significant number of long-lived bugs could be reduced through careful triaging and prioritization if developers could predict their severity, change effort, and change impact in advance.
Zhang et al. [140] studied factors affecting delays incurred by developers in bug fixing time. They analyzed three Eclipse projects: Mylyn, Platform and PDE. They found that metrics such as severity, operating system, description of the issue and comments are likely to impact the delay in starting to address and resolve the issue.
Hooimeijer and Weimer [55] analyzed the correlation between bug triaging time and the reputation of a bug reporter. They designed a model that uses bug report fields, such as bug severity and submitter's reputation, to predict whether a bug report will be triaged within a given amount of time in the Mozilla Firefox project.

3.1.2 Bug Resolution

Panjer [99] carried out a case study on Eclipse projects and showed that the most influential factors affecting bug fixing time are found in initial bug report fields (e.g., severity, product, component and version) and post-submission information (e.g., comments).
Giger et al. [44] found that the assigned developer, bug reporter and month when the bug was reported to have the strongest influence on the bug fixing time in Eclipse, Mozilla and Gnome. Marks et al. [81] studied different features of a bug report in relation to bug fixing time using bug repository data from Mozilla and Eclipse. The most influential factors on bug fixing time were bug location and bug reporting time.
Zou et al. [143] investigated the characteristics of bug fixing rate and studied the impact of a reporter's different contribution behaviors on the bug fixing rate in Eclipse and Mozilla. Among others, they observed an increase in fixing rate over the years for both projects. On the other hand, the observed rates were not high, especially for Mozilla.
Rwemalika et al. [108] studied the characteristics and differences between pre-release bugs and post-release bugs in 37 industrial Java projects. They found that post-release bugs are more complex to fix since they require modification of several source code files, written in different programming languages and configuration files.
Lamkanfi et al. [73] proposed a dataset that provides a comprehensive information bundle on the historical evolution of the most relevant attributes from the Eclipse and Mozilla project bug reports. Such dataset motivates the bug analysis and the reproducibility and comparison of the bug detection models in Eclipse and Mozilla.
Lamkanfi and Demeyer [72] observed that open source data on bug resolution times could be heavily distorted and include nonrealistic data with resolution times of less than a minute. They found that such outliers may confuse data mining techniques and produce distorted results. Thus, removing such data would improve the result of the model.
Athanasiou et al. [9] proposed a model using source code metrics to assess test code quality. They calibrated that model with 86 open source and commercial Java systems for the ratings of a system's test code to reflect its quality compared to those

systems. They showed that there is a high correlation between test code quality and throughput and productivity of issue handling.

All these studies are valuable in understanding the overall bug fixing process, factors affecting bug fixing time, bug fixing time estimation and triaging automation. In our study in Chapter 4, we focus on the bug triaging and fixing time and bug resolution and fixing rate. We are not aware of any related study comparing these metrics before and after the release is delivered and how they evolve over successive releases considering the bug severity level. Moreover, we study the impact of feature freeze periods on the bug handling process.

3.2 Rapid Releases

3.2.1 Benefits and Challenges of Adopting Rapid Releases

According to a recent literature review by Mäntylä et al. [80], rapid releases are followed in multiple domains including finance, automotive, telecom, space and energy. Prior research studied the benefits and challenges of adopting rapid releases. Such releases are claimed to offer reduced time-to-market and faster user feedback [63]. End-users may benefit from this because they get faster access to functionality improvements and security updates [80]. Moreover, Zimmermann [142] has shown that the adoption of a shorter release cycle has successfully managed to provide more stable versions, with less breaking changes, that are easier to upgrade. In our study in Chapters 4 and 5, we will analyze the impact of rapid releases on the bug handling process.

Joshi et al. [57] introduced a publicly available dataset consisting of 994 open source projects on GitHub featuring rapid releases. This dataset and its documentation and scripts aim to facilitate future empirical research in release engineering and agile software development.

Kerzazi and Khomh [60] examined over 14 months of release data from 246 rapid Firefox releases to determine the several types of factors that affect release time. They identified three factors: technical factors (e.g., code merging and integration and automated tests), organizational factors (e.g., design and management of branches and release planning) and interactional factors (e.g., coordination policies amongst teams). Their analysis reveals that testing is the most time-consuming activity in the release process (86%). Their analysis shows that the time spent on code merging, stabilizing and packaging activities account for only 6%, 6% and 2% of the cycle time, respectively. Moreover, they note that a lack of socio-technical congruence among teams can delay releases. Socio-technical congruence refers to the alignment between the technical dimension of work and the social relationship between team members [23].

Maleknaz et al. [96] analyzed (among others) the release cycle times of 6,003 mobile apps on Google Play as a *treatment* to predict as an *outcome* the customer satisfaction expressed through an app rating. To do so, they introduced a generic analytical approach called the Gandhi-Washington Method (GWM). For the specific scenario of mobile app rating, the method consists of encoding and summarising the sequence of release cycle times of each app using regular expressions over the alphabet S (short release cycle), M (medium release cycle), L (long release cycle); followed by statistical tests over those generated expressions to determine causal effects on the outcome

variable. They found that apps with sequences of long releases followed by sequences of short releases have the highest median app rating. Apps with sequences of long followed by sequences of medium releases get a lower median rating. Finally, apps with sequences of long releases exclusively get the lowest median rating.
Castelluccio et al. [22] investigated the reliability of the Mozilla uplift process. Patch uplift is the practice where patches that fix critical issues or implement high-value features are promoted directly from the development channel to a stabilization channel because they cannot wait for the next release. This practice is risky because the time allowed for the stabilization of the uplifted patches is short. The authors examined patch uplift operations in rapid release pipelines and formulated recommendations to improve their reliability. They investigated the decision making process of patch uplift at Mozilla and observed that release managers are more inclined to accept patch uplift requests that concern certain specific components and/or that are submitted by certain specific developers.

3.2.2 Rapid Releases and Software Quality

Khomh et al. [63] empirically studied the effect of rapid releases on Mozilla Firefox software quality. They quantified quality in terms of runtime failures, the presence of bugs and outdatedness of used releases. Related to our work, they compared the number of reported, fixed and unconfirmed bugs and the fixing time during both the testing period, i.e., the time between the first alpha version and the release, and the post-release period. They found that fewer bugs are fixed during the testing period and that bugs are fixed faster under a rapid release model. In a follow-up work [64], the authors reported that, although post-release bugs are fixed faster in a shorter release cycle, a smaller proportion of bugs is fixed compared to the traditional release model. Interviews conducted with six Mozilla employees revealed that they could be “less effective at triaging bugs with the rapid release" and that more beta testers using the rapid releases can generate more bugs.
Da Costa et al. [28] studied the impact of Mozilla’s rapid release cycles on the integration delay of addressed issues. They showed that the rapid release model does not integrate addressed issues more quickly into consumer-visible releases compared to the traditional release model. They also found that issues are triaged and fixed faster in rapid releases. In a follow-up work [27] they reported that triaging time is not significantly different among the traditional and rapid releases.
Baysal et al. [12] found that bugs were fixed faster in versions of Firefox using a traditional release model than in Chrome rapid releases, but this was not statistically significant.
Clark et al. [24] empirically analyzed the security vulnerabilities in Firefox and found that a rapid release cycle does not result in higher vulnerability rates. The authors also found that frequent releases increase the time needed by attackers to learn the software code, in contrary to the popular belief that frequent code changes result in less secure software.
Mäntylä et al. [80] analyzed the impact on software testing efforts when switching to rapid releases in a case study for Firefox. They found that tests have a smaller scope and are performed more continuously with less coverage under rapid releases. They also found that the number of testers decreased in rapid releases, which increased the test workload.

In the literature, several benefits of rapid releases were mentioned: rapid feedback, improved quality focus of the developers and testers, easier monitoring of progress and quality, customer satisfaction, shorter time-to-market, and increased efficiency due to increased time-pressure. Despite these benefits, previous research has shown that rapid releases often come at the expense of reduced software reliability, longer integration delays, accumulated technical debt and increased time pressure. Most of the studies done on rapid release were performed on the Firefox project which switches its release cycle in 2012 from 12-18 months to 6-weeks release. However, in our study in Chapter 5, we revisit some of the previous work, such as [64] and [80] to study the effect of rapid releases on software quality and testing. However, we study these activities before and after the recent changes in the release policies(i.e., the removal of *Aurora* and the switch to 4-weeks cycle).

3.3 Summary

In this chapter, we have reviewed previous research that has inspired much of the work presented within this dissertation. We covered two main areas: the bug handling process and rapid releases. We first reviewed the research from the area of the bug handling process, including previous research efforts related to bug fixing and bug triaging. We then reviewed the related work on Rapid releases and its impact on the different activities in the software development process.

Chapter 4

Impact of Release Policies on Bug Handing Activity

Software organizations must deliver new features and bug fixes fast to gain and sustain the satisfaction of users due to today's competitive business world [13]. As a result, many large software projects have adopted rapid releases policies. This can impact different parts of the software development process. Software maintenance is one of the significant activity in the software development process. Rapid releases might impact software maintenance activity as there is less time to handle bugs.

This chapter presents an empirical study of the impact of adopting rapid releases on the bug handling process. Given that releases are delivered more often, and the community may have less time to address unresolved bugs.

The content of this chapter is mainly based on our previous publication in the International Conference on Software Maintenance and Evolution 2019 (ICSME) [2] and an article in the Journal of Systems and Software 2020.

4.1 Introduction

Continuous software engineering is a common practice for large collaborative software projects [18]. Every major release provides a significant amount of new or modified functionalities compared to the previous release. Developers strive to resolve as many bugs as possible before the next release deadline [97]. More and more large software projects are switching to a rapid release cycle [28] to reduce their time-to-market [131]. Rapid release policies might, however, negatively affect the number of bugs being handled, since releases are delivered more often, and the community may have less time to address unresolved bugs. A good preparation is, therefore, of paramount importance to increase the success of adopting a rapid release policy.
Another common practice of large software projects is to impose a *feature freeze* when approaching the next release deadline [41, 87]. During the feature freeze period that lasts until the release date, all work on adding new features is suspended, shifting the effort towards fixing bugs, and carrying out a series of test-and-fix iterations to improve quality and stability. Each such iteration results in a new so-called *release candidate*.
Following the Goal-Question-Metrics (GQM) approach [130], we study the evolution of Eclipse – a large and long-lived open source project – with the aim to analyse its bug handling process for the purpose of assessing the impact of rapid releases and feature freeze periods from the point of view of maintainers in the context of bug handling. Software maintenance is part of the software development process. In maintenance, bug fixing comes as a priority to run the software. This chapter discusses the possible advantages and disadvantages of the rapid release on the bug handling activity. This overall objective is divided into two main goals, each composed of two research questions that will guide the case study design and empirical analysis:

Goal 1: The first research goal aims to study if the transition to rapid releases resulted in less time for the community to handle bugs before the release, leading to potentially more post-release bugs in need of resolution. To do so, we analyse if the bug handling activity is different before and after each release, and whether and how this changed after the transition to rapid releases. Our investigation will be guided by two research questions:

- **RQ1.1:** ***How does the bug handling rate evolve across releases?*** We empirically analyse if the bug handling rate increases for successive releases. We also investigate the bug handling rate before and after each release, as well as if any notable difference could be observed for the rapid releases.
- **RQ1.2:** ***How does the bug handling time differ across releases?*** Since maintainers strive to deliver project releases with as few bugs as possible, we study whether bug handling activity *before* an upcoming release leads to faster bug triaging and fixing times than *after* the release. We thus investigate the differences between these two periods in terms of days elapsed to triage and fix bugs.

Goal 2: The second research goal aims to study to which extent bug handling activity is affected by the presence of feature freezes, and whether the transition to rapid releases has lead to an observable difference as their shorter duration can potentially affect the bug handling activity. Our investigation will be guided by two research questions:

RQ2.1: ***How does the feature freeze period impact bug handling rate?*** This research question analyses the bug handling rate before and during the feature freeze period of each considered release, the effort spent in these periods, and whether all of this has changed for rapid releases.

RQ2.2: ***How does the feature freeze period impact bug handling time?*** This research question focuses on bug handling time before and during the feature freeze period of each considered release. We study whether bugs are indeed triaged and fixed faster during such feature freeze periods. We also study whether the triaging and fixing time of bugs targeting the current release increases during these periods, since maintainers may prefer to focus on bugs of the upcoming release rather than on those targeting the current release that has already been delivered.

This chapter presents our longitudinal case study of the bug handling process for the Eclipse Core projects over a 15-year period. We analyse 17 consecutive major releases (from 3.0 till 4.10) and study how the bug handling time and rate differ before and after an upcoming scheduled release; and how it evolves over time. We rely on four measurements to quantify bug handling: triaging and fixing time and resolution and fixing rate. We focus on a specific and important aspect in the evolution of Eclipse Core, namely how the transition to a rapid release policy during the 4.x series has influenced the bug handling process. To do so, we compare the bug handling activity of seven annual releases (4.2 – 4.8) against seven quarterly releases (4.9 – 4.15), based on a dataset of over 36K bug reports from Bugzilla. We follow a mixed-method approach, by quantitatively analyzing the impact of rapid releases on bug handling activity, and supporting this analysis with qualitative anecdotal evidence about the perceived benefit of the transition by consulting five Eclipse Core maintainers. In addition, we analyze the effect of the feature freeze period of each release on bug handling, and whether and how this has changed after the adoption of a rapid release policy. We additionally study to which extent the bug severity plays a role in each research question as more severe bugs can be expected to be prioritized and thus handled more quickly [74]. This effect might be reduced in rapid releases where there may not be enough time to handle all pending bugs, potentially leading to an increased backlog for less severe bugs.

4.2 Methodology

This section presents the setup of our empirical study, the selected case study, the data extraction process, the metrics that will be used to answer the research questions, and the process we have followed to receive qualitative feedback from five Eclipse maintainers. The dataset and scripts used for the current study are publicly available in a replication package on Zenodo [62].

4.2.1 Selected Case Study

For our empirical study we have selected Eclipse as a case study because it is a long-lived open source ecosystem, with a large community of contributors. It has fixed durations for both annual or rapid releases, allowing to make fair comparisons across successive releases. More importantly, during its 4.x release series it has switched in 2018 from a yearly to a quarterly (i.e., every 13 weeks) release policy.
Eclipse is an open source ecosystem that has been widely studied in software evolution research [21, 84] and in research about bug handling in particular [70, 99]. Eclipse is

a large and mature project with a long release history and associated bug reporting activity hosted through the Bugzilla bug tracking platform [37].
The development of Eclipse is subdivided into the Core projects and the plugins. Eclipse 3.0 is the first release coordinated and shipped by the independent Eclipse Foundation; previous releases were coordinated and controlled by IBM. The Core projects have a stable development community and a regular release process. We focus only on bugs related to the Core projects, since the plugins do not follow the same regular release policy and their bug handling activity can be affected by the absence of factors such as continuous bug monitoring and continuity of the plugin development. The Core projects of Eclipse are *Platform*, *JDT*, *Equinox* and *PDE* [48]. We do not consider the *e4* and *Incubator* projects in our analysis because: (i) *e4* is an incubator for community exploration of future technologies of Eclipse and uses a different versioning scheme than the other projects; (ii) *Incubator* contained only very few reported bugs (49), all with an unspecified version of the Eclipse release.
In our preliminary analysis in Section 4.7, we start from release 3.0 to study how the Eclipse Core projects have evolved over time. The main research goals and associated research questions that will be explored in Section 4.8 (RQ1.1, RQ1.2) and Section 4.9 (RQ2.1, RQ2.2) focus on the 4.x release series, starting with release 4.2 in June 2012. We exclude the 3.x series because it is not able to provide us any actionable results given that it concerns old data based on a different bug handling process. Indeed, the analysis presented in Section 4.7 has revealed that bug handling metrics values for the 3.x series are quite different from the recent 4.x series, implying that the way in which bugs were handled during the 3.x series do not reflect the current practices of Eclipse Core maintainers.
The 4.x release series starts with release 4.2 in June 2012. Until release 4.8 in June 2018, Eclipse followed an **annual** "simultaneous release" scheme[1] where in June a new release was delivered simultaneously for every Core project. Each release until 4.5 was followed by two "service releases" in September and February, respectively. Releases 4.6 and 4.7 had three "update releases" in September, December and March. Since release 4.9, Eclipse has switched from an annual to a **quarterly** release policy (i.e., every 13 weeks) without intermediate update releases.

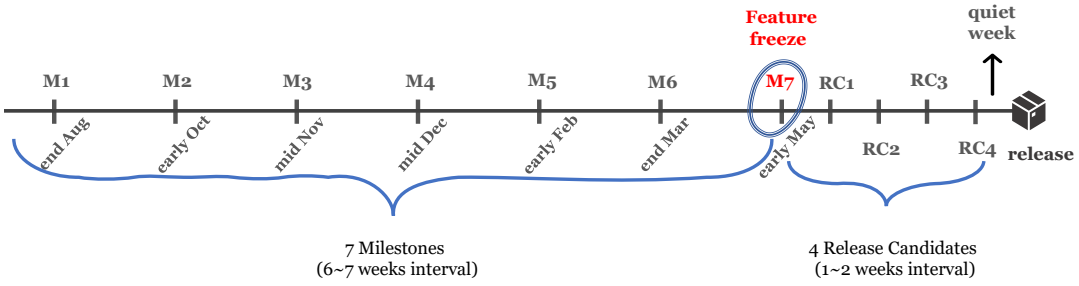


FIGURE 4.1: Annual release schedule for releases 4.2 till 4.8 of Eclipse Core projects.

The Eclipse Core project release schedule is composed of several successive milestone builds (M_1, M_2, etc.), followed by a *feature freeze* period after the last milestone, during which maintainers are no longer allowed to introduce new features, and have to concentrate instead on fixing bugs for the release under development. During

[1]This terminology is used by the Eclipse foundation to reflect a coordinated release effort including the Eclipse Platform and other Eclipse projects. See https://wiki.eclipse.org/Simultaneous_Release.

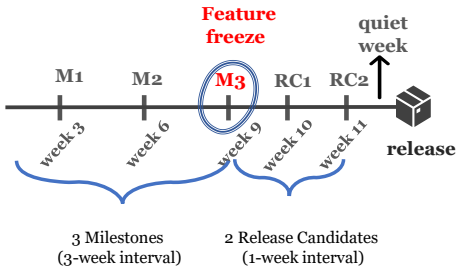


FIGURE 4.2: Quarterly release schedule for releases of Eclipse Core projects since 4.9.

this freeze period, a number of successive release candidates (RC_1, RC_2, etc.) are created. Fig. 4.1 shows the annual release schedule, including 7 milestones occurring at roughly 6-week intervals, followed by 4 release candidates during the feature freeze period. The quarterly release schedule (Fig. 4.2) is composed of 3 milestones occurring at 3-week intervals, followed by 2 weekly release candidates during the feature freeze period. In both annual and quarterly releases, the last week before a release is always a quiet week during which there are no further builds. This week is reserved for final in-depth testing and preparation for the release.

4.2.2 Extracting and Processing Bug Report Data

Our empirical analysis is based on bug report data extracted from Bugzilla for each release of each Eclipse Core project. A typical bug report contains a wide variety of fields. A description of those fields that are relevant in the context of our empirical analysis is summarised in Table 2.2.

The *Severity* of bugs is reported by their owners based on their personal perspective. The Eclipse community considers seven levels [2]: enhancement, trivial, minor, normal, major, critical and blocker. We excluded 28,579 bugs marked as enhancement from our analysis since feature enhancements are considered to be new functionality requests rather than bugs [109].

To extract the bug histories of all reported Eclipse bugs we used the Bugzilla API[3]. Since we focus on Eclipse Core projects only, we only considered bug reports for which the *Product* field was tagged with *Platform*, *JDT*, *Equinox* or *PDE*. We extracted 215,591 bug histories corresponding to these projects. Our dataset was fetched on 27 July 2020, and the earliest and latest dates of reported bugs in our dataset correspond to 11 October 2001 and 27 July 2020, respectively.

In a next step, we filtered bugs based on their *Version* field. As our goal is to study the bug resolution process in relation to each Eclipse release, we only considered those bug reports whose version corresponds to an actual Eclipse release ranging between 3.0 and 4.15. To this end, we excluded 3,296 bugs with unspecified *Version* field and 33,701 bugs corresponding to versions outside of the specified version range. We restricted ourselves to values that actually correspond to valid Eclipse releases; e.g., the valid version values of the 4.7 release found in our dataset were 4.7, 4.7.1, 4.7.1a, 4.7.2, 4.7.3 and 4.7.0 Oxygen. From the remaining bugs, we excluded 2,569 bugs that corresponded to versions that are not listed in the official releases of Eclipse, i.e., 4.0 and 4.1. Our final dataset consists of 143,606 bug reports, of which 107,397 (i.e., 74.8%) belonging to the 3.x version range, and 36,209 (i.e., 25.2%) belonging to the

[2] https://wiki.eclipse.org/Eclipse/Bug_Tracking#Severity
[3] https://bugzilla.readthedocs.io/en/latest/api/core/v1/bug.html#search-bugs

4.x version range. While the analysis in Section 4.7 focuses on all these bugs, the research questions in Section 4.8 and Section 4.9 focus only on the 4.x release range during which the transition to a faster release cycle took place. Within this range, there are 29,831 bug reports for annual releases (4.2→4.8) and 6,378 bug reports for quarterly releases (4.9→4.15).

For the remainder of the analysis, we partitioned all reported bugs into groups according to their major release number. For example, group 4.2 contains all reported bugs whose *Version* field prefix is 4.2. The aforementioned processing steps mitigate several threats that could bias the results of our study.

As pointed out by Tu et al. [127], incorrect use of bug tracking data may threaten the validity of research findings because the values of bug report fields (e.g., *Version*, *Status*, *Severity*) may change over time. They recommend researchers who rely on such data to mitigate data leakage risks by fully understanding their application scenarios, the origin and change of the data, and the influential bug report fields. We therefore assessed this threat for the bug report fields *Version* and *Status* in Eclipse.[4] Examining the bugs that changed their *Version* field during the bug fixing cycle, we found 1,437 bugs that were reassigned to different releases throughout their history, out of which 1,291 bugs that were reassigned to different major releases. We handle such bugs by considering them only for the last major release they affected as including the same bugs in multiple releases would bias the results for our pre-release analysis. From these bugs, only 21 out of 1,233 `RESOLVED` bugs are resolved in multiple major Eclipse releases, thus the impact on our analysis is minimal. In all our research questions, we consider the impact of bug severity on prioritization of bugs. We used the categorization strategy of Gomes et al. [47] to aggregate bugs into two groups: *severe* (including blocker, critical, and major severity) and *non-severe* (including normal, minor and trivial severity). The threats related to changes in the bug severity field during the bug history was examined, and we found 2,290 bug reports that changed their severity over time, out of which 1,503 bugs being reassigned to different severity category. In those cases, we used the latest severity category assigned to each bug, as changes in the severity level indicate that prior severity levels were not accurate.

A reported bug is considered as resolved if somewhere in the bug history the *Status* field is changed to `RESOLVED`. The corresponding resolution date can be found in the *When* column of the bug history. The *Resolution* field allows to mark a `RESOLVED` bug as `FIXED` and the fixing date can be found in the *When* column of the bug history. Since the value of the *Status* field can be modified multiple times, we register the *last* date that the bug was marked as `RESOLVED`. We opt for this strategy as the presence of multiple resolutions of the same bug implies that resolutions prior to the last one were not satisfactory.

Similarly, the *Status* field allows to mark a reported bug as `ASSIGNED`, and the assignment date can be found through the *When* column of the bug history. In case of multiple reassignments of a bug to different developers, we register the *first* assignment date, reflecting the moment when the bug was triaged for the first time. Note that the Eclipse community has been assigning bugs using an alternative process since 2009[5]: it is possible to use an *assigned to* task without using the *Status* field. This assignment method always assigns bugs to an email address in the form `[component_name]-triaged@eclipse.org`. We identified and marked as `ASSIGNED` 5,449 bugs corresponding to this case.

[4] For the *Severity* field we already discussed earlier in this subsection how we coped with this threat.

[5] https://wiki.eclipse.org/Platform_UI/Bug_Triage

4.2.3 Proposed Bug Handling Metrics

This section introduces all formal notations and metrics that are needed to address the different research questions presented in Section 4.1. We introduce the following notations to refer to the sets of bugs considered during our analysis:

- B_{report} is the set of all reported bugs
- $B_{\text{assign}} \subseteq B_{reported}$ is the set of all `ASSIGNED` bugs
- $B_{\text{resolve}} \subseteq B_{reported}$ is the set of all `RESOLVED` bugs
- $B_{\text{fix}} \subseteq B_{resolve}$ is the set of all `FIXED` bugs
- The superscript notation B^r allows to constrain these sets to bugs targeting a specific release r. For example, $B^{4.7}_{\text{resolve}}$ contains all ***`RESOLVED`*** bugs for which the *Version* field refers to release `4.7`.

Based on these sets, we define functions returning the dates corresponding to specific activities in the history of a given bug report:

- $D_{\text{report}} : B_{\text{report}} \rightarrow Date$ returns the creation date of a bug report.
- $D_{\text{assign}} : B_{\text{assign}} \rightarrow Date$ returns the *first* date the bug report *Status* field has been set to `ASSIGNED`.
- $D_{\text{resolve}} : B_{\text{resolve}} \rightarrow Date$ returns the *last* date the bug report *Status* field has been set to `RESOLVED`.
- $D_{\text{fix}} : B_{\text{fix}} \rightarrow Date$ returns the *last* date the bug report *Status* field has been set to `RESOLVED` with value `FIXED` for the *Resolution* field.

Using the above sets and functions, we define four metrics that will be used to answer our research questions. We define bug **triaging time** T_{triage} and bug **fixing time** T_{fix} as follows:

$$\forall b \in B_{\text{assign}} : T_{\text{triage}}(b) = D_{\text{assign}}(b) - D_{\text{report}}(b)$$

$$\forall b \in B_{\text{fix}} : T_{\text{fix}}(b) = D_{\text{fix}}(b) - D_{\text{report}}(b)$$

Given a date range $d = [d_1, d_2[$, we define

$$B_{\text{resolve}}(d) = \{b \in B_{\text{resolve}} \mid D_{\text{resolve}}(b) \in d\}$$

and similarly for $B_{\text{report}}(d)$ and $B_{\text{fix}}(d)$. Using this notation, we define bug **resolution rate** *ResRate* as the proportion of reported bugs that has been `RESOLVED` in the considered date range, and bug **fixing rate** *FixRate* as the ratio of `FIXED` over *reported* bugs:

$$ResRate(d) = \frac{|B_{\text{resolve}}(d)|}{|B_{\text{report}}(d)|} \qquad FixRate(d) = \frac{|B_{\text{fix}}(d)|}{|B_{\text{resolve}}(d)|}$$

The following example illustrates these two metrics. Suppose 30 bugs are *reported* during date range d, of which 20 bugs are `RESOLVED` and 12 of those `RESOLVED` bugs are actually `FIXED`. Then $ResRate = \frac{20}{30} = 0.66$, and $FixRate = \frac{12}{20} = 0.6$.

4.2.4 Applying the Metrics to Specific Eclipse Releases

Since our empirical analysis aims to relate bug handling activity to specific time periods, such as the period separating two successive releases, the feature freeze period, and the development period preceding the feature freeze, we introduce functions and sets enabling us to work with such information.
Let $R = \{3.0, 3.1, 3.2, 3.3, 3.4, 3.5, 3.6, 3.7, 3.8, 4.2, 4.3, 4.4, 4.5, 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, 4.13, 4,13, 14,5\}$ be the ordered set of considered Eclipse releases. We define the following functions:

- date: $R \rightarrow Date$ returns the date of a given release
- prev: $R/\{3.0\} \rightarrow R$ returns the previous release of a given release
- next: $R/\{4.15\} \rightarrow R$ returns the next release of a given release
- freeze: $R \rightarrow Date$ returns the start of the feature freeze period for a given release

Given a release $r \in R$, we can focus the analysis on **only those bugs targeting release** r, based on the value of their *Version* field in the bug report. To do so, we introduce different release-dependent data ranges:

- $d^{<\text{freeze}}(r) = [\text{date}(\text{prev}(r)), \text{date}(\text{freeze}(r))[$ is the development period of release r
- $d^{>\text{freeze}}(r) = [\text{date}(\text{freeze}(r)), \text{date}(r)[$ is the feature freeze period of release r
- $d^{\text{before}}(r) = [\text{date}(\text{prev}(r)), \text{date}(r)[\quad = \quad d^{<\text{freeze}}(r) \cup d^{>\text{freeze}}(r)$
- $d^{\text{after}}(r) = [\text{date}(r), \text{date}(\text{next}(r))[$

Based on these data ranges, we restrict the set of reported bugs B_{report} (see Section 4.2.3) as follows:

- $B^{\text{before}}_{\text{report}}(r) = \{b \in B^r_{\text{report}} \mid D_{\text{report}}(b) \in d^{\text{before}}(r)\}$
- $B^{\text{after}}_{\text{report}}(r) = \{b \in B^r_{\text{report}} \mid D_{\text{report}}(b) \in d^{\text{after}}(r)\}$
- $B^{<\text{freeze}}_{\text{report}}(r) = \{b \in B^r_{\text{report}} \mid D_{\text{report}}(b) \in d^{<\text{freeze}}(r)\}$
- $B^{>\text{freeze}}_{\text{report}}(r) = \{b \in B^r_{\text{report}} \mid D_{\text{report}}(b) \in d^{>\text{freeze}}(r)\}$

In a similar way, we restrict the sets B_{assign}, B_{resolve} and B_{fix}.
With these notations, we can compute the *resolution rate* before and after each release, as well as during the development period or the freeze period of any given release, **restricted to only those bugs targeting release** r, by using

$$ResRate(d^{\text{before}}(r)) \qquad ResRate(d^{\text{after}}(r))$$

$$ResRate(d^{<\text{freeze}}(r)) \qquad ResRate(d^{>\text{freeze}}(r))$$

In a similar way, we restrict the *fixing rate* to a specific period related to a given release r.
To focus on bugs belonging to specific severity groups, we use the auxiliary function $severity : B_{report} \rightarrow \{normal, high\}$ to return the severity group (as explained in Section 4.2.2) of a given bug. Given a set of bugs B and a severity group s, we define $B \mid_s = \{b \in B \mid severity(b) = s\}$ as the subset of all bugs in B belonging to this severity group. Using this definition, we can for example define $B^{\text{before}}_{\text{report}} \mid_{high} (r)$ and similar for all other possible variations.

4.3 Statistical Methods

Our quantitative analysis in this chapter will rely on a range of statistical tools. Most of our analysis aims to compare two populations by testing if their distributions are different. We test all statistical hypotheses for different significance levels α. We reject a null hypothesis if $p < \alpha$, and denote this with * if $\alpha = 0.05$, ** if $\alpha = 0.01$, and *** if $\alpha = 0.001$.
Since software engineering data often do not meet the normality assumption [77], and this is the case for Eclipse bug data in particular [95], we select appropriate non-parametric tests that do not require this assumption [54]. Normality is tested for both populations that are compared using the Kolmogorov–Smirnov test. In case both populations are normally distributed, we compare their distributions and mean values with a *two-sided t-test* if the populations are related; otherwise, we use the *Welch t-test* if the samples are independent or of unequal size. For non-normal distributions, we use the *Wilcoxon rank sum test* if the samples are related; otherwise, we use the *Mann–Whitney U test*.
We compute the *effect size* using Cliff's delta d [25, 52] and interpret the results using [52]. In all cases where the null hypothesis is rejected; the sign of d allows us to determine which of both distributions is higher than the other one.
The analysis of *RQ*1.2, reported in Section 4.8, uses the technique of *survival analysis* (a.k.a. event history analysis or duration modeling) to model the expected time duration until the occurrence of a specific *event of interest* with the aim to estimate the survival rate of a given population [1]. Survival analysis is frequently used in medical sciences (e.g., to study the effect of a particular treatment on patients suffering from a disease), economics and sociology. It has also been applied in software evolution research [31, 113]. Survival analysis models take into account the fact that some observed subjects may be "censored", either because they leave the study during the observation period, or because the event of interest was not observed for them during the observation period. A common non-parametric statistic used to estimate survival functions is the Kaplan-Meier estimator [58]. To compare survival curves, we use the *log-rank test* [17] to test the null hypothesis that there is no difference between the populations in the probability of an event at any time point.
The analysis of *RQ*2.2 reported in Section 4.9, uses boxen plots [53] to show the distributions of dataset. These plots visualise different quartile values and convey precise estimates of head and tail behavior.

4.4 Feedback from Eclipse Maintainers

In order to verify if the empirical results we obtained correspond to the perception of the Eclipse community, we complemented the quantitative analysis with a small qualitative analysis by consulting Eclipse Core maintainers that actively experienced the transition from the yearly to the quarterly release cycle. We identified maintainers that were active in bug fixing activities, both *before* and *after* the change in release policy by manually analysing their presence and historical activity in the extracted bug tracker data.
Four out of five of the respondents reported they have at least been contributing to Eclipse Core for five years, while the remaining one has been contributing for two years. Using an online form (see Appendix A), we solicited their experience on the transition to a rapid release cycle and how they perceived this has impacted the bug handling process. The first three questions (#3–#5) collected demographic information. Questions #6–10 focused on the impact of switching to a rapid release

cycle, by means of open questions about the Eclipse preparation and difficulties they faced during the transition. Questions #11–17 inquired about the quantitative results we obtained for *RQ*1.1 to *RQ*2.2, in order to check whether our results confirm with their perception about the transition to a rapid release cycle. Moreover, we asked questions to (1) check if there were any other important changes in the Eclipse release process and if changes in tooling may have affected the effectiveness of bug handling and related activities; (2) ask their opinion if the community needs more contributors to be involved in bug handling; and (3) check if and how the transition to quarterly releases impacted the pressure on the developers. Finally, questions #17–20 informed about the perceived benefit of rapid releases in general, the respondent's preference of the type of release cycle (fixed or variable), and any advice for future adopters of rapid releases.

4.5 Bug Handling Process Discovery

Process mining is a research discipline that sits between machine learning and data mining on the one hand, and process modeling and analysis on the other hand. The idea of process mining is to discover, monitor and enhance real processes by extracting knowledge from event logs available in today's Information Technology systems [129]. Event logs are the starting point of all process mining techniques that use them to discover, verify or extend models for the process. In some cases, these models are used to gain insight, document, or analyze the process. Each event in such a log refers to an activity (i.e., a well-defined step in a process) and is related to a particular case (i.e., a process instance). The events belonging to a case are ordered and describe one "run" of the process. Also, the process mining techniques use supplementary information such as the "resource" (e.g., person) executing or initiating the activity and the timestamp of the event. For process discovery, there are many process mining tools such as ProM[6] (open source) and Disco[7](commercial) used to obtain process model. We exploit Disco to mine the process map and other statistical information (e.g., absolute frequency of activities) for the Eclipse bug handling process. Disco miner is based on the proven framework of the Fuzzy Miner, process discovery algorithms, with an entirely new set of process metrics(e.g., process case duration) and modeling strategies [8]. It is used to discover the runtime process from the actual event log generated during the progression of a bug.

We extract Status, Resolution, Assignee, Version and Timestamp from the bug history to derive the process map. Fig. 4.3 displays the snapshot of bug history in Bugzilla of Eclipse's bug (ID=40495). The bug report history serves as the process event log generated by the bug tracking system. In our analysis in Section 4.6, Bug ID is selected as a case to associate all activities of the same bug ID and hence, we can visualize the lifecycle of a bug. We record creation timestamp with the activity *NEW*. For open bugs, the status can be `NEW`, `UNCONFIRMED`, `ASSIGNED` and `REOPENED`, which is captured as an activity. For closed bugs, `VERIFIED`, `CLOSED` and `RESOLVED` statuses are captured as they are as an activity. The resolution activity often refers to `FIXED`, `INVALID`, `WONTFIX`, `DUPLICATE`, `WORKSFORME` and `INCOMPLETE` resolutions. The timestamp corresponding to activity is obtained from the *When* field of the bug history as can be seen on the right of Fig. 4.3 to order the activities in the sequence of their actual execution (while generating the process map via Disco). The one who

[6] https://www.promtools.org/

[7] https://www.fluxicon.com/

[8] https://fluxicon.com/disco/files/Disco-Tour.pdf

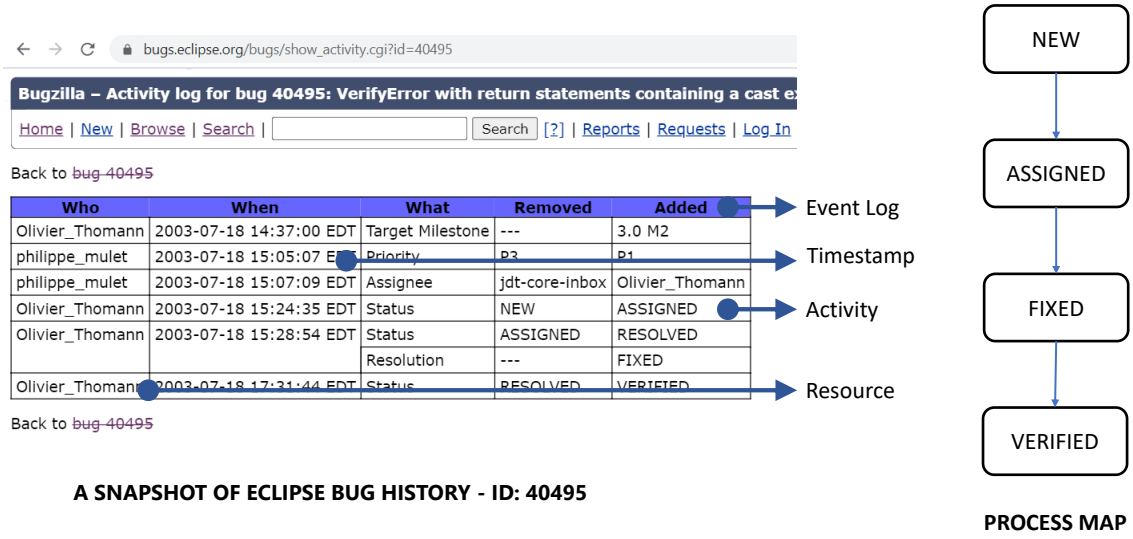


FIGURE 4.3: Snapshot of the bug history (event log) of bug 40495 of ECLIPSE in Bugzilla and corresponding process map.

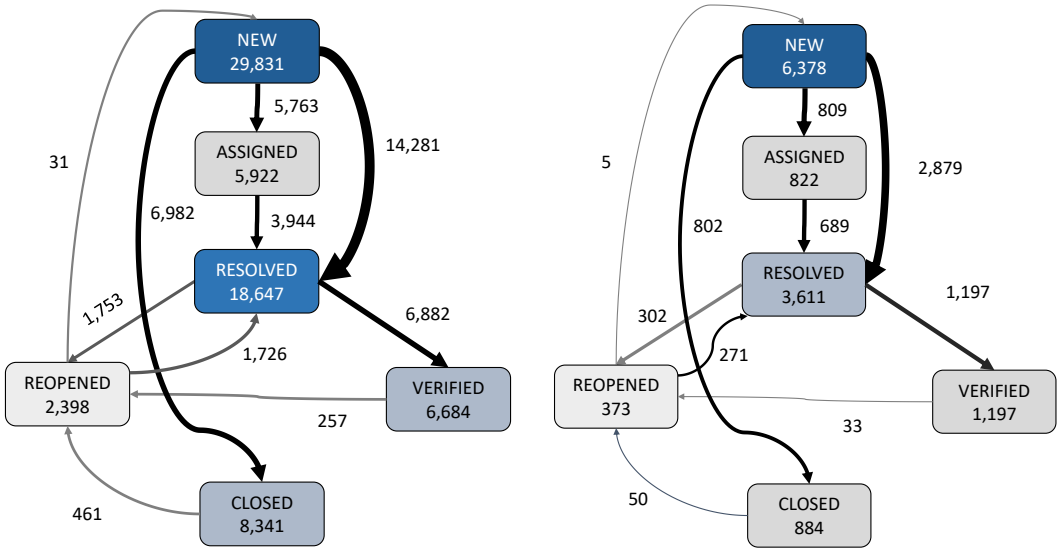
did the activity is treated as a resource and is retrieved from the who field of bug history.
One of the major challenges in software process mining is to produce a log conforming to the input format of the process mining tool [50]. Therefore, data preprocessing is required. Preprocessing steps performed are as follows:

- Activities with timestamp, bug ID and associated resource make the event log for process map generation.
- *NEW* event is not stored in bug history, so it is added to the event log with creation timestamp.
- Initial state of the bug is not captured in history if no activity leading to change in status and resolution is performed. For such cases, only *NEW* (with creation timestamp) is stored in the event log and not the initial state. Otherwise, the initial state is also stored with the same timestamp as Reported. For instance, the process map from the history of a bug is shown on the right of Fig. 4.3.

4.6 Applying Process Mining on Bug Handling Process

As mentioned in Section 4.2.2, Eclipse bug handling activity follows the Bugzilla life cycle shown in Fig. 2.4. To analyse the Eclipse bug cycle process, we apply Disco to the event logs (containing the bug report history) retrieved from Bugzilla.
We do this separately for the bug reports corresponding to the Eclipse annual releases and the Eclipse quarterly releases, respectively. An event log containing 29,831 cases (i.e., bugs) is given as an input to Disco to obtain the bug handling process for annual releases. Fig. 4.4a shows the obtained process map based on the bug reports of the annual releases. An event log containing 6,378 cases (i.e., bugs) is given as an input

to Disco to obtain the bug handling process in the quarterly releases. Fig. 4.4b shows the obtained process map based on the bug reports for quarterly releases.



(A) Annual releases (B) Quarterly releases

FIGURE 4.4: Process maps computed based on the event logs of the extracted Bugzilla bug reports for the Eclipse annual and quarterly releases.

The process 6 nodes, each corresponding to an activity (i.e., step in the bug handling process) for both annual and quarterly releases, and a directed edge that represents the transition between two activities (nodes). The number associated with each node corresponds to the absolute frequency of occurrence of all activities, and the number associated with each edge indicates the absolute frequency of occurrence of each transition. The shade of each node reflects its frequency (i.e., the darker, the more frequent). The shade and thickness of each edge reflect its frequency (i.e., the darker and thicker, the more frequent). For example, the transition $NEW \rightarrow RESOLVED$ is more frequent than transition $NEW \rightarrow ASSIGNED$, so it is thicker.
Contribution guidelines embody a software project's contribution process, however, most active projects that use CI do not follow their own guidelines (if they have any) [38]. In order to determine how faithfully Eclipse bug reports are actually following the recommended Bugzilla life cycle shown in Fig. 2.4, we used an algorithm proposed by Gupta et al. [50] to compute the *degree of conformance* (a.k.a. *fitness*). Conformance checking aims to detect inconsistencies between a design-time process model (here, the Bugzilla life cycle) and the as-is process model extracted from the runtime event logs (here, the bug life cycles extracted for the Eclipse annual and quarterly releases, respectively). The higher the fitness, the better the design-time process model describes the recorded run-time process. A fitness of 1 indicates that the design-time process model reproduces every trace in the event log, a value of 0 means that the design-time process model cannot repeat any of the run-time cases. We obtained a high fitness of 0.72 for annual Eclipse releases and an even higher fitness of 0.86 for quarterly Eclipse releases. This shows that the Eclipse bug handling process mostly conforms to the recommended Bugzilla process model.

As a second analysis, we aimed to determine if the Eclipse Core bug handling process has changed after the transition from annual to quarterly releases. To do so, we carried out a χ^2-test. The null hypothesis H_0 states that there is no statistically significant difference between the absolute frequency of bug statuses (node values in the process maps) for the annual and quarterly releases process. We could reject H_0 with high confidence ($p = 0$), signifying that the differences between bug status frequencies of annual and quarterly releases are statistically significant. Given that such a difference has been found between annual and quarterly releases, Section 4.8 and Section 4.9 will explore how other aspects might have change such as the bug handling (e.g., in terms of bug handling rate and time) between both types of releases.

4.7 Quantitative Analysis of the Evolution of Eclipse Bug Handling Activity

Thus, before starting to study the actual research questions presented in Section 4.1 we carry out a preliminary analysis of how Eclipse bug handling evolved over time since release 3.0. In order to assess the efficiency of the bug handling [78], for each considered release, we computed B^r_{report}, B^r_{resolve}, B^r_{fix} and B^r_{assign}, without any restriction on the date range. Fig. 4.5 shows these statistics and reveals that the number of reported bugs targeting a given release (blue line) is monotonically decreasing all along the 3.x range of annual releases. Starting from annual release 4.2, the number of bug reports appears to become stable. For the quarterly releases starting from 4.9, the numbers are still stable, but much lower than for the annual releases. This can be due to their shorter duration or less code churn.

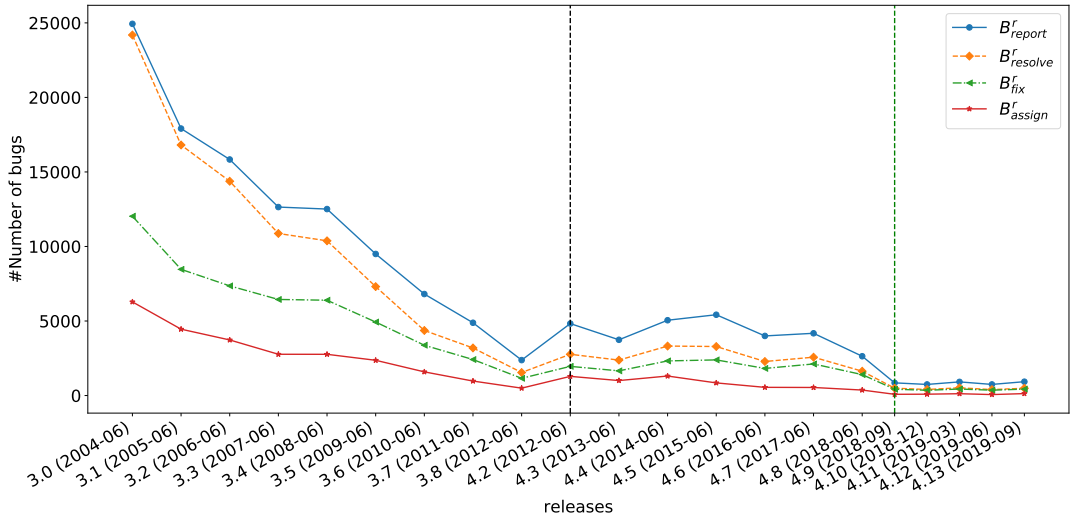


FIGURE 4.5: Number of bugs targeting a specific release. The black vertical dashed line indicates the switch from Eclipse 3.x to 4.x. The black vertical dashed line indicates the transition from an annual to a quarterly cycle since 4.9.[9]

Fig. 4.6 shows the evolution of *ResRate* and *FixRate* per considered release r. We observe a *decreasing* resolution rate, while the fixing rate is *increasing* across releases. For the 3.x release range, *ResRate* decreases from 0.97 to 0.67, while *FixRate* increases

[9]In all our figures, the black vertical line signifies the switch from Eclipse 3.x to 4.x and the green vertical dashed line signifies the transition from an annual to a quarterly cycle.

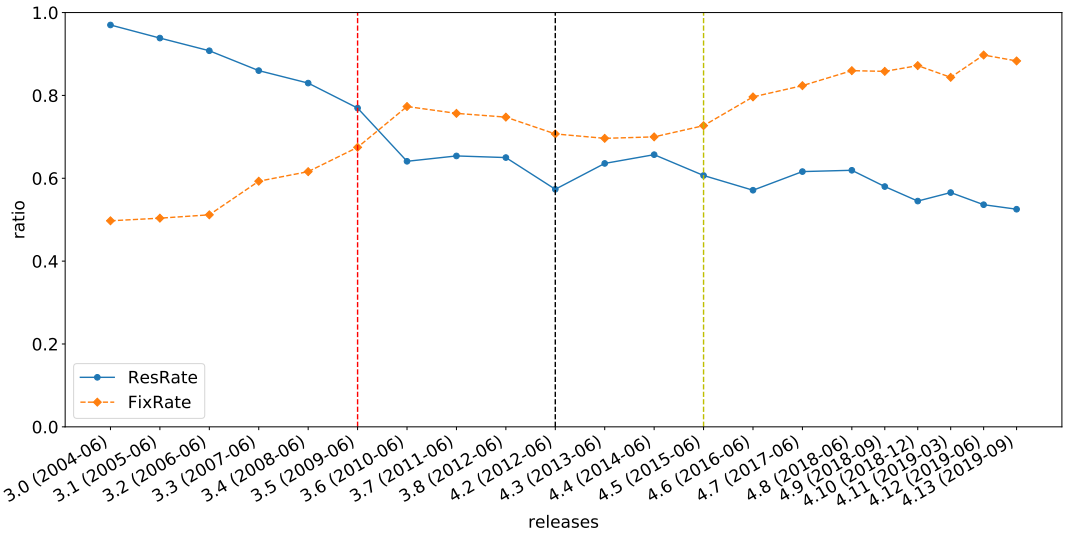


FIGURE 4.6: Resolution and fixing rates per targeted Eclipse release. The red vertical line indicates the last release where `REMIND/LATER` resolution was used. The yellow vertical line corresponds to the introduction of the AERI error reporting tool.

from 0.50 to 0.72. Starting from release 3.6, *FixRate* is consistently higher than *ResRate*, and the difference between both rates continues to become more pronounced over time.

The observed change in behaviour since release 3.6 can be explained by the practice of "resolving" a bug by giving it the `LATER` or `REMIND` status in the *Resolution* field, corresponding to the desire to postpone the bug resolution.[10] This practice was fairly common early on in the 3.x release range, but gradually declined and is no longer used since release 3.6 (red vertical line in Fig. 4.6). By analyzing the delays of a follow-up resolution of the `REMIND` and `LATER` bugs (3,633 bugs), we find that the majority of them (56%) lingered for more than 3 years before getting their final resolution. We study these events by applying survival analysis to model the expected time duration for a bug to be subsequently resolved without `REMIND/LATER` resolution. The event of interest is the time when a bug has been `RESOLVED`, and the duration is computed from the first time the bug was marked as `LATER` or `REMIND` until the first resolution that is different from `LATER` and `REMIND`. The Kaplan-Meier survival curves in Fig. 4.7 visualise the survival model. We observe that, for more than 50% of the bugs that were marked as `LATER` or `REMIND`, it takes more than 1,220 days to actually resolve them later on (see red dashed lines in Fig. 4.7). The follow-up resolution statuses are, in decreasing order of frequency: `WONTFIX` (1,257 bugs), `INVALID` (1,058 bugs), `FIXED` (495 bugs), `DUPLICATE` (253 bugs), `WORKSFORME` (242 bugs) and `NOT_ECLIPSE` (11 bugs). 317 of the `REMIND/LATER` bugs (i.e., 8.7%) are not resolved with another resolution until today. We checked if the severity group (*non-severe* or *severe*) of the bugs marked as `LATER/REMIND` influences the resolution time but we did not observe any such effect.

Coming back to Fig. 4.6, for the 4.x annual release range we observe a stability in the rates up until release 4.5, after which *ResRate* continues to decrease and *FixRate* continues to increase. This change coincides with the introduction of an Automated Error Reporting Client (called AERI)[11] since June 2015 (yellow vertical line). AERI

[10] See https://www.eclipsezone.com/eclipse/forums/t83053.html

[11] See https://www.eclipse.org/community/eclipse_newsletter/2016/july/article3.php

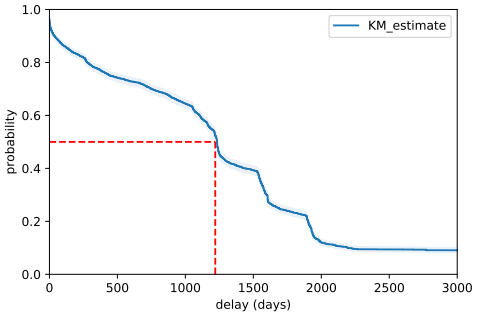


FIGURE 4.7: Kaplan-Meier survival curves modeling the time duration until a bug marked as `LATER` or `REMIND` gets resolved at some later time.

facilitates reporting errors as users do not need to create Bugzilla entries and *'it automatically uploads issues to a central server, providing valuable information as to where issue may exist in Eclipse'*.[12]
In turn, users can provide comments with their reports which are helpful when fixing bugs. According to Sewe's report on AERI [115], commented bug reports are more than twice as likely to be fixed compared to those without user comments because this helps the project developers reproduce the issue.
The bug handling metrics reported in Figures 4.5 and 4.6 for the 3.x series are quite different from the recent 4.x series, implying that the way in which bugs were handled during the 3.x series does not reflect the current practices of Eclipse Core maintainers. Because of this, Sections 4.8 and 4.9 only focus on the 4.x series during which the transition to rapid releases happened.

Summary: The resolution rate tends to decrease over releases, while the fixing rate is increasing over time. The decrease of resolution rate can be explained by the drop-off of resolving a bug by giving it `LATER` or `REMIND` status. The introduction of AERI was also considered beneficial by the Eclipse community as the fixing rate increases.

4.8 Impact of Rapid Releases on the Bug Handling Process of Eclipse

Our first research goal aims to study if Eclipse's transition to rapid releases resulted in less time for the community to handle bugs before the release, leading to potentially more post release bugs in need of resolution. In this section, we analyse if the bug handling activity is different before and after each release, and whether and how this changed after the transition to rapid releases. Our investigation is guided by the first two research questions (RQ1.1 and RQ1.2).
The quantitative results that will be presented throughout this section will be corroborated by the feedback we received from the five consulted Eclipse maintainers (cf. Section 4.4).

[12] https://www.infoq.com/news/2015/03/eclipse-mars-reporting/

4.8.1 RQ1.1 How does the bug handling rate evolve across releases?

In this research question, we study the difference in bug resolution and fixing rate before and after each release. We intuitively expect that contributors handle bugs more intensively in the period *before* than *after* the upcoming release date, as they strive to not deliver a buggy release.
First, we compute, for each release r in the 4.x series, the resolution rate before and after the release, i.e., $ResRate(d^{\text{before}}(r))$ and $ResRate(d^{\text{after}}(r))$. Fig. 4.8 suggests a slightly decreasing rate after the different releases while the rate is fluctuating before the release. A linear regression analysis confirms this trend for the resolution rate after each release ($R^2 = 0.74$). The regression analysis could not reveal any linear trend for the resolution rate before each release because of a too small R^2 value (0.0018).

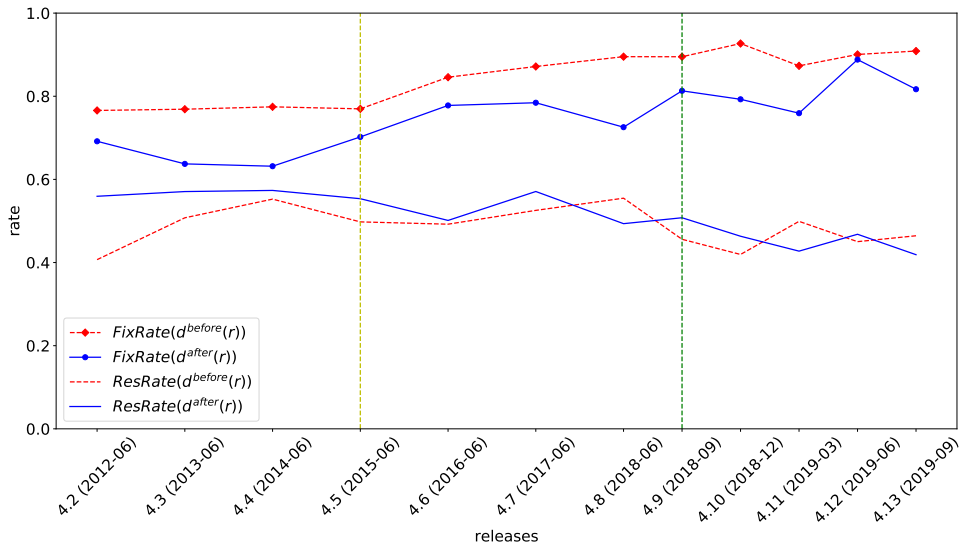


FIGURE 4.8: Evolution of *ResRate* and *FixRate* before and after each 4.x release. The yellow vertical line corresponds to the introduction of the AERI error reporting tool.

The resolution rates *before* and *after* each release fluctuate between 0.4 and 0.6. This signifies that around 1 out of 2 bugs do not get resolved. We used a Wilcoxon rank sum test to verify the null hypothesis $H0_1^r$ stating that there is no statistically significant difference between the resolution rates before and after the release. We could not reject $H0_1^r$ so we have no evidence that the resolution rates before and after each release are different.
Next, we computed the fixing rate before and after each release r, i.e., $FixRate(d^{\text{before}}(r))$ and $FixRate(d^{\text{after}}(r))$. Fig. 4.8 suggests that the fixing rate *before* each release is higher than *after* the release. Using the Wilcoxon rank sum we test the null hypothesis $H0_1^f$ stating that there is no statistically significant difference between fixing rates before and after a release. $H0_1^f$ was rejected ($p < 0.05$) with large effect size ($d = 0.54$). This shows that the bug fixing rate before the release date is higher than after that date. A linear regression analysis confirms an increasing linear trend for fixing rate before each release ($R^2 = 0.89$) as well as after each release ($R^2 = 0.74$).
Four out of five of the consulted Eclipse Core maintainers stated that they do not really differentiate between bugs before and after the release. The received responses were based on a 5-point Likert scale, i.e., participants rated factors using ranks from

1 (strongly disagree) to 5 (strongly agree). This complies with our observation that there is no difference between resolution rate before and after release. However, as we see in Fig. 4.8, a higher proportion of bugs is fixed before compared to after the release.

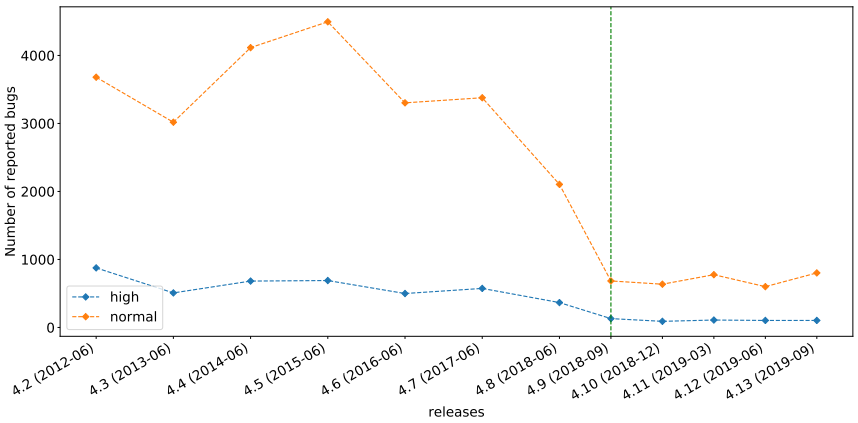


FIGURE 4.9: Evolution of the number of reported bugs per severity group.

We also investigated if maintainers differentiate between bugs according to their severity. Fig. 4.9 shows that the number of reported *non-severe* bugs decreases over time to become stable after the transition to a quarterly release cycle. We also computed $ResRate(d^{\text{before}}(r))$ and $ResRate(d^{\text{after}}(r))$ per severity group (*severe* and *non-severe*) and found a decreasing evolutionary trend for both, similar to what was observed in Fig. 4.8.[13] Using the Wilcoxon rank sum we test the null hypotheses $H0_1^r\,|_{non-severe}$ and $H0_1^r\,|_{svvere}$ stating that there is no statistically significant difference between resolution rates before and after a release for *non-severe* and *severe* bugs, respectively. We could not reject the null hypothesis for either group, however.
When computing the bug *fixing* rates $FixRate(d^{\text{before}}(r))$ and $FixRate(d^{\text{after}}(r))$ per severity group, we found an increasing evolutionary trend, similar to what was observed in Fig. 4.8. Using the Wilcoxon rank sum, we test the null hypotheses $H0_1^f\,|_{non-severe}$ and $H0_1^f\,|_{severe}$ stating that there is no statistically significant difference between the fixing rates before and after a release for *non-severe* and *severe* bugs respectively. Both $H0_1^f\,|_{non-severe}$ and $H0_1^f\,|_{severe}$ were rejected ($p < 0.05$ and $p < 0.01$ respectively) with a large effect size ($d = 0.53$ and $d = 0.7$ respectively). This confirms that, for both severity groups, the bug fixing rate before the release date is higher than after that date. This signifies that maintainers strive to fix as many bugs as possible of either severity group for the next release.
Finally, we study if any difference could be observed between the two bug severity groups in terms of resolution and fixing rate. We first assess whether a difference between *non-severe* and *severe* bugs can be observed *before* a release. We carry out a first Wilcoxon rank sum test with null hypothesis $H0_1^r\,|_{before}$ stating that, before a release, there is no difference in *resolution rates* between *non-severe* and *severe* bugs. Similarly, for *fixing rate* we carry out a test with null hypothesis $H0_1^f\,|_{before}$. We could not reject $H0_1^r\,|_{before}$ and $H0_1^f\,|_{before}$, hence there is no statistical evidence of a difference between bug severity groups.
We performed the same hypothesis test to check for a difference in resolution rate and in fixing rate between bug severity groups *after* a release. The respective null

[13] The results per severity group can be found in our replication package [62].

hypothesis $H0_1^r \mid_{after}$ for *resolution rate* can be rejected for $p < 0.05$ with large effect size ($d = 0.48$), and for *fixing rate* the null hypothesis $H0_1^f \mid_{after}$ can be rejected for $p < 0.01$ with large effect size ($d = 0.6$). This implies that after the release, the bug resolving and bug fixing rates for *non-severe* bugs are higher than for *severe* bugs. Overall, our results show that there is a tendency to resolve more *non-severe* than *severe* bugs after the release. Regarding our findings, we asked the five Eclipse maintainers if they differentiate between bugs according to their severity when handling bugs, but this was not the case. One consulted maintainer stated that long release cycles lead to a higher amount of bugs that are not important or relevant to the bug reporter anymore because users found a workaround, or because the tooling has changed since. With shorter cycles, this is no longer the case, so it becomes less relevant for maintainers to distinguish between important and less important bugs.

Summary: The resolution rate tends to decrease over releases, but there is no significant difference before and after each release. The fixing rate is higher before than after a release. There is no significant change in bug handling rate behavior after the switch to quarterly releases. There is a tendency to resolve more non-severe than severe bugs after the release. However, Eclipse maintainers mentioned that they do not tend to differentiate between bugs based on their severity.

4.8.2 RQ1.2 How does the bug handling time differ before and after each release?

We expect that bug handling activity is more intense *before* a release than *after* it, as maintainers strive to deliver project releases with as few bugs as possible. Hence, we expect to see faster bug triaging and fixing times *before* the release. In this question, we investigate the differences between these two periods in terms of days elapsed to triage and fix bugs.

Triaging time analysis For each release r in the 4.x series of Eclipse Core, we compute bug *triaging time* $T_{\text{triage}}(b)$ before a release for the population of bugs $B^{\text{before}}_{\text{report}}(r) \cap B^{\text{before}}_{\text{assign}}(r)$ that were reported *and* assigned during that period; and similarly after a release for the population $B^{\text{after}}_{\text{report}}(r) \cap B^{\text{after}}_{\text{assign}}(r)$. We use survival analysis on these populations to model the expected time duration $T_{\text{triage}}(b)$ until *bug b gets assigned for the first time*. Fig. 4.10 and Fig. 4.11 show the Kaplan-Meier survival curves for annual and quarterly releases, respectively, together with their confidence intervals, before and after the release date. We observe that for some releases, triaging time is different before than after that release. For instance, triaging time before the release is higher than after for releases 4.2 and 4.3, while the opposite is true for release 4.6. However, for most of the releases, the survival curves are partially overlapping. We also observe that the difference between the survival curves tends to decrease over successive releases. We use a log-rank test to verify the null hypothesis $H0_2^t$ stating that there is no difference between the survival distributions of the triaging time before and after a release. Table 4.1 reports for which releases $H0_2^t$ can be rejected. We observe that $H0_2^t$ is rejected for all annual releases except 4.6 and 4.7, while it is not rejected for any of the quarterly releases. Hence, for annual releases triaging times of the bugs before a release were different than after the release, while this is no longer the case for the quarterly releases. The transition from an annual to a quarterly release policy appears to have been beneficial, since such a difference

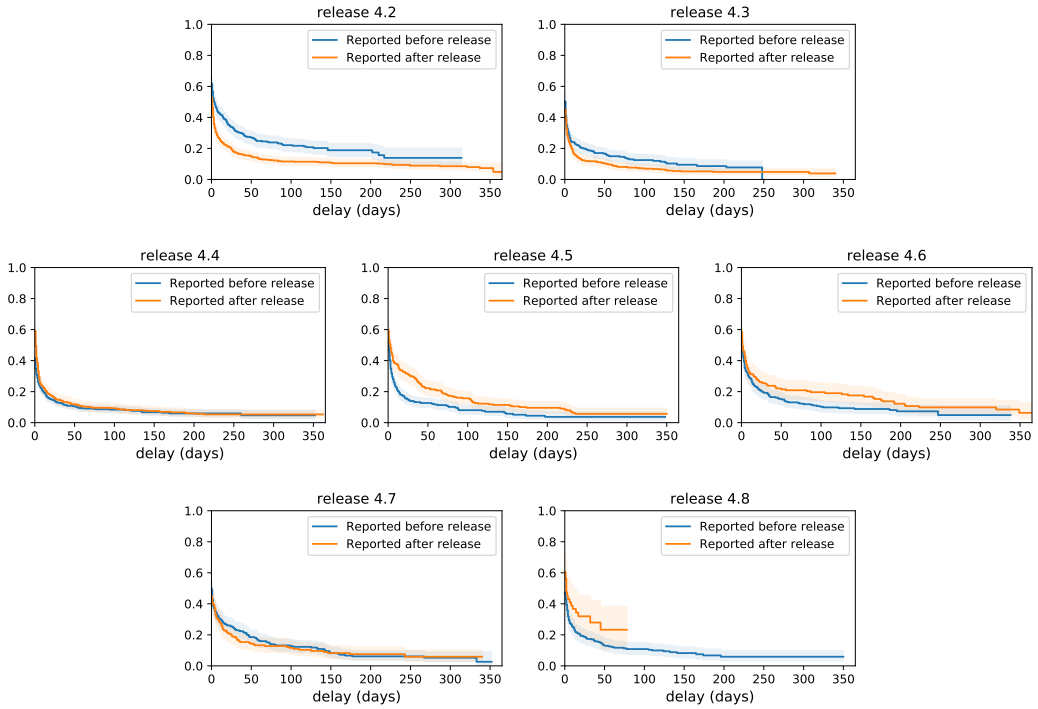


FIGURE 4.10: Kaplan-Meier survival curves with 95% confidence interval (indicated by the shaded areas) for *triaging time* before and after each *annual* release.

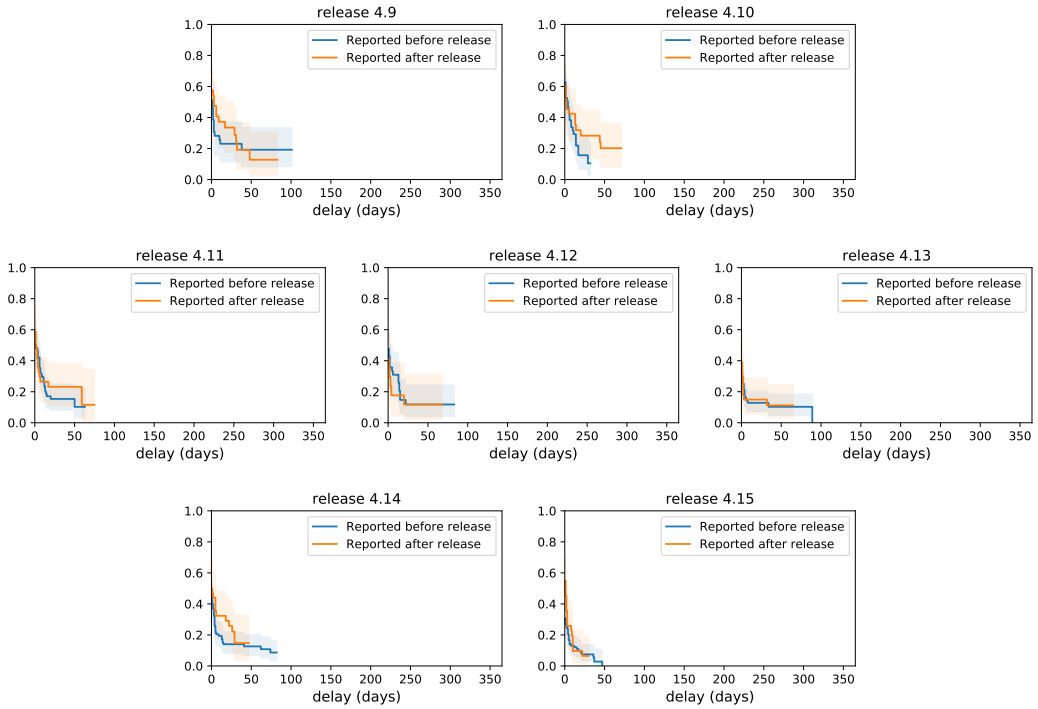


FIGURE 4.11: Kaplan-Meier survival curves with 95% confidence interval (indicated by the shaded areas) for *triaging time* before and after each *quarterly* release.

in triaging time is no longer observed. We also studied the impact of bug severity on

release	triaging time ($H0_2^t$)
Eclipse 4.x annual releases	
4.2	R^{***}
4.3	R^{*}
4.4	R^{*}
4.5	R^{***}
4.6	–
4.7	–
4.8	R^{*}
Eclipse 4.x quarterly releases	
4.9	–
4.10	–
4.11	–
4.12	–
4.13	–
4.14	–
4.15	–

TABLE 4.1: Log-rank test for difference between the survival distributions of **triaging time** *before* and *after* each release. – indicates that the null hypothesis could not be rejected. Whenever it could be rejected (R), the number of stars denotes the significance level α (*=0.05; ** =0.01; *** =0.001).

triaging time but the results were the same as what has been reported in Table 4.1. Hence, bug severity does not seem to have a measurable effect on bug triaging time. To study the impact of quarterly releases on the bug handling process, we analyzed the bug triaging time before and after switching to the quarterly releases. We used survival analysis to model the expected time duration $T_{\text{assign}}(b)$ until bug b is triaged. Fig. 4.12 presents the survival curves for bug triaging time for all the annual releases (blue line) and for the quarterly releases (orange line). The figure shows that the bugs are triaged slightly faster after the switch to quarterly releases. For example, 90% of bugs of the quarterly releases are triaged within 50 days while it took more than 100 days to triage 90% of the annual release bugs (see red dashed lines in Fig. 4.12). With a statistical log-rank test, we verify that there is a difference between the bug triaging time before and after the quarterly releases ($p < 0.01$).

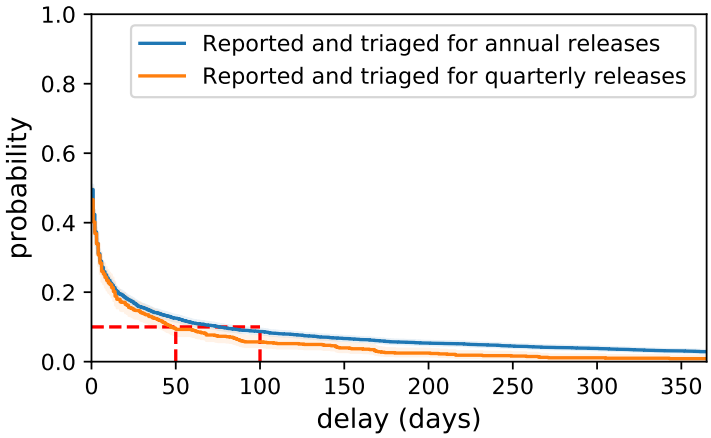


FIGURE 4.12: Kaplan-Meier survival curves for bug *triaging time* for annual and quarterly releases.

Summary: For annual releases, bugs tend to get triaged faster before than after the release. The transition from an annual to a quarterly release policy appears to have been beneficial, since such a difference in triaging time is no longer observed. The bug severity does not seem to have a measurable effect on bug triaging time before and after the release. Moreover, bugs are triaged faster after the switch to quarterly releases.

Fixing time analysis Similar to the triaging time analysis, we compute bug *fixing time* $T_{\text{fix}}(b)$ before a release for the population of bugs $B^{\text{before}}_{\text{report}}(r) \cap B^{\text{before}}_{\text{fix}}(r)$ that were reported *and* fixed during that period; and similarly after a release for the population $B^{\text{after}}_{\text{report}}(r) \cap B^{\text{after}}_{\text{fix}}(r)$. We use survival analysis on these populations to model the expected time duration $T_{\text{fix}}(b)$ corresponding to *the last recorded time that the bug gets fixed*.

Fig. 4.13 and Fig. 4.13 show the Kaplan-Meier survival curves for the bug fixing time for annual and quarterly releases, respectively. The graphs reveal at most a small difference in the time to fix a bug before and after a release. It seems to take slightly less time to fix a bug before compared to after a release. With a log-rank test we verify hypothesis $H0^f_2$ that there is no difference between the fixing time survival distributions before and after a release. We can reject $H0^f_2$ for releases 4.4 and 4.5 only. There is no difference between fixing time before and after the release except for releases 4.4 and 4.5.

We also studied the impact of bug severity on fixing time but the results were essentially the same as what has already been reported in Table 4.2. Hence, bug severity does not seem to have a measurable effect on bug fixing time.

To study the impact of the release cycle on the bug handling process, we analyzed the bug fixing time before and after switching to quarterly releases. We used survival analysis to model the expected time duration $T_{\text{fix}}(b)$ until a bug b is fixed. Fig. 4.15 compares the survival curves for bug fixing time for all the annual releases (blue line) and for the quarterly releases (orange line). The figure shows that bugs are fixed faster for quarterly releases. With a log-rank test, we verify that there is a difference between the bug fixing time before and after the quarterly releases ($p < 0.001$). When consulting Eclipse maintainers about this phenomenon, three out of five believed

release	fixing time ($H0_2^f$)
Eclipse 4.x annual releases	
4.2	–
4.3	–
4.4	R^{***}
4.5	R^{***}
4.6	–
4.7	–
4.8	–
Eclipse 4.x quarterly releases	
4.9	–
4.10	–
4.11	–
4.12	–
4.13	–
4.14	–
4.15	–

TABLE 4.2: Log-rank test for difference between the survival distributions of **fixing time** *before* and *after* each release. – indicates that the null hypothesis could not be rejected. Whenever it could be rejected (R), the number of stars denotes the significance level α (*=0.05; ** =0.01; *** =0.001).

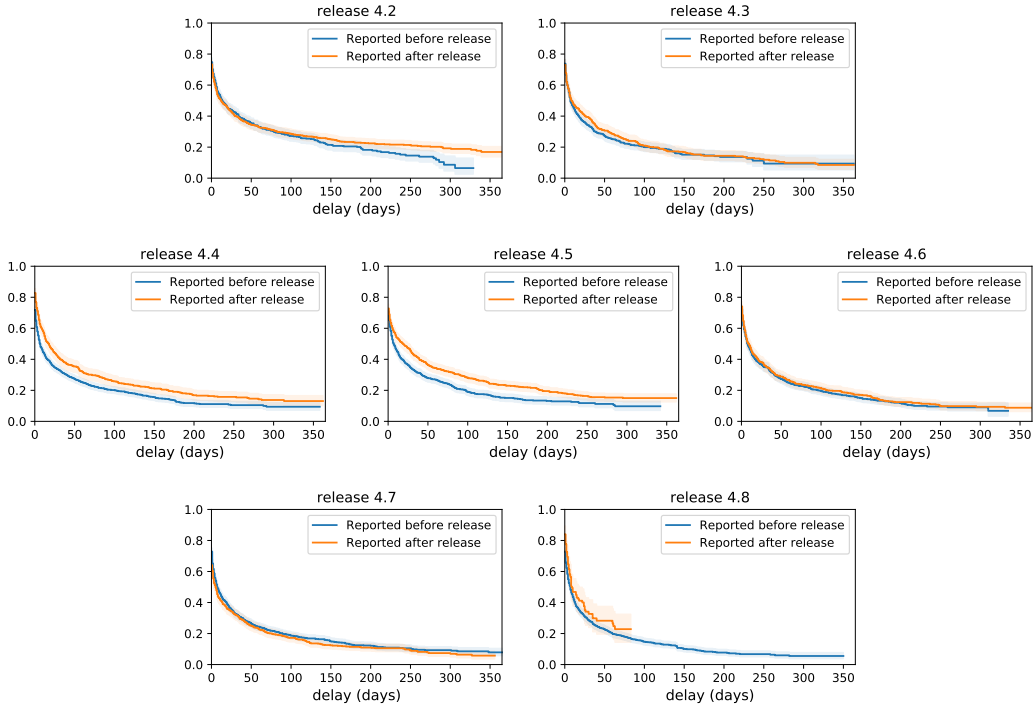


FIGURE 4.13: Kaplan-Meier survival curves with 95% confidence interval (indicated by the shaded areas) for *fixing time* before and after each *annual* release.

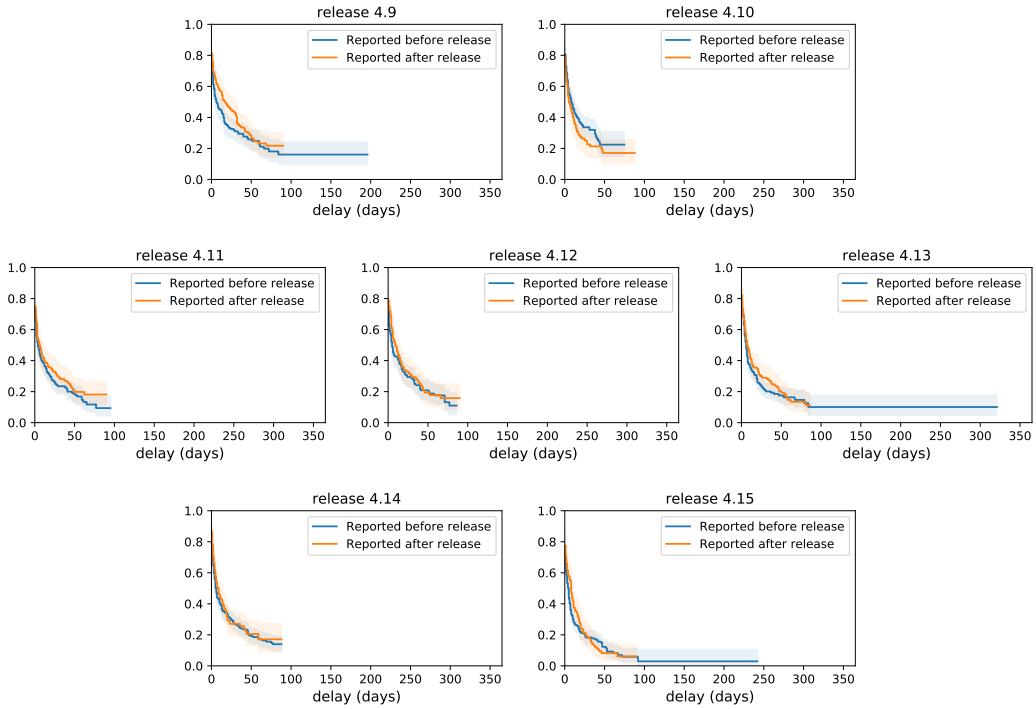


FIGURE 4.14: Kaplan-Meier survival curves with 95% confidence interval (indicated by the shaded areas) for *fixing time* before and after each *quarterly* release.

that bugs are fixed faster in the quarterly releases, which is in conformance with our quantitative analysis.

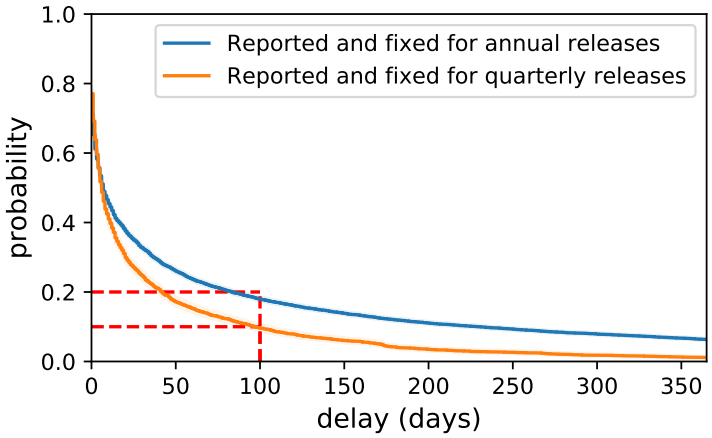


FIGURE 4.15: Kaplan-Meier survival curves for bug *fixing time* for annual and quarterly releases.

Summary: There is no difference between the bug fixing time before and after the release. The bug severity does not seem to have a measurable effect on bug fixing time before and after the release. Bugs are fixed faster after the transition to the quarterly releases, and this result was confirmed by the consulted Eclipse maintainers.

4.9 Impact of Feature Freezes on Bug Handling in Eclipse

Our second research goal aims to study to which extent bug handling activity is affected by the presence of feature freezes, and whether the transition to rapid releases has lead to an observable difference as their shorter duration can potentially affect the bug handling activity. Our investigation is guided by two research questions (RQ2.1 and RQ2.2).
The quantitative results that will be presented throughout this section will be corroborated by the feedback we received from five consulted Eclipse maintainers (cf. Section 4.4).

4.9.1 RQ2.1 How does the feature freeze period impact bug handling rate?

This question studies the bug handling rate before and during the feature freeze period of each considered release. As explained in Section 4.2.1, during the *feature freeze period*, Eclipse Core project maintainers stop introducing new features, and concentrate only on fixing bugs to stabilize the upcoming release. As visualized in Fig. 4.1, for the annual releases of the 4.x series this period varied between 1 to 3 months.[14] For the quarterly releases, the feature freeze period starts 3 weeks before the release date.[15]
As maintainers focus on bug fixing in the feature freeze period, we study if there is a difference in the bug resolution and fixing rate in the development period and during the feature freeze. We group our results in two date ranges based on when the resolution or fixing took place: (1) during the development period $d^{<\text{freeze}}(r)$ of the *next* release r; and (2) during the feature freeze period $d^{>\text{freeze}}(r)$ of the *next* release r. For each release in the 4.x series and each period we compute the resolution rate as well as the fixing rate.

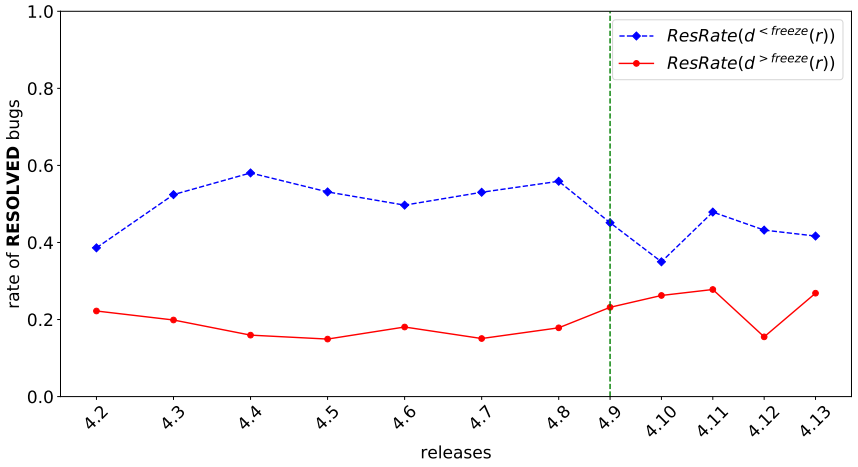


FIGURE 4.16: Evolution of *ResRate* during the development and feature freeze period for each 4.x release.

Fig. 4.16 shows that the *resolution rate* is fluctuating during the development period and the feature freeze period. A regression analysis could not reveal any linear trend for $ResRate(d^{<\text{freeze}}(r))$ ($R^2 = 0.07$) nor for $ResRate(d^{>\text{freeze}}(r))$ ($R^2 = 0.19$) because

[14] https://www.eclipse.org/eclipse/development
[15] https://wiki.eclipse.org/SimRel/Simultaneous_Release_Cycle_FAQ

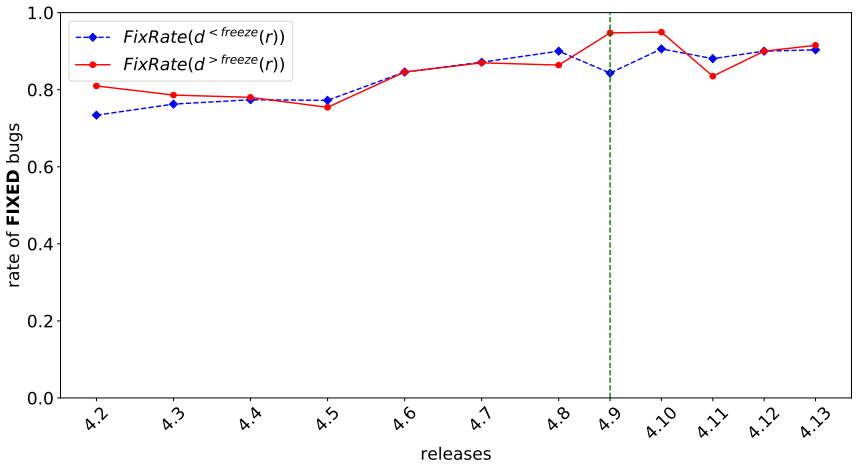


FIGURE 4.17: Evolution of *FixRate* during the development and feature freeze period for each 4.x release.

of a very small R^2 value. We used a Wilcoxon rank sum test to verify the null hypothesis $H0_3^r\ |_{\text{freeze}}$ stating that there is no statistical difference between the resolution rates before and during the freeze period. We could reject $H0_3^r\ |_{\text{freeze}}$ ($p < 0.001$) with the largest possible effect size ($d = 1$), since for any given release the resolution rate is higher during the development period than during the freeze period.
Fig. 4.17 shows that the *fixing rate* is increasing over releases during the development period and feature freeze period. A linear regression analysis confirms an increasing linear trend for $FixRate(d^{<\text{freeze}}(r))$ ($R^2 = 0.9$) as well as for $FixRate(d^{>\text{freeze}}(r))$ ($R^2 = 0.58$). Both curves are overlapping over all releases; we do not observe any difference between $FixRate(d^{<\text{freeze}}(r))$ and $FixRate(d^{>\text{freeze}}(r))$. Using the Wilcoxon rank sum we test the null hypothesis $H0_3^f\ |_{\text{freeze}}$ stating that there is no statistically significant difference between fixing rates before and during the feature freeze period. $H0_3^f\ |_{\text{freeze}}$ could not be rejected ($p = 0.78$), indicating that no difference can be reported between the bug fixing rate in the development and feature freeze period.

The feature freeze period aims to focus on fixing bugs and improving the overall stability of the upcoming release. We, therefore, hypothesize that more effort is spent on fixing bugs during this period than during the development period. We quantify the *bug fixing effort* as the weekly average number of fixed bugs in the considered period. Taking a weekly average allows us to account for the shorter total duration of the feature freeze and development periods for quarterly than for annual releases. Fig. 4.18 clearly suggests that bug fixing effort is higher during the feature freeze period than during the development period except for releases 4.4 and 4.12. In release 4.4, Java 8 was supported and we find that a large number of bugs was reported and fixed for *JDT*,[16] that worked on Java 8 tooling in Eclipse.[17] This can explain the high effort during the development period for release 4.4. We attribute the low observed effort for the feature freeze period of release 4.12 to a low number of bugs that were reported for this release. A Wilcoxon rank sum test confirms our observation. We could reject the null hypothesis $H0_3^{\text{effort}}$, stating that there is no statistically significant difference between the bug fixing effort in the development

[16] https://projects.eclipse.org/projects/eclipse/reviews/4.4.0-release-review
[17] https://eclipsesource.com/blogs/2014/03/25/eclipse-support-for-java-8/

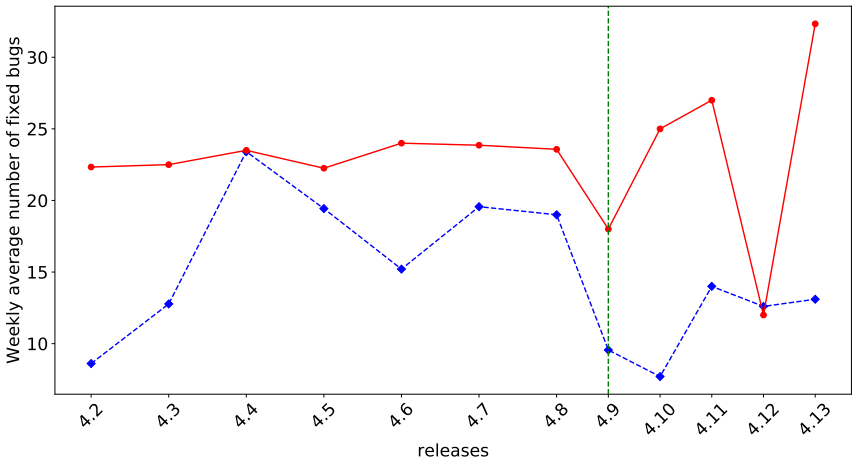


FIGURE 4.18: Comparison of *bug fixing effort* (weekly average number of fixed bugs) between development period (dashed blue lines) and feature freeze period (straight red lines) for each release.

and the feature freeze period, with high significance ($p < 0.001$) and large effect size (d = -0.8).

In addition to the above, the difference in bug fixing effort between development and feature freeze periods appears to become more pronounced after the transition from an annual to a quarterly release cycle. This suggests that the rapid release cycle has shifted more of the bug handling effort to the feature freeze period. Unfortunately, we do not have sufficient data points yet to confirm this hypothesis.

Another aspect of the feature freeze period is that more intense testing is being carried out [138]. These tests lead to more bugs being reported during the feature freeze period. To verify this, we investigate if maintainers during the feature freeze period focus more on fixing bugs that were reported during this period as opposed to during the development period. Considering the bugs targeting the next release r, and considering only those bugs fixed during the feature freeze period $d^{>\text{freeze}}(r)$, we group them into two categories based on when they have been reported: during the development period $d^{<\text{freeze}}(r)$ or during the feature freeze period $d^{>\text{freeze}}(r)$. Fig. 4.19 shows that more bugs reported in the feature freeze period (red line) are being fixed compared to those having been reported during the development period (blue line). We verified this using a Wilcoxon rank sum test with null hypothesis $H0_3^f\,|_{freeze}$ stating that, for the number of bugs fixed during the feature freeze period, there is no statistically significant difference between the ones created in the feature freeze period and those created in the development period. We could reject $H0_3^f\,|_{freeze}$ with high significance ($p < 0.001$) and the largest possible effect size ($d = 1$).

Summary: There is no observable difference in fixing rate between the development period and feature freeze period of each release. However, the feature freeze period focuses more on bugs being reported in that same period than on bugs reported earlier. As expected, more effort is spent on fixing bugs during the feature freeze period than during the development period, and this difference in effort appears to have increased for quarterly releases.

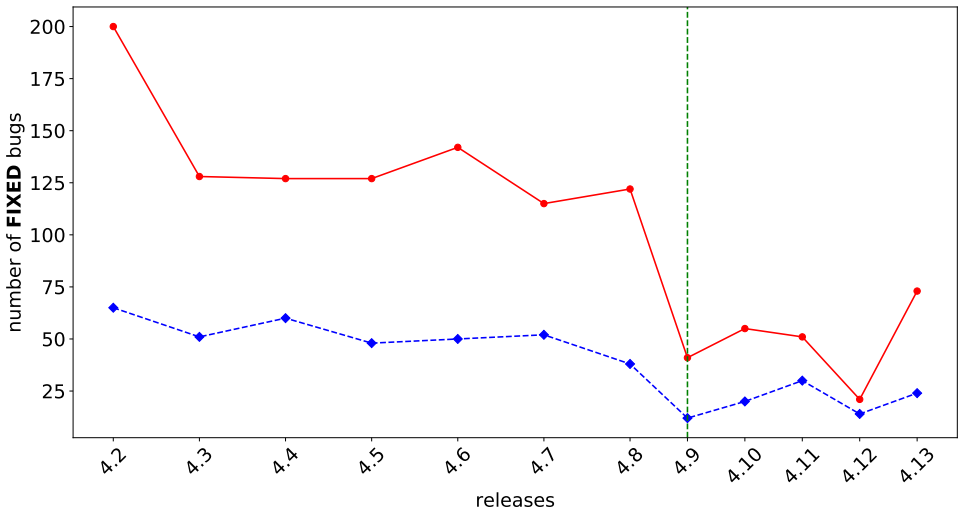


FIGURE 4.19: Evolution of number of *fixed* bugs during the feature freeze period for each 4.x release. The red line corresponds to bugs reported in the feature freeze period and the blue line to bugs reported in the development period.

4.9.2 RQ2.2 How does the feature freeze period impact bug handling time?

During the feature freeze period, maintainers focus on testing and bug fixing before delivering the next release. This research question studies if, during the development period $d^{<\text{freeze}}(r)$ and the feature freeze period $d^{>\text{freeze}}(r)$ of the next release r, there is a difference in resolution time and fixing time for bugs corresponding to the current release $prev(r)$ and bugs corresponding to the next release r. On the one hand, maintainers are still involved in handling bugs of the *current* release $prev(r)$ since it may still have unresolved bugs after its release date, and new additional bugs may have been reported by its users after the release. On the other hand, maintainers will also be involved in handling bugs of the *next* release r for which they aim to resolve as much bugs as possible before its release date.

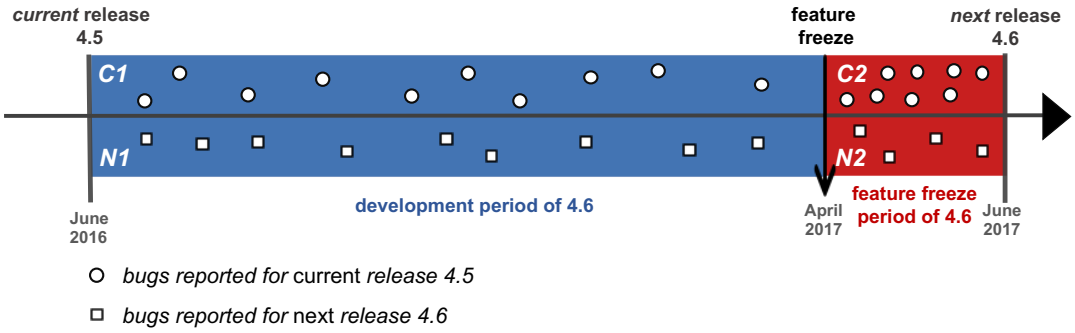


FIGURE 4.20: Categories of bugs considered for $RQ2.2$

Fig. 4.20 visually represents all these cases, assuming for the sake of the example that next release $r = 4.6$ and current release $prev(r) = 4.5$. The reported bugs for current release 4.5 are represented as white circles, and those for next release 4.6 as white rectangles. In the analysis for $RQ2.2$ we distinguish between four categories of bugs:

C1 All bugs of current release $prev(r)$ reported during development period $d^{<\text{freeze}}(r)$ of the next release r.

C2 All bugs of current release $prev(r)$ reported during feature freeze period $d^{>\text{freeze}}(r)$ of the next release r.

N1 All bugs of next release r reported during its development period $d^{<\text{freeze}}(r)$.

N2 All bugs of next release r reported during its feature freeze period $d^{>\text{freeze}}(r)$.

For each release r, and for each of these four categories, we compute triaging time $T_{\text{triage}}(b)$ for each bug $b \in B_{\text{assigned}}^{<\text{freeze}}(r)$ and $b \in B_{\text{assigned}}^{>\text{freeze}}(r)$. We compute, in a similar way, bug fixing time $T_{\text{fix}}(b)$ for each bug $b \in B_{\text{fix}}^{<\text{freeze}}(r)$ and $b \in B_{\text{fix}}^{>\text{freeze}}(r)$.

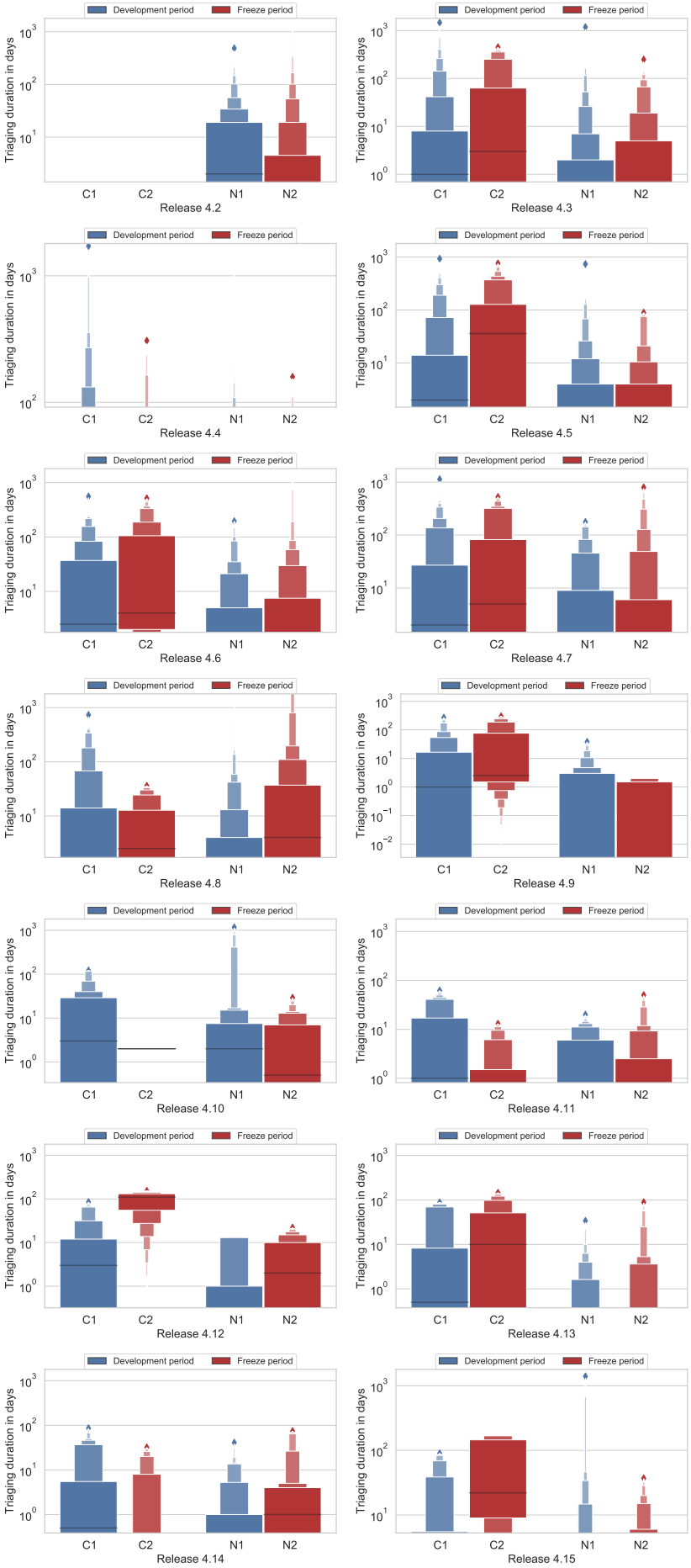


FIGURE 4.21: Boxen plots of triaging time distributions during the development and feature freeze period for each release.

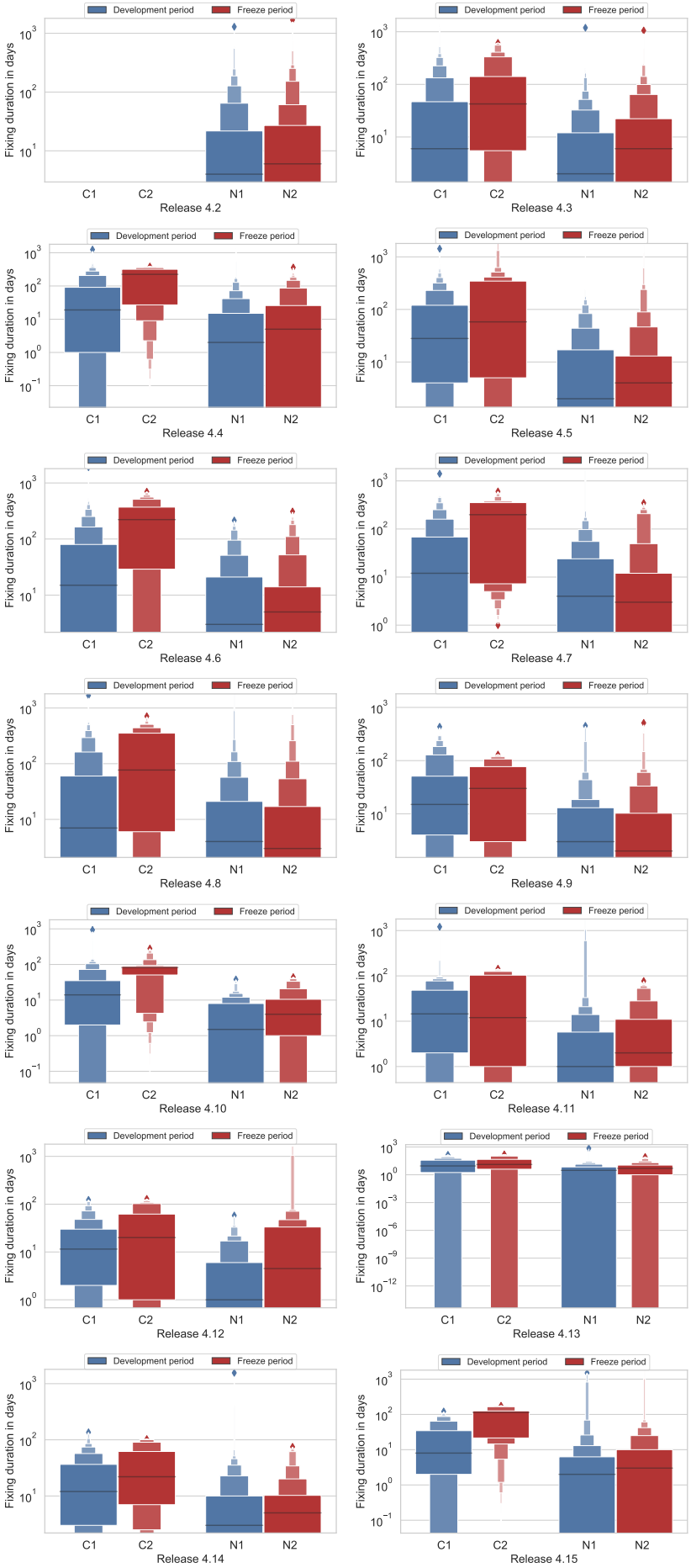


FIGURE 4.22: Boxen plots of fixing time distributions during the development and feature freeze period for each release.

	triaging time			fixing time		
release	$H0^t_{4a}$	$H0^t_{4b}$	$H0^t_{4c}$	$H0^f_{4a}$	$H0^f_{4b}$	$H0^f_{4c}$
Eclipse 4.x annual releases						
4.3	**S***	**S****	N*	**M*****	**S*****	**S*****
4.4	–	–	–	**L*****	**M*****	N***
4.5	**L*****	**S****	–	**M*****	**S***	–
4.6	**M*****	**S***	–	**L*****	**L*****	–
4.7	**S***	–	–	**L*****	**L*****	–
4.8	–	–	**S****	**L*****	**M*****	–
Eclipse 4.x quarterly releases						
4.9	–	–	–	**L*****	–	–
4.10	–	–	–	**L*****	**L*****	**S***
4.11	–	–	–	**S****	–	**S****
4.12	–	–	**M***	–	–	**S***
4.13	–	–	–	**S****	–	**S****
4.14	–	–	**S***	**M****	–	–

TABLE 4.3: Mann-Whitney U test for difference between development and feature freeze period for **triaging time** and **fixing time**. – indicates that the null hypothesis could not be rejected. Whenever it could be rejected, cell values summarise the significance level α (* =0.05; ** =0.01; ***=0.001) and the effect size: N(egligible), **S**(mall), **M**(edium), **L**(arge).

(a) Comparing bugs during the feature freeze period During the feature freeze period $d^{>\text{freeze}}(r)$ we compare the triaging and fixing time of bugs of the *current* release (C2) to those of the *next* release (N2). We visually observe that during this feature freeze period, bugs for the next release are triaged (Fig. 4.21) and fixed (Fig. 4.22) *faster* than for the current release. This can be explained by the fact that the feature freeze period aims to triage and fix the bugs of the next release fast and to deliver the release with as few bugs as possible.
For triaging time, Mann-Whitney U tests (see Table 4.3) confirm that the null hypothesis $H0^t_{4a}$, stating that the time to triage bugs of the current and next release is not different during the feature freeze period, can be rejected for all annual releases except 4.4 and 4.8. For the quarterly releases, $H0^t_{4a}$ could not be rejected. Similarly, for fixing time we verify the null hypothesis $H0^f_{4a}$ that the time to fix bugs of the current and next release is the same during the feature freeze period. $H0^f_{4a}$ can be rejected for all releases except 4.12 (see Table 4.3) with *small* effect size for release 4.11 and 4.13, *medium* for releases 4.3, 4.5 and 4.14, and *large* for the others. Our results indicate that bugs of the next release are fixed faster in the feature freeze period compared to the bugs of the current release.

(b) Comparing bugs of the current release during development and feature freeze period of the next release We investigate the differences in triaging and fixing time of bugs of the current release $prev(r)$ between the development period $d^{<\text{freeze}}(r)$ (C1 in Fig. 4.21) and the feature freeze period $d^{>\text{freeze}}(r)$(C2 in Fig. 4.21) of the next release r. We observe a longer triaging and fixing time during the feature freeze period than during the development period. These results indicate that bugs triaged and fixed during the feature freeze period have been open for a long time and that maintainers tend to focus on bugs that had lived for a long time in the current

release before releasing the next one. A possible explanation can be that maintainers tend to resolve and close bugs of current release before to deliver it in the next one.
For triaging time, Mann-Whitney U tests (see Table 4.3) verify the null hypothesis $H0^t_{4b}$, stating that there is no difference in triaging time of bugs of the current release between the development period and the feature freeze period. $H0^t_{4b}$ can be rejected with small effect size for all annual releases except 4.4, 4.7 and 4.8. For the quarterly releases, the null hypothesis could not be rejected. For fixing time, the null hypothesis $H0^f_{4b}$, stating that there is no difference in fixing time of bugs of the current release between the development period and the feature freeze period, can be rejected for all annual releases. For the quarterly releases, the null hypothesis cannot be rejected except for 4.10.
Therefore, for all annual releases, bugs of the current release took longer to fix during the feature freeze period compared to the development period. This is not the case anymore for the quarterly releases.

(c) Comparing bugs of the next release during its development and feature freeze period We investigate the differences in triaging and fixing time of bugs of the next release r between its development period $d^{<\text{freeze}}(r)$ (N1 in Fig. 4.21) and its feature freeze period (N2 in Fig. 4.21). For bug triaging time, we do not observe a difference between the feature freeze period and the development period. Maintainers appear to triage bugs of the next release as soon as possible, regardless of the considered period. Mann-Whitney U tests (Table 4.3) verify the null hypothesis $H0^t_{4c}$, stating that there is no difference in *triaging time* of the next release between the development period and the feature freeze period. $H0^t_{4c}$ cannot be rejected for the annual releases except for 4.3 and 4.8. Even for those releases where $H0^t_{4c}$ can be rejected, the effect size is *negligible* or *small* (for 4.8). For the quarterly release, $H0^t_{4c}$ cannot be rejected for the releases except for 4.12 and 4.14 with a medium and small effect respectively. This implies that there is little or no difference in bug triaging time of the next release between the development period and the feature freeze period.
For bug fixing time, we visually observe a longer time during the feature freeze period than during the development period. We verified the null hypothesis $H0^f_{4c}$, stating that there is no difference in bug fixing time of the next release during the development period and the feature freeze period. $H0^f_{4c}$ can be rejected only in release 4.3 and 4.4 for annual releases. The hypothesis was rejected for the quarterly releases except 4.9 and 4.14. For all releases where $H0^f_{4c}$ can be rejected, the effect size is *small* except for 4.4 it is negligible. This implies that, with the exception of those releases, there is very little difference in the time to fix bugs of the next release between the development period and the feature freeze period.
We also tested all the previous hypotheses after grouping per bug severity level, but we found similar results to those in Table 4.3.[18] It takes less time to triage and fix bugs of the next release compared to the current release during the feature freeze period . Bugs of the current release took longer to fix during the feature freeze period compared to the development period for annual releases but it is not the case for the quarterly releases. The feature freeze period does not affect the triaging and fixing time of bugs for the next release. As a result, the bug severity does not seem to impact the results.
In contrast to our quantitative findings, four out five of the consulted Eclipse maintainers stated that bugs are prioritized according to their severity.

[18]The analysis per severity result can be found in our replication package [62].

Summary: It takes less time to triage and fix bugs of the next release compared to the current release during the feature freeze period of the next release. Bugs of the current release that are triaged and fixed during the feature freeze period have stayed open longer compared to those in the development period for the annual releases while this is not the case anymore for the quarterly releases. The feature freeze period does not affect the triaging and fixing time of bugs for the next release. The severity of the bugs handled does not influence the results.

4.10 Discussion

Evolution of the bug handling process of Eclipse.
Changes in the bug handling process can have a direct impact on the efficiency of bug handling activities such as triaging, fixing and resolving bugs. As an example, as illustrated in Section 4.7, the Eclipse Core project developers, since June 2010, stopped using resolution statuses such as `LATER` or `REMIND` that signal that the bug will be processed later on. This decision led to a natural decrease in the resolution rate, while the fixing rate increased. In fact, these resolutions appeared to be problematic for the Mozilla community as well[19] and although these resolutions remained available later on, they no longer appeared in the default list of resolutions since Bugzilla 4.0, that was released on 15 February 2011[20]. Another example of such changes is the introduction of automated tools that helped the community in handling bugs. The introduction of an Automated Error Reporting client (AERI) was also considered as beneficial by the Eclipse community. AERI facilitates reporting errors as users do not need to create Bugzilla entries; such entries are handled by the Eclipse community based on the reports they receive. In turn, users can provide comments with their reports which are helpful when fixing bugs; according to [115] commented bug reports are more than twice as likely to be fixed compared to those without user comments. Our empirical analysis has confirmed the positive effects of AERI on the bug fixing rate.

Guideline: Before changing their bug handling process, project communities should carefully assess the pros and cons of such changes upfront, plan ahead these changes to avoid negative impacts, and measure after changing the process if the targeted improvements have been reached. Researchers need to be aware of such changes in the bug handling process when carrying out empirical studies, since the effect of these changes may play an important role in the obtained results, as we have clearly observed with our Eclipse case study.

Bug handling during feature freeze period. We observed a clear difference in bug handling activity between the feature freeze period and development period of releases. The results of *RQ*2.2 revealed that, during feature freeze, maintainers focus more on triaging and fixing bugs of the next release than on those of the current release. Also, the results of *RQ*2.1 showed that maintainers focus more on fixing bugs reported during the feature freeze period than on bugs reported earlier. While the consulted Eclipse maintainers claimed to tackle bugs of high priority during feature freeze, we observed the same proportion of severe bugs being fixed as was the case during the development period. Moreover, bugs of the current release that are triaged

[19] A relevant discussion between Mozilla developers can be found at https://bugzilla.mozilla.org/show_bug.cgi?id=35839

[20] https://www.bugzilla.org/releases/4.0/release-notes.html

during the feature freeze period have been open for a long time. This can happen because long-lived and possibly complex bugs are planned to be fixed for the next release.

> Recommendation: Intuitively, one might expect feature freeze periods to impose extra stress during rapid releases, since they tend to be relatively short and, in addition, they take away an important part of the time that could otherwise be devoted to the development of new features. Nevertheless, feature freeze periods need to be preserved, since they allow to spend more focused effort fixing bugs for the upcoming release. In addition to this, as recommended by Eclipse maintainers, rapid release adopters should invest in test automation especially in presence of rapid releases.

Benefits and challenges in switching to a more rapid release policy. All five consulted Eclipse maintainers found the transition to a quarterly release cycle very beneficial w.r.t. the time it takes for bugs to be fixed and new features to reach users. They also indicated that it helped to reduce stress and increase community growth. One maintainer stated *"Now, if I miss a deadline, it's not the end of the world, and I don't have to rush as much..."* and another one said *"Bug fixes get out in a timely manner. No more backporting of bugfixes from master to a service release."* Overall, the consulted maintainers claimed not to have faced many difficulties because of the transition, and they agreed that faster releases lead to faster feature and bug fix integration, increasing user benefit and making development more efficient. On the downside, they mentioned that beta testing on a release decreased due to less milestone releases per cycle (3 as opposed to 7 for the annual releases, cf. Fig. 4.1 & Fig. 4.2) and lack of time to contribute larger features. They also found it harder to keep development environments up-to-date and expressed the need to reorder tasks so that more release engineering work can be automated. They highlighted the importance of a good management of their platform, as well as the fact that instability and regressions might occur.

Khomh et al. [63, 64] studied the effect of the transition to a rapid release cycle on the bug handling activity in a different project, Mozilla Firefox. Considering a period of two years of bug activity, they found this transition to lead to shorter bug fixing times, but on the downside less bugs were being triaged and fixed in a timely fashion. Our results for the more recent switch of Eclipse to a quarterly release schedule partially align with these findings, as we also observed less bugs being triaged and fixed after the transition Section 4.7. Different from Firefox, however, we observed an increased fixing rate after the transition ($RQ1.1$) and bugs being fixed faster ($RQ1.2$). A possible explanation for this difference is that Firefox developers were not given enough time to prepare for the transition to smaller release cycles [64]. In contrast, the Eclipse community has been preparing the transition for over a year [11] by starting to introduce intermediate quarterly "update" releases since Eclipse 4.6 in 2016. Two of the consulted Eclipse maintainers clearly indicated that these update releases were part of the preparation. Carefully planning the transition to quarterly releases was therefore essential to its successful adoption.

As part of the preparation towards a more rapid release cycle, the consulted Eclipse maintainers highlighted the importance of a good testing plan. Adopters of rapid releases should carry out tests all along the release development cycle, not only during the feature freeze period. They should also heavily invest in adequate automated tooling and support processes, especially for continuous integration and deployment, to make it "just work".

Kula et al. [69] studied the effect of rapid release in ING, a large Netherlands-based internationally operating bank that develops in-house software solutions. They reported that ING introduced the Continuous Delivery as a Service project to automate the complete software delivery process to make shorter release cycles practical. ING put a continuous delivery pipeline in place for all teams to enforce an agile development process and reduce their testing and deployment effort.
It is important to highlight that there are limitations related to the nature of the case studies. For instance, different communities are involved in our two case studies Eclipse and Firefox. Since Eclipse is an integrated software development environment, it can be regarded as software whose main users are software developers themselves. In contrast, Firefox is a web browser, so its main users are unlikely to be developers. This difference in user communities is likely to play a role in bug handling activities. Given that Eclipse users are developers themselves, they are more likely to write good bug reports as they are more familiar with bugs. This makes it easier for maintainers to resolve them, as compared to bugs being reported by end users that are not developers, as is most often the case for Firefox.

Lesson learned: In the context of ever more projects moving to faster release cycles [57, 80], the findings for Mozilla Firefox and Eclipse highlight that careful preparation and planning is key for a successful transition to faster release cycles. It enables the developer community to become more effective in bug handling activities, provided the presence of a good testing plan, a good release management policy, and adequate automated tooling and support processes. Other projects can benefit from these lessons learned.

What is the most adequate release duration? Projects that have adopted a rapid release policy have done so with cycles of different durations. For example, Eclipse opted for a 13-week cycle, while Mozilla Firefox opted for a much shorter 4-week cycle. It remains an open question what constitutes the most optimal duration, and the answer will be specific to each project. We consulted the Eclipse maintainers about their opinion and four out of five responded that 13 weeks is an adequate duration, mentioning benefits such as "*short enough for features and bugfixes to get to users at adequate speed*" and "*long enough to allow using single stream of development thus saving the team for branch merging*". However, one of these respondents was concerned that "*a lot of process time goes into building and shipping each version, and too little time is devoted to automated testing*". Another respondent would favour an even higher release frequency, but acknowledges that this "*would require way more automation, standardization, etc. of all the releases train projects, and that seems impossible with the rather loosely coupled projects at eclipse.org*". In contrast, yet another maintainer signaled that even 13 weeks is already a very short period of time. In future research we aim to investigate how the type of release cycle (fixed or variable) can impact the bug handling activity.

Conclusion: The 13-week release cycle of Eclipse appears to be a good compromise; further shortening the release duration may be challenging and the added value of doing so is yet unknown.

4.11 Threats to Validity

Following the structure recommended by [135], we discuss the threats that may have affected the validity of our findings, and how we have tried to mitigate them.

A threat to *construct validity* in our study concerns the bug assignment identification. We minimized this threat by identifying bug assignments based on both the *Status* field (`ASSIGNED`) and the alternative practice of assigning bugs through [component_name]-triaged@eclipse.org (cf. Section 4.2.2).
A possible threat to *internal validity* is that our analysis only relied on Bugzilla bug reports. However, we verified that all Eclipse Core bugs are handled through Bugzilla; even the bugs submitted using AERI are stored in Bugzilla. Another threat stems from bugs not having a *Version* field, or having a *Version* field not corresponding to any of the considered releases. We excluded such bugs as it is not possible to automatically deal with such cases. A third known threat [127] concerns the presence of multiple occurrences of the same activity in a bug report, such as a bug that may be reassigned, a bug that may have had multiple resolutions or fixes, the *Version* field value that changes over time, and the *Severity* level that may be modified. We mitigated this threat by considering only the date of the *first* assignment (reflecting the moment when the bug was triaged for the first time), the date of the *last* resolution/fixing activity (reflecting the fact that prior resolutions/fixes of the bug were not satisfactory), the last reported *Version* field value, and the latest *Severity* level of the bug.
We quantified the possible bias stemming from bugs tagged with different major versions throughout their history. We found that 4,266 bugs were reassigned to different releases throughout their history, out of which 3,973 bugs were reassigned to different major releases. From these bugs, only 21 out of 2,741 `RESOLVED` bugs are resolved in multiple major Eclipse releases, thus the bias they introduce is insufficient to alter our findings.
Concerning possible bias due to changes in the bug severity, we found 2,016 bug reports that changed their severity over time, out of which 1,421 bugs (5% w.r.t the total number of considered bugs in the 4.x series) being reassigned to a different severity category, thus the impact on our analysis is minimal.
A fourth threat concerns our study in *RQ*2.1 and *RQ*2.2. We study the fixed bugs targeting only the upcoming release during its feature freeze, however, other bugs are fixed during this period that target other releases than the upcoming one. We measured the percentage of fixed bugs that target the upcoming release compared to the ones targeting other releases. We find that the majority of the fixed bugs during the feature freeze period of an upcoming release target it, thus, the impact of such threat is likely to be minimal.
A fifth threat concerns the presence of Eclipse Genie, an automated bot, that closes bugs that have not had any activity for a long time. It closes bugs assuming that the problem got resolved, was a duplicate of something else, or became less pressing or maybe it is still relevant but has not been triaged yet. This bot is used so that these bugs will not appear open anymore for maintainers. Starting from 2020, Genie closes bugs from the 4.x annual releases that have been open for a long time. In our analysis, bugs that are closed by Genie are not considered as resolved. Thus, our quantitative analysis is not affected by the presence of Genie.
Regarding *external validity*, we cannot generalize our results as we only analyzed a single case study of Eclipse. While the followed methodology is applicable to other systems, the obtained findings are not generalizable. Smaller and less mature projects are likely to reveal other evolutionary characteristics in their bug fixing behavior. Even for Eclipse itself, the findings are only valid for the Core projects that have a large number of bugs and an active developer community. The analysis and findings will differ for smaller and/or peripheral projects within Eclipse that have different versioning, release policies and evolutionary dynamics.

We mitigate threats to *reliability validity* by providing a publicly available replication package [62] and a detailed description of the followed methodology in Section 4.2.

4.12 Conclusion

In this work, we aim to study the impact of rapid releases on different parts of the software development processes. Software maintenance is one of the significant parts in the software development process. Bug handling comes as a priority in the software maintenance. In this chapter, we conducted an empirical study of bug handling activity in the Eclipse Core projects. We focused the study on the 4.x release range, featuring a transition to a more rapid release cycle as of release 4.9. We compared seven annual releases before this transition to seven quarterly releases after the transition. We evaluated the evolution of bug triaging time, bug fixing time, bug resolution rate and bug fixing rate. We compared these metrics *before* and *after* each release date. We also studied the impact of the *feature freeze* period on these metrics. Our finding highlights that:

For the annual releases, the number of reported bugs per release is decreasing over time, and for the quarterly releases it is stabilising at a low value, suggesting that the Eclipse bug handling process is mature and of high quality. This difference is no longer observed for the quarterly releases, mostly because of a smaller number of reported bugs. We also could not observe a difference between bugs reported before and after a release in term of triaging and fixing time.

While the bug resolution rate is decreasing over time to rather low values ($< 50\%$, implying that less than 1 out of 2 bugs gets resolved for recent releases) the fixing rate is becoming very high (close to or above 90%). This improved efficiency seems to be due to a combination of a well-managed bug handling policy and the introduction of the automated AERI error reporting tool.

We observed more intense bug handling activity during the feature freeze periods, where bugs are triaged and fixed faster, and priority is being given to fixing bugs of the next release as opposed to bugs of the current release. During these periods, more effort is being spent on bug fixing, and this is maintained after the transition to quarterly releases. In our study, we did not find any measurable effect of the bug severity on the bug handling process.

The transition from an annual to a quarterly release cycle has allowed the Eclipse Core projects to have a more stable bug handling process, since some observed differences in triaging and fixing times and rates before and after annual releases are no longer present for the quarterly releases. Moreover, we did not observe any negative effect of the switch to quarterly releases. We therefore believe that the transition to a more rapid release cycle has been beneficial to Eclipse in terms of bug handling activity. This was confirmed by feedback from five consulted Eclipse maintainers. It remains an open question if even faster release cycles would continue to yield further benefits or on the contrary would have negative consequences.

The story of Eclipse has shown that feature freeze periods and faster release cycles can be beneficial for well-managed software projects, provided that Eclipse has already put in place a well-defined bug handling process. Switching to more rapid releases requires careful planning and tracking the transition, and being aware of the possible pitfalls. Adopters of rapid releases should test and fix bugs as soon and as frequently as possible. They should raise awareness to their developer community, invest in the most appropriate tooling and support processes, and automate as much as possible.

Chapter 5

Revisiting the Impact of Rapid Releases on Software Development Activities

Many large software projects adopted rapid releases policies to deliver new features and bug fixes fast to gain and sustain the satisfaction of users. These policies can impact different activities of the software development process. For example, rapid releases might impact software maintenance activity as there is less time to handle bugs. This chapter presents an empirical study of the impact of switching to a more rapid release cycle in the Mozilla Firefox project. We revisit previous studies on the impact of rapid releases on the bug handling process and testing process. In addition, we study its impact on the patch uplift process.

5.1 Introduction

The advent of rapid release practices has significantly reduced the amount of stabilization time available for new features, forcing companies to use innovative techniques to ensure that important features are made public, timely and of good quality. While rapid release cycles allow for faster feedback from users and are easier to plan because of their smaller scope, they also significantly impact software quality. On the one hand, enterprises currently lack time to stabilize their software [117], and customer support costs increase due to frequent upgrades [59].
Porter et al. [101] noted that with short release cycles, testers have less time to test all possible configurations of a released product, which can negatively impact the quality of the software. On the other hand, other studies have reported positive effects of rapid releases on software testing and quality in the context of agile development, where testing has become more focused [76]. Software testing is an import part of the software development process, and it plays a major role in quality assurance. Mozilla Firefox switched from traditional to rapid releases in 2011. This switch has been widely studied in previous research. Mäntylä et al. [80] found that rapid releases in Firefox have more tests executed per day but with less coverage. They also observed that the number of testers decreased in rapid releases, increasing the test workload. Khomh et al. [64] found that bugs that are associated with crash reports tend to be fixed more quickly in the rapid Firefox releases than the traditional ones.
To optimize the process to cope with short release cycles, and make it more reliable for all users, over the years Firefox developed a release process that includes four 'pre-release' channels: a development channel known as *Nightly*, two stabilization channels (*Aurora* and *Beta*), and a main *Release* channel. Features for a new release are developed on the *Nightly* channel over six weeks. Afterwards, the code is transferred to *Aurora*, where it is tested by Mozilla developers and contributors, for six weeks, and then to *Beta* where a selected group of external users tests it. Finally, mature *Beta* features are imported into the main Release channel and delivered to end-users. However, release cycle time has required to subvert the model regularly over the years by uplifting new features to meet market requirements. Besides, *Aurora* channel was not meeting the expectations as a first stabilization channel. Thus, in April 2017, Firefox removed the *Aurora* stabilization phase from the process. In 2019, Firefox had many requests to bring features to market sooner. Moreover, as they believe that shorter release cycles provide greater flexibility to support product planning and priority changes due to business or market requirements, Firefox decided to move to a four-week release cycle in the first quarter of 2020. They claim that they can be more agile and ship features faster while applying the same rigour and due diligence required for a stable, high-quality release.
The release process of Firefox is frequently subverted by urgent patches, implementing high-value features or critical fixes, that cannot wait for the next release cycle. These patches are directly promoted from the development channel to stable channels (i.e., *Aurora*, *Beta*, and main *Release*), a practice called patch uplifting. Patch uplifting is risky because the time to stabilize the patches is shortened. Therefore, it is important to carefully select the patches that are uplifted and ensure that developers review them properly, to reduce the risk of regressions. There is a set of rules in place at Mozilla to control this uplift process. However, despite these rules, multiple uplifted patches still introduce regressions in the code. It is important to study to which extent a rapid release cycle impacts this practice.
In this chapter, we revisit the impact of rapid releases on the software development process in Mozilla Firefox. Different from previous studies, we study the recent

removal of *Aurora* channel to the recent switch four weeks release cycle. Following the Goal-Question-Metrics approach, we study the evolution of the Mozilla Firefox project with the aim to analyze its bug handling process, testing process and patch uplift process for the purpose of assessing the possible consequences of rapid release from the point of view of developers in the context of software development. In this thesis, we aim to provide valuable insights into the advantages and disadvantages of adopting rapid releases and the release management plans and tools needed for successful adoption. For this, the overall objective of this chapter is *to understand whether and how transitioning to a more rapid release model can impact different aspects of the development process of Firefox.* The objective is divided into two main goals, each composed of several research questions that will guide the case study design and empirical analysis:

Goal 1: The first research goal aims to study **if and how the transition to more rapid releases impacts the software quality and the testing workload**. Given the fact that developers have less time to stabilize their releases in rapid release models, this might result in more post-release bugs which might affect the software quality and user experience. Moreover, the testing phase is a vital point during the software development process, and it might be impacted because of the less time available to test all features. Hence, we analyze if there is a change in the number of post-release bugs and testing activity after the switch to more rapid releases. Four research questions will guide our investigation:

RQ1.1: ***How does switching to more rapid releases impact the number of post-release bugs?***

Shorter release cycles could negatively impact the quality of software systems since there seems to be less time for testing. Many reported bugs are likely to remain unfixed until the next release, which in turn might expose users to more post-release bugs. In this question, we analyze the daily number of reported post-release bugs, the proportion of fixed bugs, the fixing time and how and whether this changes when switching to more rapid releases.

RQ1.2: ***How does switching to more rapid releases affect the number of manually performed tests?*** This question analyzes possible correlations between reducing the release cycle and testing activity. Since the short release cycle time seems to leave less time for developers to test the system, QA teams may decide to reduce the amount of testing for their rapid releases versions in order to cope with their tight schedule. In this research question, we verify this by investigating the testing effort performed for each release model in terms of the number of test and test coverage.

RQ1.3: ***How does switching to more rapid releases affect the number of testers working on a project?***

Mäntylä et al. [80] found that the number of testers decreased after the switch from traditional to rapid releases in 2012 with an increase in the test workload. The rapid succession of releases may make it harder or easier to retain testers, and we expect that testers have less workload. In this question, we investigate how the number of testers evolves after Firefox switched to more rapid releases.

RQ1.4: ***How does the frequency of intermittent test failures change after switching to more rapid releases?*** Intermittent tests fail non-deterministically.

For example, a test may both pass and fail when performed in a different moment on the same build even though there are no changes in the tested code. As a result, developers cannot trust an intermittent test. Many intermittent test failures could be avoided by good test writing principles. Mozilla has a problem with intermittent failures that occur at an ever-increasing rate. One possible solution for such frequently failing tests is to disable them, but this means reducing the test coverage. Intermittent failures can be an indicator of testing quality. In this research question, we analyze the frequency of intermittent failures over time and if this changes with a shorter release cycle.

Goal 2: The second research goal aims to study **to which extent the patch uplifting practice is affected by switching to more rapid release cycles** as the latter may impact the acceptance rate and effectiveness of the uplifts. Our investigation will be guided by two research questions:

RQ2.1: ***How does the number of accepted and rejected uplifts evolve over time?***

As the time to stabilize a channel is shorter after reducing the release cycle, the release manager is likely to become more strict in enforcing the uplift rules. In this question, we analyze whether and how the number of accepted and rejected uplifts changes after reducing the release cycle. We expect fewer patches to be accepted after switching to rapid releases as the release manager will probably be more strict.

RQ2.2: ***How effective are the patch uplifts and how does this change with more rapid releases?***

The fact that reviewers in rapid releases have less time to review patches before uplifting might cause regressions later in the code. It is important to study whether and how shorter release cycles impact the number of regressions caused by uplifts. There is less time to review a patch in more rapid releases, so we expect to find more regressions caused by these uplifts.

5.2 Methodology

In Section 5.2.1, we introduce the selected case study that we have selected for empirically evaluating our research questions. Then, in Section 5.2.2, we present the experimental design, the data extraction process and the metrics that will be used to answer our research questions. The datasets and scripts generated for this study are publicly available in a replication package in [61].

5.2.1 Selected Case Study: Mozilla Firefox

For our empirical study, we have selected Mozilla Firefox as a case study because it is a long-lived open source project, with a large community of contributors. Firefox is a free open source web browser developed by the Mozilla Foundation. Firefox reduced its release cycle multiple times through its lifecycle. It has been studied widely in studies about software evolution [6, 12] and the impact of rapid releases [27, 64, 80]. Firefox version 1.0 was released on November 2004 and Firefox followed a traditional release model until version 4.0 (March 2011). Starting with version 5.0, Firefox adopted a rapid release model to accelerate the delivery of its new features.

Then, the process shifted to a four-week cycle with the release of Firefox 71 on March 10, 2020.

5.2.1.1 Mozilla's Development Process

In 2011, Firefox followed a pipelined release process, with four release channels [94]. The four channels available are *Nightly* (a.k.a "Central"), *Aurora*, *Beta*, and *Release* (RC) of 6 week intervals (see Fig. 5.1). The *Nightly* channel gets new features once they are ready, and it has the lowest stability of the four channels. The *Aurora* channel gets new features on a regular basis, but some of them might be disabled if it appears to require more work. The *Beta* channel receives only new features that are scheduled for the next Firefox release. New features are rarely directly added to the *Aurora* or *Beta* channels. The number of users on each channel increases about ten times as the changes make their way through the release process, and features on each channel require an accompanying improvement in the stability.
Aurora was created in 2011 to provide more user feedback after Firefox shifted from version 5 to the rapid release cycle. It was initially intended to be the first stabilization channel to provide further user feedback. This original intent never materialized. The release cycle time has required Firefox to regularly subvert the model over the years by uplifting new features to meet market requirements. In order to address the complexity and cycle duration issues, in April 2017, the release management team, in coordination with Firefox product management and engineering, decided to remove the *Aurora* stabilization phase from the cycle reducing the release cycle by 6-8 weeks [93] (see Fig. 5.2). The *Aurora* cycle was used to finalize some features. Instead, features will be stabilized during the *Nightly* cycle. To improve the overall quality of Nightly, Firefox follows a few initiatives:

Nightly merge criteria: New end-user facing features landing in *Nightly* builds should meet Beta-readiness criteria [1]before being pushed to *Beta* channel.

Static analyzers: To detect issues at the review phase, static analyzers are integrated as part of the workflow. They are able to identify potential defects but also limit the technical debt.

Code coverage: Code coverage results are used to analyze the quality of the test suite and the risk introduced by the change.

Risk assessment: Identify the potential risks carried by changes before they even land by correlating various data sources (VCS, Bugzilla, etc.). The idea is to identify the functions where a modification has more chance to induce regression(s).

[1]See a list of criteria that evaluate feature readiness to merge to *Beta* in [93]

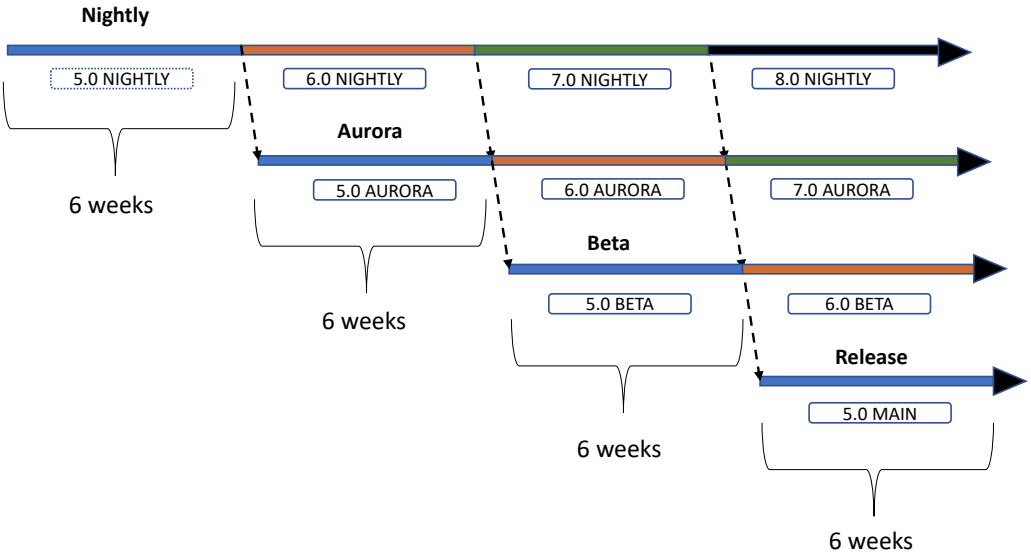


FIGURE 5.1: Development and Release Process of Mozilla Firefox for releases 5 till 53.

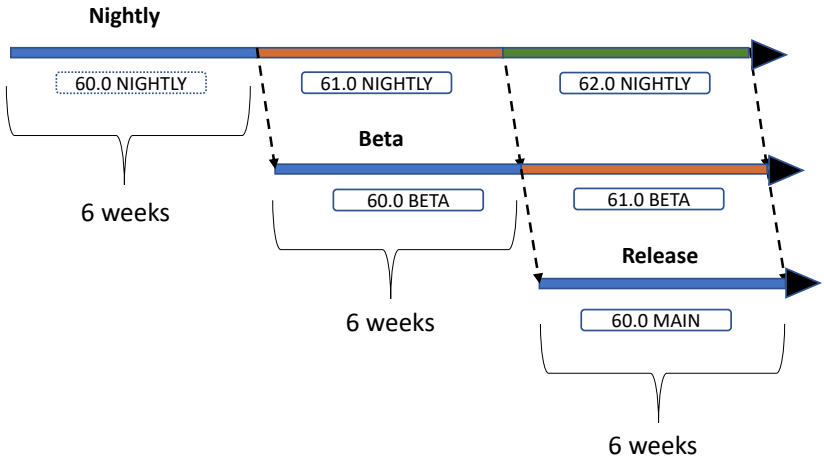


FIGURE 5.2: Development and Release Process of Mozilla Firefox for releases 54 till 70, after the removal of *Aurora* channel.

Starting from Firefox 71, a new major Firefox release is shipped every 4 weeks[2]. Fig. 5.3 shows the current development and release process of Mozilla Firefox. Firefox's release management team believes that a shorter release cycle provides greater flexibility to support product planning and priority changes due to business or market requirements. With four-week cycles, they can be more agile and ship features faster while applying the same rigor and due diligence needed for a high-quality and stable release.

Fig. 5.4 shows the timeline of the major releases of Firefox. In this chapter, we will refer to the release model with 6-weeks interval till release 53 (before the removal of *Aurora*) by **RR6**, the 6-weeks release model until release 71 by **RR6-A** and the release model of 4-weeks by **RR4**.

[2] https://hacks.mozilla.org/2019/09/moving-firefox-to-a-faster-4-week-release-cycle/

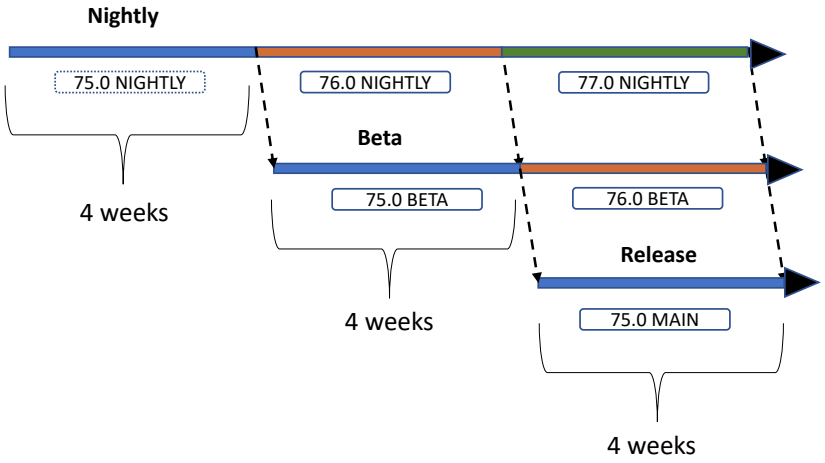


FIGURE 5.3: Development and Release Process of Mozilla Firefox since release 71.

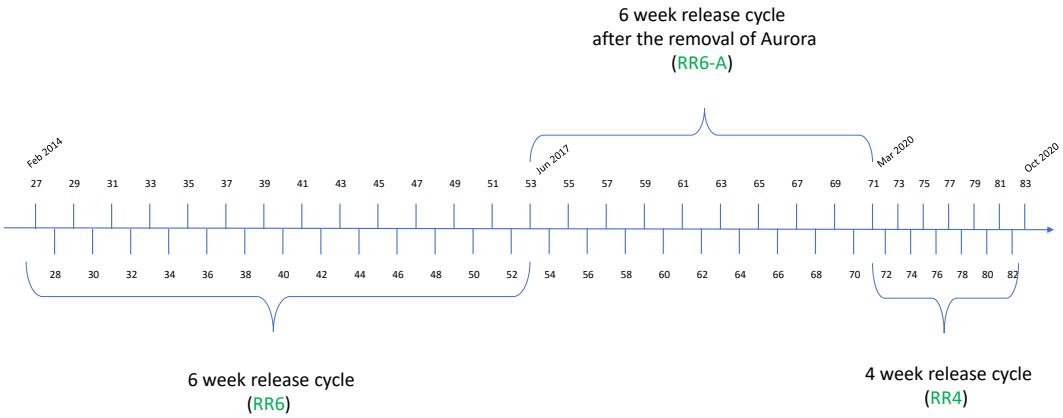


FIGURE 5.4: Timeline of major Firefox versions.

5.2.1.2 Firefox Testing & Quality Assurance

Mozilla's QA team leads software quality assurance activities throughout Mozilla and plays a key role in the delivery of a variety of software products on time. In every Mozilla project, they explore new functionality, write and run tests, discover and file bugs, build and maintain tools, collect and analyze metrics. As was the case for Eclipse in Chapter 4, all Mozilla projects use Bugzilla for reporting and handling bugs.

For manual testing, TestRail is a test case management tool used by most Mozilla QA teams to catalogue and monitor test cases during the lifetime of a project. TestRail is the unique source of truth for all testing data (test plans, reports, failures, etc.) and is thus an important component in the overall software life cycle at Mozilla. Fig. 5.5 shows a test plan for Firefox 71. Typically, when a project starts, a QA/Test Engineer will create a test plan for the feature/project. This process will begin once the product manager has a clear idea of what needs to be built. The test plan must be reviewed and approved before any code is written. Once the engineer feels confident in the test plan, he/she enters the test plan into Testrail. When a testable build is ready (*Nightly*, *Beta*, *Release*), the test plan may be executed using the "test run"

feature of TestRail. A test run captures the results of the execution of all test cases against a version of the product and allows the creation of reports to distribute to stakeholders.

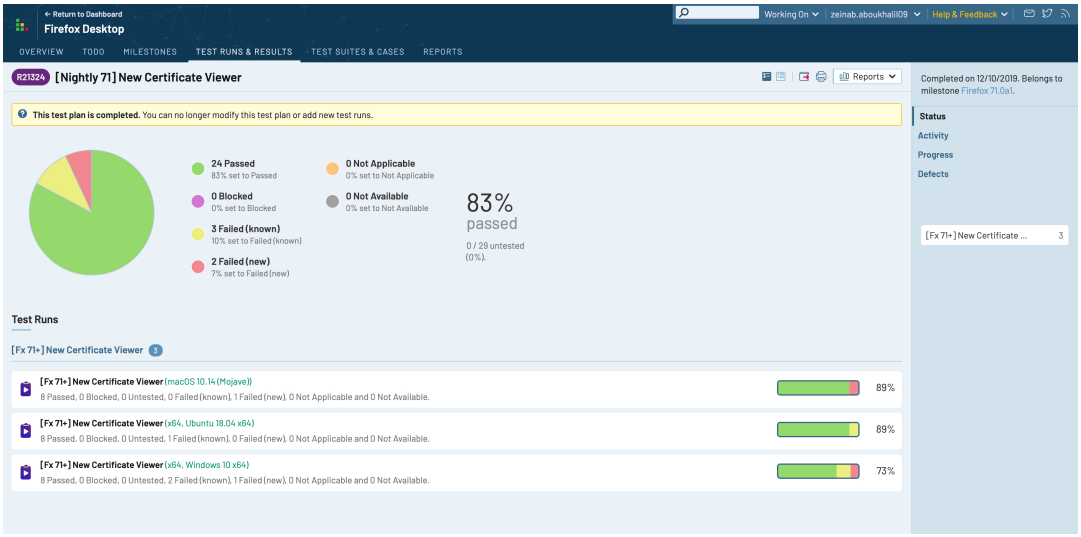


FIGURE 5.5: Screenshot of a test plan on TestRail.

5.2.1.3 Mozilla's Patch Uplifting Process

The majority of the development work is done in the *Nightly* channel, where patches can be committed after a regular review process. To keep the channels as stable as possible as code committed to *Aurora* and *Beta* gets closer to be released, a different process for committing patches has been installed for the stabilization channels: **patch uplifting**. Patches with important features or severe issue fixes that cannot wait for the next major release are promoted directly from the development channel to one of the stable channels, omitting the stabilization phase on one or more channel(s). The lifecycle of an uplifted patch goes through a series stages: developers write a patch, which gets reviewed by one or more reviewers. The patch is committed to the *Nightly* channel after a successful review. If developers believe that the patch is critical (e.g., it fixes a frequent crash), they can request to uplift the patch to one (or more) of the stable channels. Release managers, who are independent and different from the reviewer, are responsible for deciding which patches can be uplifted. After careful consideration of the risks involved, they accept or reject the patch uplift request.
The more a channel is stable, the higher is the approval bar for uplift requests. Below we present a selection of the uplifting rules on the different channels[3].

> ***Aurora*:** Uplifts to the *Aurora* channel are less critical because they still have considerable time for stabilization. Thus, the rules are not very strict: no new features are accepted; no disruptive refactorings; no massive code should change; no user-visible strings changes unless the localization team is aware of them and has approved them.
>
> ***Beta*:** Uplifts to the *Beta* channel are more critical because they have less time for stabilization. In addition to the rules defined for *Aurora*, the uplifted changes to the *Beta* channel should be (1) ideally reproducible by QA, so that

[3] https://wiki.mozilla.org/Release_Management/Uplift_rules

they can easily be verified; (2) they should have been verified on *Aurora/Nightly* first and should have demonstrated a decrease in crash or reproducibility, and (3) should not include changes to the user-visible strings in the application. The uplifted changes can lead to performance improvements, fixes to top crashes, and fixes for recent regressions. The release managers become more strict in enforcing these rules as the release date comes closer.

***Release*:** In general, uplifts to the Release channel are not recommended unless an issue cannot wait till the next major release. Possible uplifts can be fixes for major top crashes with evidence of impact provided, security issues, functional regressions with a broad impact or problem in a major feature. Once a patch is accepted for uplift, code sheriffs or the developers themselves can commit it to the stabilization channel(s) where the patch was approved.

5.2.2 Data Processing

Fig. 5.6 shows an overview of our data collection or data processing approach. We describe each step of the approach below. Many Firefox Desktop bugs will either be filed in the Firefox product or the Core product[4].
We retrieved 268,603 issues reported for Firefox and Core products between February 2014 (release date of Firefox 27) and October 2020 (release date of Firefox 83), available in Mozilla's Bugzilla bug tracker. Our dataset was fetched on 17 Oct 2020, and the earliest and latest dates of reported issues in our dataset correspond to 1 Jan 2014 and 5 Oct 2020, respectively. We differentiate issues that are related to faults, from new feature requests or improvements using the strategy of Castelluccio et al. [22]. To automatically identify fault-related issues (bugs), they use a keyword-based heuristic to search for information in the title, description, flags, and user comments of each issue report. Their list of keywords includes: "crash", "regression", "failure", "leak", "steps to reproduce (STR)", and "hang". The total number of bugs we get is 175,458.

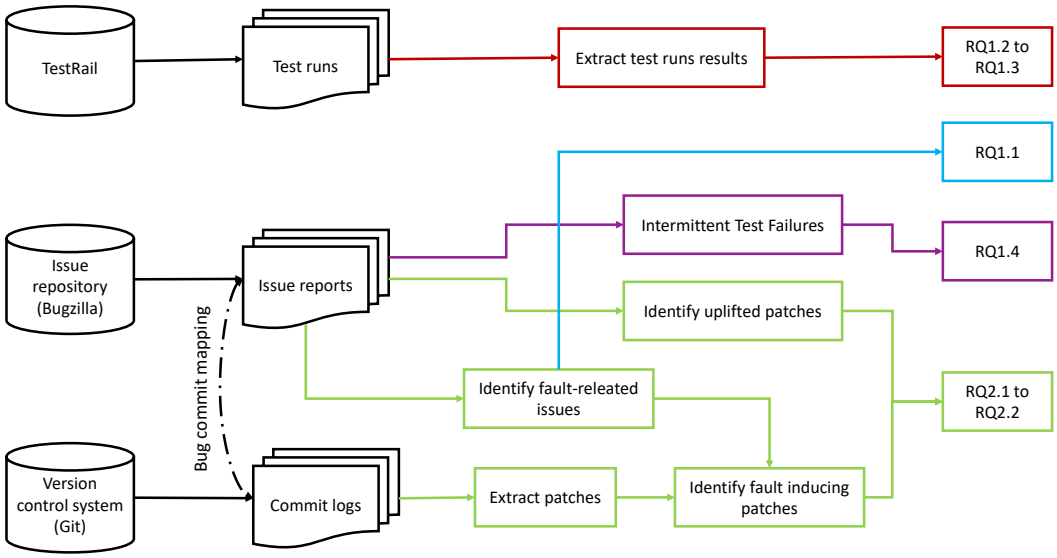


FIGURE 5.6: Overview of our data collection and processing approach

[4] https://bugzilla.mozilla.org/describecomponents.cgi?product=Firefox

Post-release bugs: To determine the post-release bugs, we filtered bugs based on their Version field. However, given that the version field is not mandatory in Bugzilla, not all the reported bugs are assigned to a version. For the bugs with specified version, there is only a major release number (50.0–83.0) with no explicit information of the individual release (alpha, beta, minor) each bug was related to. We found 112,209 bugs with unspecified Version field. To get the version of the unspecified bugs, we used the *"Tracking Flags"* field. Tracking flags are used by developers, triagers, QA and the release management teams to keep track of bugs whose fixes are slated to go into a particular product release. The tracking status flag shows the release where the bug is present or the release(s) that are affected by it. We linked 14,364 bugs to the release where they are found. The total number of bug reports with specified and unspecified versions are 96,636 and 77,613, respectively. Fig. 5.7 shows the evolution of reported bugs with specified and unspecified versions. For the analysis in RQ1.1, we exclude the bugs with unspecified versions because we focus on post-release bugs.

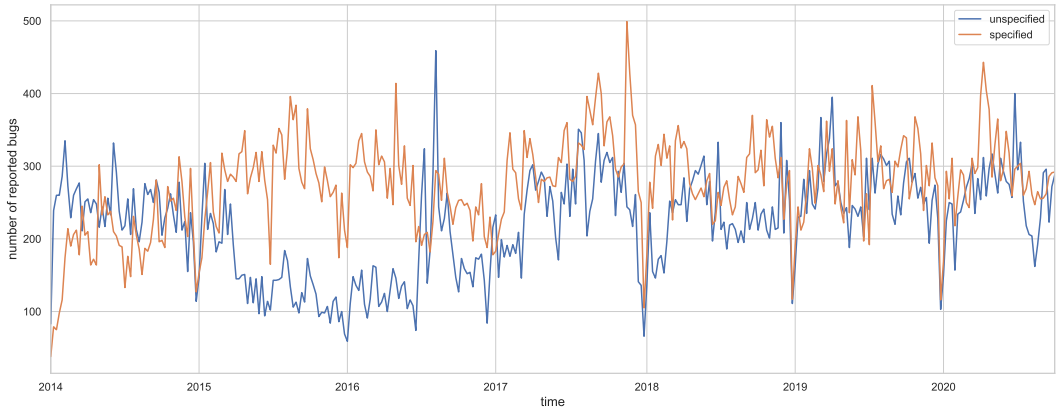


FIGURE 5.7: The distribution of specified and unspecified bugs over time

Intermittent Test Failures: Intermittent (aka flaky[5]) failures are test failures which happen intermittently, in a seemingly random way. It is often a test that passes fine locally on a machine, but when ran thousands of times on various CI environments (some of them under heavy load), it may start to fail randomly. Many of such failures could be avoided by good test writing principles [26]. Mozilla has a problem with intermittent failures that occur at an ever-increasing rate [124]. Thus, they initiated the Stockwell project to help reduce the intermittents failures. This project defines a scenario where very frequently failing tests get disabled. This should ideally be avoided because this means reducing the test coverage [6].
These intermittent failures are tracked in Bugzilla. When a test starts being intermittent a bug is filed in Bugzilla (usually by a Mozilla code sheriff). Once the bug exists for a given test failure, all further similar failures of that test will be reported as comments within that bug. These reports are usually posted weekly and look like this:

> "6 failures in 4850 pushes (0.001 failures/push) were associated with this bug in the last 7 days[7]."

[5]Intermittent failures are also referred to as flaky failures in research [36]

[6]https://firefox-source-docs.mozilla.org/devtools/tests/debugging-intermittents.html

[7]See example https://bugzilla.mozilla.org/show_bug.cgi?id=1593230#c4

Sometimes, tests start failing more frequently, and these reports are then posted daily. The information on Bugzilla is limited to the number of failures/frequency of failures (overall, per-platform and channel) for intermittently failing tests. We processed all the comments of the bugs related to the intermittent failures for all the retrieved bugs. We found 30,957 out of 175,458 bugs (17.6 %) related to intermittent failures.

Manual Testing Data We used the TestRail API [8] to extract the test plans, test runs and test execution result data for official Firefox versions 50 to 83. We consider data only since November 2016, since a test case manager called MozTrap[9] was used before this date and it was replaced by TestRail because of its slow maintenance[10]. Overall, we identified 2,240 unique test plans with 19,197 test runs for a total of 299,416 test executions across 4 years of testing (11/2016–11/2020). During this time frame, the Firefox project made 36 major releases, of which 5 RR6, 17 RR6-A and 14 RR4 releases reported their testing activity into the TestRail system. In TestRail, all test runs provide the following information: major version number of the release (50.0–83.0), unique identifier, summary, defect identifier (if any), creator identifier, the test plan the test belongs to (if any), and links to the corresponding test execution results. Each test execution contains the following information: status ("pass"/"fail"/"test unclear or broken"), test case identifier, time-stamp, platform (e.g., Windows), operating system (e.g., Windows XP), build identifier, creator identifier, referenced bugs (if any), comments (if any) and the test logs (if any). However, there was no explicit information of the individual release (alpha, beta, release-candidate, major or minor) each test execution was related to.

Patch Uplift Analysis: Mozilla developers request for patch uplifts on Bugzilla. Fig. 5.8 shows a fragment of an issue report history for where a developer requests for a patch uplift. Developers use customized Bugzilla flags for their requests. These flags have the form approval-mozilla-CHANNEL, where CHANNEL can be *Aurora*, *Beta*, or *Release*. The postfix of the flag is set to a question mark (?) when a developer asks for an uplift. The release managers will make a risk assessment to accept (set the flag to +) or reject the uplift (set the flag to -). We rely on these flags to identify uplifted patches. We relied on the methodology of Castelluccio et al. [22] to identify and analyze the patches. At Mozilla, release managers usually inspect all patches in an issue report before deciding whether they can be uplifted together. Thus, we consider uplift characteristics at the issue level. If an issue contains multiple patches, we bundle the patches together. We computed the number of patches that were uplifted each month, starting from September 2014 to February 2020. We extracted several metrics from Bugzilla reports: information about the review process (e.g., how long a review took, how many reviewers inspected a patch), information about the uplifting process (e.g., whether an uplift was accepted, how long before a release did the manager decide to accept or reject an uplift request) and the developer assigned to an issue.

[8] https://www.gurock.com/testrail/docs/api
[9] https://wiki.documentfoundation.org/MozTrap
[10] https://groups.google.com/g/mozilla.dev.quality/c/Sa75hV8Ywvk

tnikkel	2016-07-08 11:34:52 PDT	Attachment #8769302 Flags		review?(mstange)
mstange:moz	2016-07-08 11:50:06 PDT	Attachment #8769302 Flags	review?(mstange)	review+
tnikkel	2016-07-11 22:48:49 PDT	Attachment #8769997 Flags		review?(mstange)
mstange:moz	2016-07-12 14:29:45 PDT	Attachment #8769997 Flags	review?(mstange)	review+
tnikkel	2016-07-12 14:30:28 PDT	Flags	needinfo?(tnikkel)	
cbook	2016-07-13 00:24:59 PDT	CC		cbook
		Flags		needinfo?(tnikkel)
cbook	2016-07-14 02:48:10 PDT	Status	ASSIGNED	RESOLVED
		Resolution	---	FIXED
		Target Milestone	---	mozilla50
		Closed		2016-07-14 09:48:10
		status-firefox50	---	fixed
		Comment 47 Tag		bugherder
tnikkel	2016-07-15 13:32:23 PDT	Attachment #8754303 Flags		approval-mozilla-aurora?
tnikkel	2016-07-15 13:36:51 PDT	Flags	needinfo?(tnikkel)	
bugs	2016-07-15 17:27:53 PDT	CC		bugs
		status-firefox48	affected	wontfix
gchang	2016-07-28 05:33:06 PDT	Attachment #8754303 Flags	approval-mozilla-aurora?	approval-mozilla-aurora+
ryanvm	2016-07-28 14:40:33 PDT	Flags		in-testsuite+
ryanvm	2016-07-28 14:49:05 PDT	status-firefox49	affected	fixed
		Comment 54 Tag		uplift
		Comment 54 Tag		bugherder
cbook	2016-09-16 04:02:30 PDT	Comment 56 Tag		bugherder

FIGURE 5.8: Fragment of an issue report for where a developer requests for patch uplift

To capture the characteristics of patches that were uplifted, we computed the 11 metrics defined in [22] to find the characteristics of patches in the issue reports that include patch uplift requests. These metrics are presented in Tables 5.1, 5.2 and 5.3. These metrics correspond to the following five dimensions:

Developer experience and participation metrics (m_1 - m_5): The rationale for computing these metrics is that patches written or reviewed by experienced developers may have a higher chance to be accepted for uplift, and may be less fault-prone. Long comments and long review durations may indicate the complexity of an issue and developers' uncertainty about it, which may explain its rejection or fault-proneness.

Metric	$\mathbf{m}_i$	**Description**
Developer experience	m_1	Number of previous commits of the developer requesting the patch uplift in the issue report.
Reviewer experience	m_2	Number of previous commits of the reviewer reviewing the patch in the issue report.
Number of comments	m_3	Number of comments in the issue report requesting a patch uplift.
Comment words	m_4	Average number of words in the comments to the issue requesting a patch uplift.
Review duration	m_5	Time period (in days) from a patch's submission in an issue until its approval.

TABLE 5.1: Developer experience and participation metrics (m_1 - m_5) for each issue report.

Uplift process metrics (m_6 - m_8): We computed metrics capturing the uplifting process for the following reasons. Release managers may be more inclined to accept patches with higher landing delta (as the more time a patch has been on the Nightly channel, the more time Nightly users have tested it). Patches with low release delta

are likely to be refused uplifts since patches that are developed closer to the date of release might pose more risk (as there is less time to fix potential regressions). Patches with low response delta may also be rejected (since developers have less time to evaluate the risks associated with the patch). Patches with low landing delta, low release delta, and low response delta may also lead to faults if uplifted.

Metric	$\mathbf{m}_i$	Description
Landing delta	m_6	Time elapsed (in days) between when the patch was applied to the Nightly version and when the developer asked for approval of an uplift.
Release delta	m_7	Time elapsed (in days) between when the developer asked for approval for the uplift and the date of the following release.
Response delta	m_8	Time elapsed (in days) between when the developer asked for approval for the uplift and the date of the following release.

TABLE 5.2: Uplift process metrics (m_6 - m_8) for each issue report.

Code complexity (m_9 - m_{11}): Previous works, such as [66], have shown that complex code is likely to introduce faults. We calculated code complexity metrics to understand how uplifting decisions and their success are affected by the complexity of the uplifted patches.

Metric	$\mathbf{m}_i$	Description
Patch size	m_9	Number of lines in a patch (excluding test patches).
Prior changed times	m_{10}	Number of previous commits that modified the same files that the patch is modifying.
Test patch size	m_{11}	Number of lines in a test patch.

TABLE 5.3: Code complexity metrics (m_9 - m_{11}) for each issue report.

5.2.3 Statistical Methods

The quantitative analysis in this chapter will rely on a range of statistical tools. Most of our analysis aims to compare two populations by testing if their distributions are different. We test all statistical hypotheses for different significance levels α. We reject a null hypothesis if $p < \alpha$, and denote this with * if $\alpha = 0.05$, ** if $\alpha = 0.01$, and *** if $\alpha = 0.001$.
Since software engineering data often do not meet the normality assumption [77], we select appropriate non-parametric tests that do not require this assumption [54]. Normality is tested for both populations that are compared using the Kolmogorov–Smirnov test. For non-normal distributions we use the *Wilcoxon rank sum test* if the samples are related; otherwise, we use the *Mann–Whitney U test*. Since we perform more than one comparison on the same dataset, to reduce the chances of obtaining false positive results, we use Bonferroni correction to control the family wise error rate [32]. Concretely, we calculate the adjusted p-value, which is multiplied by the number of comparisons. Whenever we obtain statistically significant differences between metric values, we compute the *effect size* using Cliff's delta d [25, 52] and interpret the results using [52]. In all cases where the null hypothesis is rejected, the sign of d allows us to determine which of both distributions is higher than the other one.
The analysis of *RQ*1.1, reported in Section 5.3, uses the technique of *survival analysis* to model the expected time duration until the occurrence of a specific *event of interest* with the aim to estimate the survival rate of a given population [1]. A common non-parametric statistic used to estimate survival functions is the Kaplan-Meier

estimator [58]. To compare survival curves, we use the *multivariate log-rank* analysis using non-parametric ANCOVA based on the test statistic with equal weights to test the null hypothesis that there is no difference between the populations in the probability of an event at any time point [114]. This test is a generalization of the log-rank test and it can deal with n>2 populations (and should be equal when n=2).
The analysis of RQ1.1 and RQ1.2 reported in Section 5.3, uses boxen plots [53] to show the distributions of the dataset. These plots visualize different quartile values and convey precise estimates of head and tail behavior.

5.3 Impact of Rapid Releases on Quality Assurance

Our first research goal aims to study if and how the transition to more rapid releases impacts the software quality in terms of the number post-release bugs. Developers have less time to stabilize their releases in rapid release models. This might result in more post-release bugs which might affect the software quality and user experience. Moreover, the testing phase is a vital point during the software development process, and it might be impacted because of the less time available to test all features. Hence, we analyze if there is a change in the number post-release bugs and testing activity after the switch to more rapid releases. Our investigation will be guided by four research questions:

5.3.1 RQ1.1 : How does switching to more rapid releases impact the number of post-release bugs?

Developers seem to have less time for testing when speeding the release cycle; thus, we expect that reported issues are likely to remain unfixed until the next release, which in turn might expose users to more, and more serious, post-release bugs. Moreover, we might expect that the developers now have less time to fix the same stream of bugs. Similarly, since the next release follows hot on the heels of the current release, bugs reported by end users for the current release might not be fixed immediately. Hence, in this research question, we investigate the daily number of reported bugs and the speed at which post-release bugs are fixed in the different release cycles.
We compare the number of reported and fixed post-release bugs reports between the different release cycles groups (i.e., RR6, RR6-A and RR4). Note that we cannot perform this comparison directly. Herraiz et al. [51] have shown that the number of reported post-release bugs of a software system is related to the number of deployments, since a larger number of deployments increases the likelihood of users reporting a higher number of bugs. As the number of deployments is affected by the length of the period during which a release is used, and this usage period is directly related to the length of the release cycle, we need to normalize the number of post-release bugs of each release to control for the usage time. Khomh et al. [63] proposed a way to control for this, namely by dividing the number of reported post-release bugs of each major release by the time in days between its release date and the next major release date. They report that it makes sense because they found that the number of crash reports for an old release quickly drops once a new release becomes available. For instance, if release 51.0 is released 42 days after release 50.0, we divide the number of post-release bugs of release 51.0 by 42.

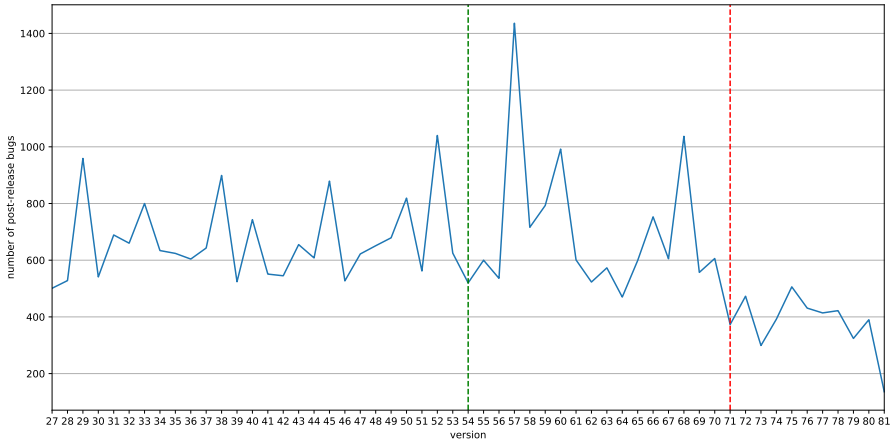


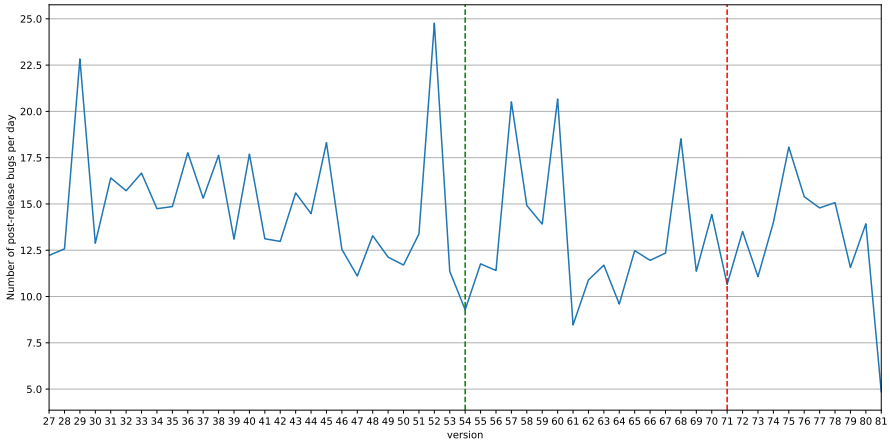
FIGURE 5.9: Number of reported of post-bugs for each release. The green vertical dashed line indicates the date of exclusion of the *Aurora* channel. The red dashed line correspond to the date of switching to the four-week release cycle (RR4).[11]

Reported Post-release Bugs. Fig. 5.9 shows the absolute number of post-release bugs for major releases. We observe that there are more post-release bugs after the removal of *Aurora*. Then the RR4 releases have fewer numbers of post-release bugs than the remaining releases. However, when we normalize by the time in days between successive releases, Fig. 5.10a shows that releases with the highest number of post-release bugs reported per day belong to RR6 releases. Fig. 5.10b shows the distribution of the normalized number of post-release bugs for RR6, RR6-A and RR4 releases. The median number of daily reported bugs is very similar for RR4 and RR6. Overall, the median number of post-release bugs of RR6 releases is similar to the number of post-release bugs for RR4 releases. We test the following null hypotheses using the Wilcoxon rank sum test for each two-pair release cycle groups:

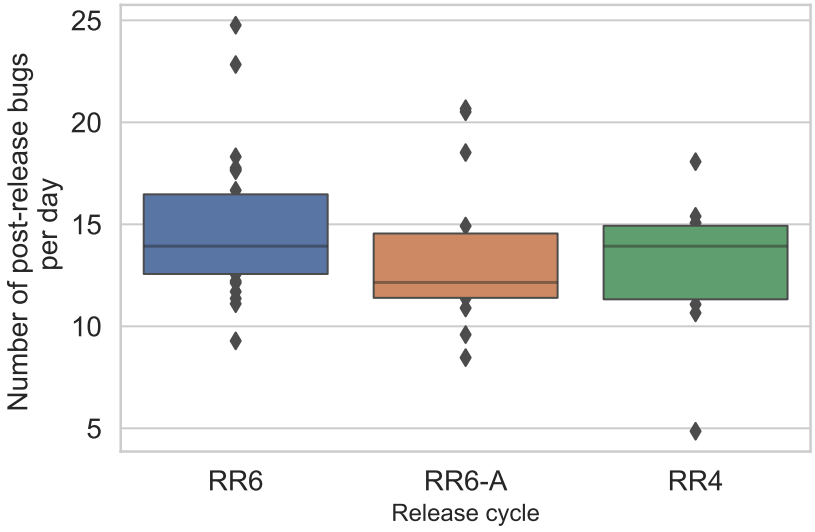
- H_{01}^{1} : *There is no significant difference between the daily number of reported post-release bugs of RR6 releases and RR6-A releases.*
- H_{02}^{1} : *There is no significant difference between the daily number of reported post-release bugs of RR6-A releases and RR4 releases.*
- H_{03}^{1} : *There is no significant difference between the daily number of reported post-release bugs of RR6 releases and RR4 releases.*

Using Wilcoxon rank sum test, we cannot reject H_{01}^{1}, H_{02}^{1} and H_{03}^{1}. The bug reporting activity is not different in the three release models.

[11] In all our figures, the green vertical dashed line indicates the date of exclusion of *Aurora* channel and the red dashed line correspond to the date of switching to 4-weeks release cycle.



(A)



(B)

FIGURE 5.10: Distribution (A) and Boxenplot (B) of the normalized number of reported post-bugs per day.

Fixed Post-release Bugs. We found that when we control for the time in between releases, there is no significant difference between the number of post-release bugs reported per day of RR6 releases and RR4 releases. However, since a shorter release cycle time allows less time for testing, we might expect that the developers now have less time to fix the same stream of bugs. Similarly, since the next release follows hot on the heels of the current release, bugs reported by end users for the current release might not be fixed immediately. Hence, in this question, we investigate the proportion of post-release bugs fixed and the speed at which post-release bugs are fixed in the different release models. For each major release, we compute the following metrics:

- # Fixed Bugs: the number of post-release bugs that are closed with the status field set to FIXED.
- Fix Time: the duration of the fixing period of a FIXED post-release bug, i.e., the difference between the bug creation time and the last fixing time.

Similar to the number of reported bugs, the number of fixed bugs also depends on the length of the release cycle. Since we are mainly interested in how many post-release bugs are fixed relative to all reported post-release bugs, we use a proportion (i.e., percentage) in this case. Hence, we test the following null hypotheses to compare the efficiency of post-release bug fixing activities of the three releases in each of the release models (RR6, RR6-A and RR4):

- H_{04}^{1} : *There is no significant difference between the proportion of fixed post-release bugs of RR6 and RR6-A releases.*
- H_{05}^{1} : *There is no significant difference between the proportion of fixed post-release bugs of RR6-A and RR4 releases.*
- H_{06}^{1} : *There is no significant difference between the proportion of fixed post-release bugs of RR6 and RR4 releases.*

Using Wilcoxon rank sum test we can reject H_{04}^{1} ($p < 0.001$), H_{05}^{1} ($p < 0.01$) and H_{06}^{1} ($p < 0.001$) with large effect size. This confirms our observation in Fig. 5.11, a smaller proportion of bugs is fixed when switching to a more rapid release.

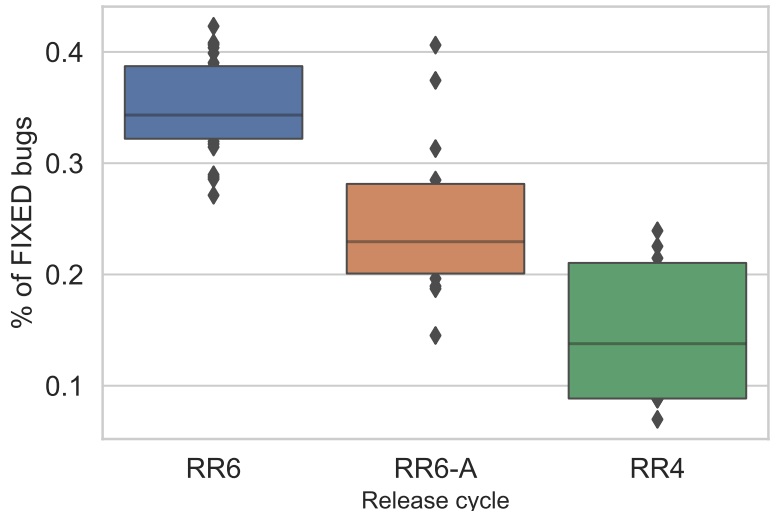


FIGURE 5.11: Boxenplots of the proportion of fixed post-release bugs.

For each release model, we compute bug *fixing time* for the releases that are related to it. We use survival analysis on these populations to model the expected time duration until the bug gets fixed for the last time. Fig. 5.12 shows the Kaplan-Meier survival curves for specific releases, together with their confidence intervals, before and after the release date. The graph reveals that there is a small difference between the fixing time of bugs in RR6 and RR6-A. However, we observe that bugs of RR4 are fixed faster than the bugs of RR6 and RR6-A. For example, the red dotted line reveals that it takes around 40 days to fix 80% of all post-release bugs in RR4, while this is more than twice as long for RR6-A (100 days) and even longer for RR6 (125 days). To assess and compare the speed at which post-release bugs are fixed under different release models, we test the following null hypothesis:

- H_{07}^{1} : *There is no significant difference between the survival distribution of Fix Time values for post-release bugs related to RR6, RR6-A and RR4 releases.*

We use a multivariate log-rank test log-rank test to verify the null hypothesis H_{07}^1. H_{07}^1 can be rejected ($p < 0.001$), thus, there is a statistical difference between the bug fixing time between different release model. Bugs of RR4 release are fixed faster than bugs of RR6 and RR6-A. Bugs of RR6-A are fixed faster than bugs of RR6. The fixing time is decreasing over time.

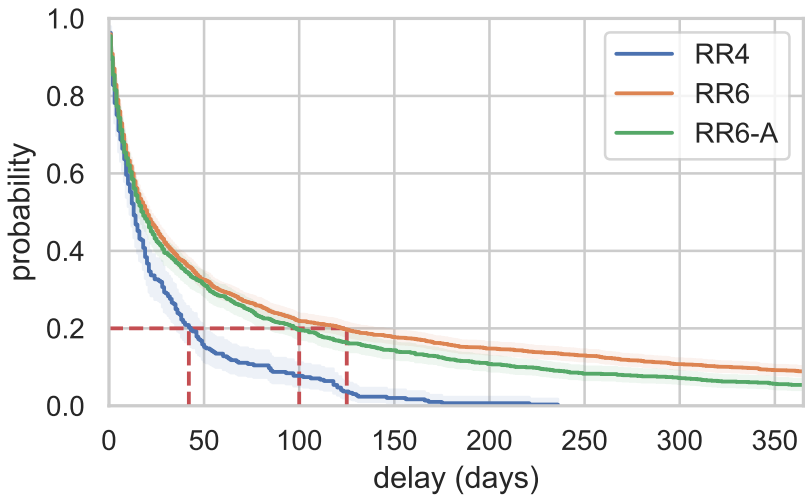


FIGURE 5.12: Kaplan-Meier survival curves with 95% confidence interval (indicated by the shaded areas) for *fixing time* of post-release bugs for releases in each release models.

Summary: The reporting activity in the different release models is similar in terms of number of reported bugs. Bugs are fixed significantly faster in the more rapid release model, however, a smaller proportion of bugs is actually fixed for them.

5.3.2 RQ1.2: How does switching to more rapid releases affect the number of manually performed tests?

Shorter release cycle time might allow less time for developers to test the system, QA teams could decide to reduce the amount of testing for their faster releases in order to cope with their tight schedule. In this research question, we verify this by investigating the amount of testing effort performed for the three release models RR6, RR6-A and RR4 of Firefox. For each version of Firefox in our data set, we compute the number of tests executed. The number of tests executed captures the amount of testing performed per release. Fig. 5.13 shows that the number of executed test is increasing over time. A linear regression analysis confirms an increasing linear trend for the amount of testing performed per release ($R^2 = 0.76$).

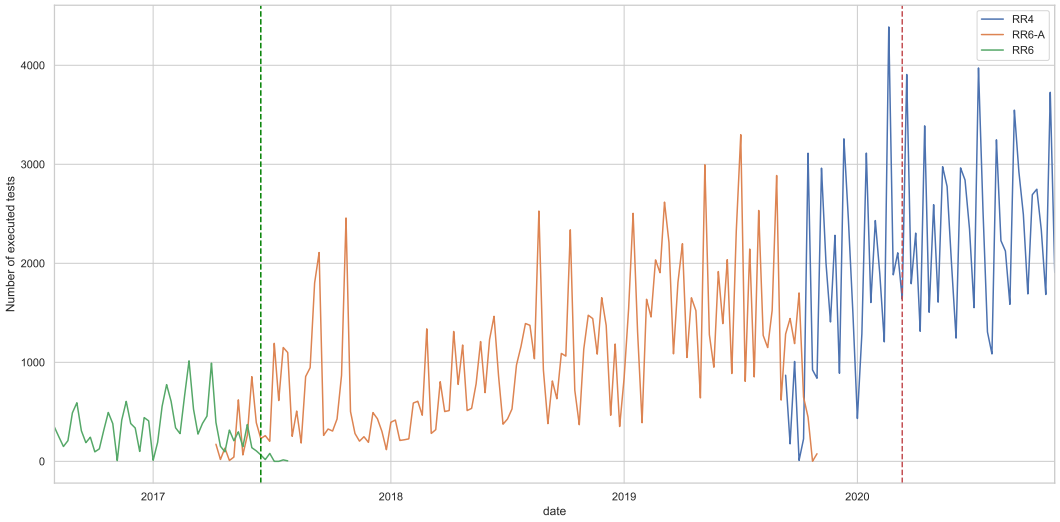


FIGURE 5.13: Number of performed tests over time.

Fig. 5.14 shows the number of tests performed per release. We observe that the number of performed tests increases from release 49 to reach its highest value during the RR6-A cycle, and then it becomes stable during the RR4 release cycle. We normalized the data to get the number of performed tests per day. Fig. 5.15 shows that the RR4 release model executes almost twice as many tests per day (median) compared to RR6-A model i.e., RR4 releases run more tests in a shorter time frame. Also, the releases in RR6-A model execute almost twice as many tests per day (median) compared to RR6 models. We test the following null hypothesis to compare the total number of test performed for each release models:

- H_{01}^2 : *There is no difference between the number of tests performed per day for RR6 and RR6-A releases.*
- H_{02}^2 : *There is no difference between the number of tests performed per day for RR6-A and RR4 releases.*
- H_{03}^2 : *There is no difference between the number of tests performed per day for RR6 and RR4 releases.*

The Wilcoxon rank sum reject H_{01}^2 ($p < 0.01$), H_{02}^2 ($p < 0.001$) and H_{03}^2 ($p < 0.01$). Therefore, the test confirms our observation. The effect size was *large* ($d > 0.8$) in all cases. Based on the effect size sign, we conclude that more tests are performed in shorter release cycles.

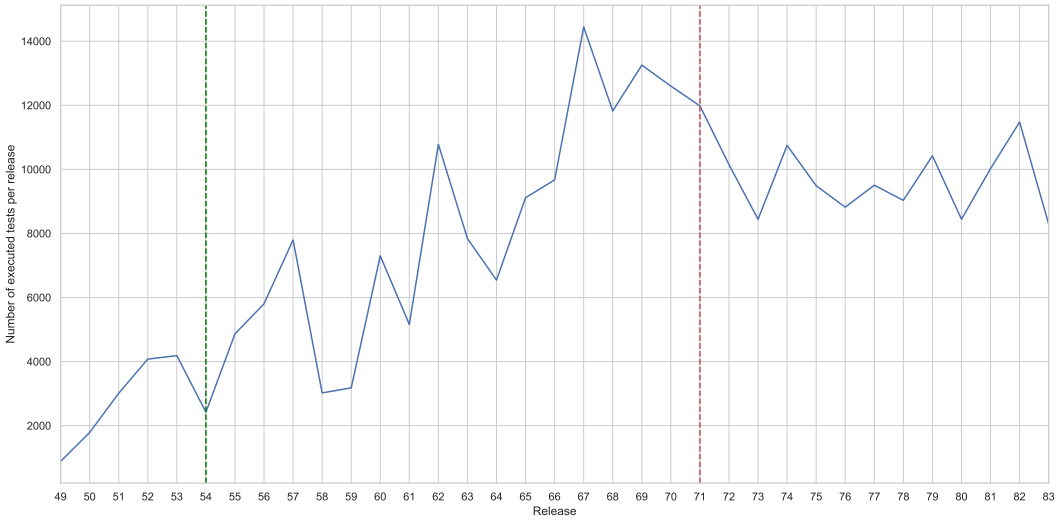


FIGURE 5.14: Number of performed tests per release.

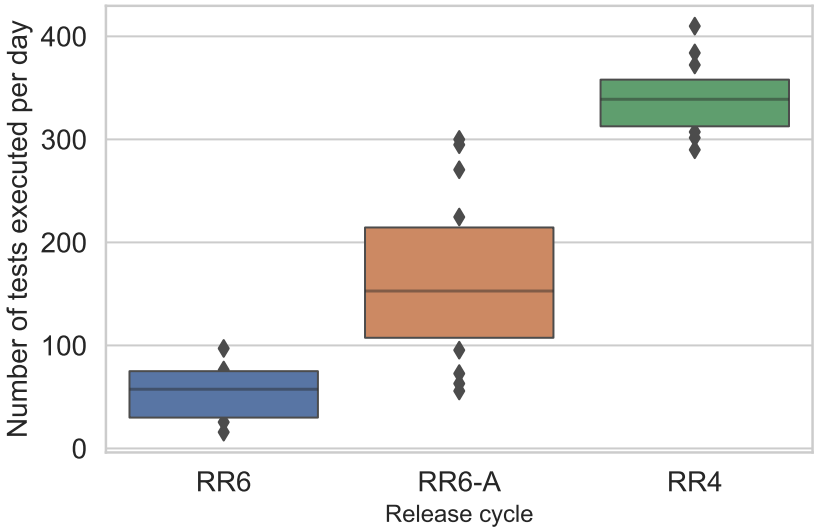


FIGURE 5.15: Boxenplot of the number of performed tests per day.

The number of unique test cases executed captures the functional coverage of the tests per release. Functional coverage is the degree to which the different features in a software version are tested by a test suite. Our only measure of functional coverage is the number of unique test cases executed (the more unique test cases are being executed, the higher the functional coverage). We measure the number of unique test cases executed for each major release; then we normalized it by the length of the release cycle to compare the number of unique test cases per day executed by each release. For this, we test the following null hypotheses to compare the number of test cases for RR6, RR6-A and RR4 releases:

- H_{04}^{2} : *There is no difference between the number of test cases executed per day for RR6 and RR6-A releases.*
- H_{05}^{2} : *There is no difference between the number of test cases executed per day for RR6-A and RR4 releases.*

- H^2_{06} : *There is no difference between the number of test cases executed per day for RR6 and RR4 releases.*

We can reject H^2_{04} ($p < 0.01$), H^2_{05} ($p < 0.001$) and H^2_{06} ($p < 0.01$). The effect size was *large* ($d > 0.9$) in all cases. Based on the effect size sign, we conclude that more unique test cases are executed in shorter release cycles. Thus, the faster release cycle has higher coverage.

Summary: The amount of test executions per day is significantly higher, with higher functional coverage, in the more rapid releases of Mozilla Firefox.

5.3.3 RQ1.3: How does switching to more rapid releases affect the number of testers working on a project?

Given that we observe that more tests are executed during shorter release cycles, we want to investigate if the same number of testers is performing the work or if the rapid releases make it harder to attract testers. Fig. 5.16 reveals that the number of unique testers that performed the test executed for the releases is increasing in RR6 and RR6-A releases before the RR4 releases. Starting from release 66, the number of testers appears to become stable.

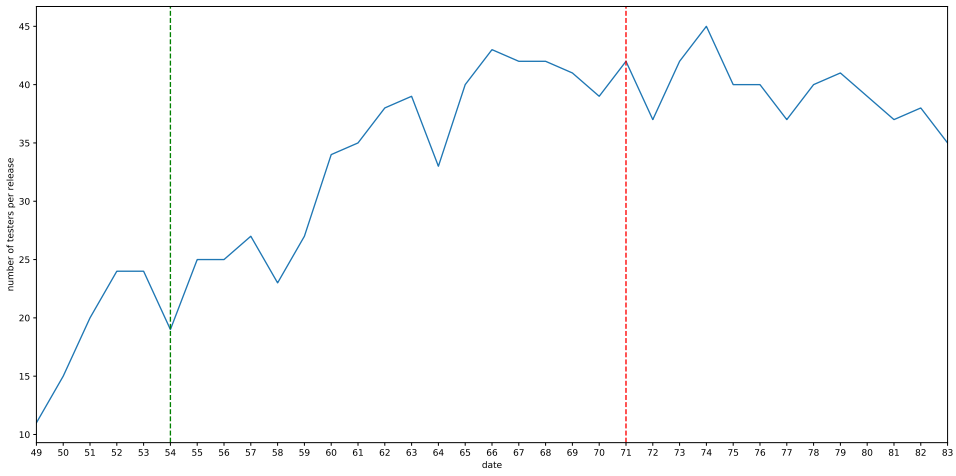
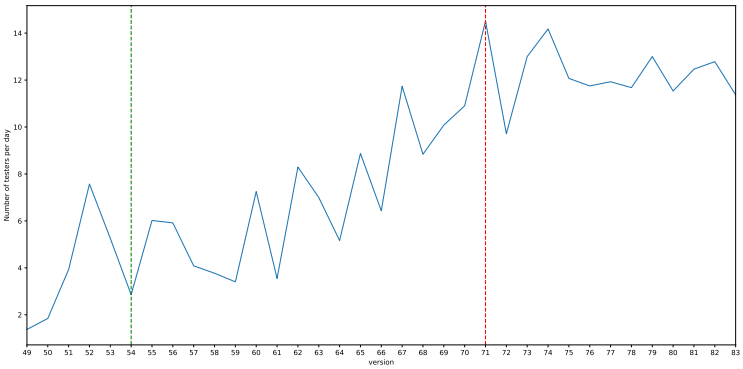


FIGURE 5.16: Number of testers per release.

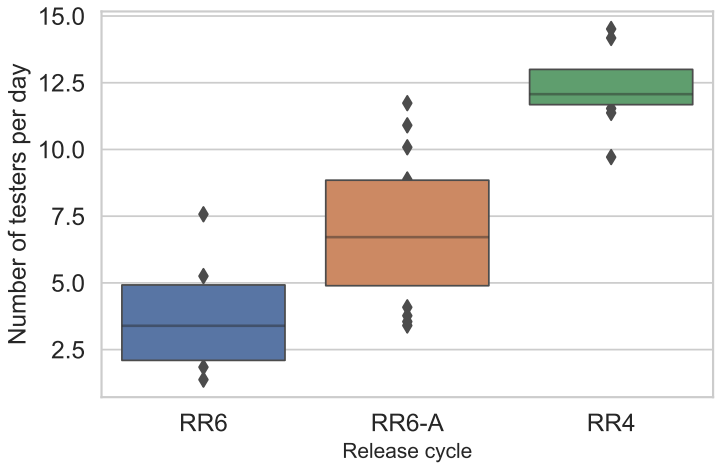
When normalizing the number of testers by the release cycle, we found that the number of testers per day increases over releases of RR6 and RR6-A; and it becomes stable for RR4 (see Fig. 5.17a). Finally, Fig. 5.17b shows the distribution of the number of individuals per day testing the RR6, RR6-A and RR4 releases.
We observe that RR6 releases have a median of 3 testers per day compared to 6.5 testers per day for RR-A releases and 12 testers per day for RR4. We test the following null hypotheses to compare the number of testers for RR6, RR6-A and RR4 releases:

- H^3_{01} : *There is no difference between the number of testers per day for RR6 and RR6-A releases.*
- H^3_{02} : *There is no difference between the number of testers per day for RR6-A and RR4 releases.*
- H^3_{03} : *There is no difference between the number of testers per day for RR6 and RR4 releases.*

The Wilcoxon rank-sum test yields a statistically significant result, i.e., we can reject H_{01}^3 ($p < 0.05$), H_{02}^3 ($p < 0.001$) and H_{03}^3 ($p < 0.01$) with large effect size. Although the number of tests executed is higher in more rapid releases. RQ1.2 has revealed that the testers' workload did not change since the tests are performed by a higher number of testers. These results show that the average workload per individual tester did not increase with the switch to more rapid releases.



(A)



(B)

FIGURE 5.17: Distribution (A) and Boxenplot (B) of the number of testers per day

Summary: After the switch of Mozilla Firefox to more rapid releases, more testers are contributing to the testing activity.

5.3.4 RQ1.4 : How does the frequency of intermittent test failures change after switching to more rapid releases?

Good test writing principles could avoid many intermittent test failures. Mozilla has a problem with intermittent failures that occur at an ever-increasing rate. One possible solution for such frequently failing tests is to disable them, but this means reducing the test coverage. Intermittent failures can be an indicator of testing quality. We measured the frequency of intermittent failures over time and if this changes with a shorter release cycle. Fig. 5.18 shows the number of intermittent failures over time in the *Aurora*, *Beta* and *Release* channel. We observe a temporary increase in

the intermittent failures in the *Beta* channel after the removal of *Aurora*. However, everything goes back to its previous range after 4 months. We did not observe any change in the *Release* channel. The increase in the intermittent failures in *Beta* can be described as a transitional period as they lost a stability channel. Also, this increase of intermittent in *Beta* can be explained by that bugs in *Aurora* have been carried over to *Beta*.

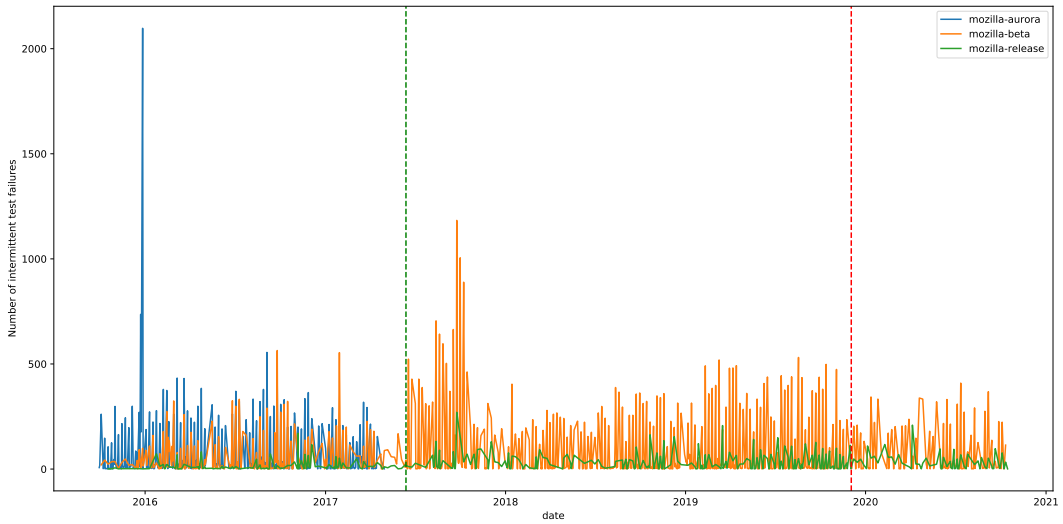


FIGURE 5.18: Number of intermittent test failures per channel.

Summary: After the switch of Mozilla Firefox to more rapid releases, we did not find any permanent change in the intermittent failures.

5.4 Impact of Rapid Releases on Patch Uplifts

5.4.1 RQ2.1: How does the number of accepted and rejected uplifts evolve over time?

To study the patch uplift process, we need to consider a period during which the practice was well established at Mozilla. Castelluccio et al. [22] found that starting from September 2014, uplift practice was well established at Mozilla, so we decided to limit our dataset to only bug reports after September 2014. Between September 2014 and August 2016, we study in total 33,664 issue reports, in which there are 15,946 uplift requests: 1,019 to Release, 9,035 to *Beta*, and 5,892 to *Aurora*. Fig. 5.19 shows the distribution of the number of uplifts in three of Firefox's release channels during this period, and each time point corresponds to a period of one month. We observe a temporary increase in the number of accepted patches after the removal of *Aurora* in the *Beta* channel while the number of accepted patches in the *Release* channel maintains its stability. We find a short impact of the removal of *Aurora* on the number of accepted uplifts on the *Beta* channel. In general, we did not observe any change in the number of accepted patches when moving to a more rapid release mode.
Fig. 5.20 shows the distribution of the number of rejected uplifts in three of Firefox's release channels during this period and each time point corresponds to a period of one month. Fig. 5.20 does not show a clear trend, except that there are fewer patch uplifts rejected after the removal of the *Aurora* in the *Beta* channel.

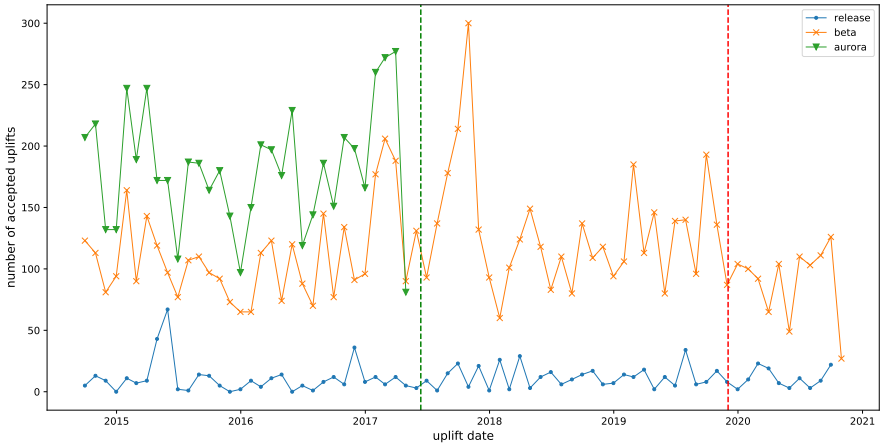


FIGURE 5.19: Number of uplifts during each month on each channel.

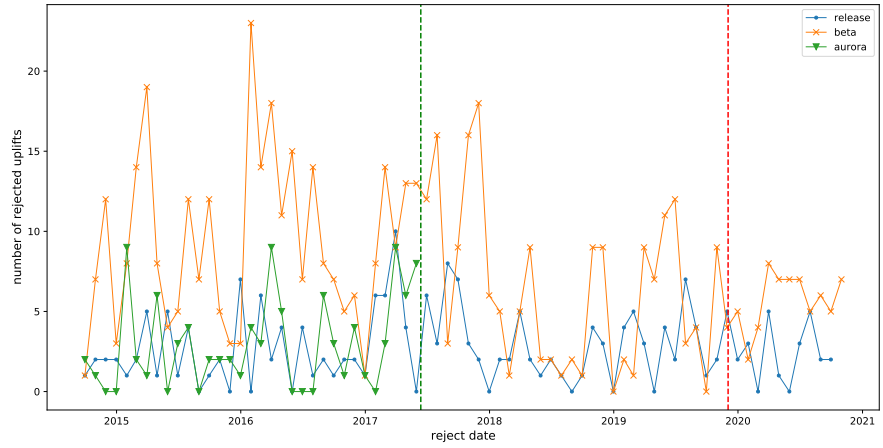


FIGURE 5.20: Number of uplifts rejected over time

Using the metrics from Table 5.1, 5.2 and 5.3, we statistically compare 11 numerical characteristics of patch uplift candidates that was accepted on each channel in the different release models. For each of the 11 metrics m_i, we formulated the following null hypothesis:

> H_i^5 : *There is no difference between the values of* m_i *for patch uplift candidates that were accepted in* p_1 *and* p_2*, where* $i \in \{1, ..., 11\}$ *and* $(p_1, p_2) \in$ *{(RR6, RR6-A),(RR6-A,RR4),(RR6, RR4)}*

Table 5.4 summarizes differences between the characteristics of patches that were accepted for an uplift in different release models. We show the median value of accepted uplifts for each metric, as well as the p-value of the Mann Whitney U test and the effect size. We report the metrics for which there is a statistically significant difference between accepted on different release model patch uplift candidates. We only show the *Beta* and *Release* channels as the *Aurora* channel was removed. We summarize the different results among the channels as follows:

(p_1,p_2)	Channel	Metric	p_1	p_2	p-value	Effect size
(RR6,RR6-A)	*Beta*	Test churn	0	0	**2.42e-03**	negligible
		Prior changes (m_{10})	193.0	297.0	**2.21e-10**	negligible
		# of comments (m_3)	23.0	22.0	**1.90e-04**	negligible
		Developer exp. (m_1)	12.0	17.5	**1.92e-07**	negligible
		Reviewer exp. (m_2)	0.00e+00	2.0	**3.28e-04**	negligible
		Review duration (m_5)	4.93	3.14	**2.97e-11**	negligible
		Comment words (m_4)	2.33	0	**7.39e-08**	negligible
	Release	# of comments (m_3)	35.5	27.0	**0.03**	small
		Developer exp. (m_1)	10.0	19.0	**4.14e-03**	small
		Review duration (m_5)	3.73	1.21	**1.02e-03**	small
(RR6-A, RR4)	*Beta*	Test churn (m_{11})	0	35.0	**2.74e-04**	medium
		Developer exp. (m_1)	17.5	74.0	**0.02**	medium
		Review duration (m_5)	3.14	-1.0	**2.51e-10**	large
		Comment words (m_4)	0	0	**0.04**	small
(RR6,RR4)	*Beta*	Test churn (m_{11})	0	35.0	**2.78e-06**	medium
		Prior changes (m_{10})	193.0	478.0	**0.05**	small
		Developer exp. (m_1)	12.0	74.0	**1.43e-03**	medium
		Review duration (m_5)	4.93	-1.0	**1.18e-13**	large
		Comment words (m_4)	2.33	0	**2.72e-03**	medium

TABLE 5.4: Accepted patch uplift candidates

We observe only a small difference in the developer's experience and participation metrics in terms of developers experience, review duration and the number of comments between RR6 and RR6-A only. Patches in RR4 are submitted by more experienced developers with a shorter review duration.

For *Beta* channel: Accepted patches in RR4 tend to be submitted by more experienced developers than in RR6-A; however, the median review duration in RR4 is lower. Uplifted patches in RR4 tend to have more complex code in terms of test patches and prior changes than in RR6-A. We found a similar difference between these metrics between the accepted patches in RR4 and RR6. Although accepted patches in RR6 and RR-A have significant differences on some other metrics, the magnitude of these differences is negligible.

Contrary to what we expect, patches do not take longer to be accepted after switching to more rapid releases (no statistical difference in the response delta). Also, it seems that after the switch to more rapid releases, the patches that are being accepted are frequently submitted by developers with high experience. The reviewing duration is becoming shorter when moving to more rapid releases.

Summary: After the switch of Mozilla Firefox to more rapid releases, the accepted patches have a shorted review duration and are being submitted by developers with high experience.

5.4.2 RQ2.2 : How effective are the patch uplifts and how does this change with more rapid releases?

As we found in RQ2.1, the review duration of patches is becoming shorter when moving to use rapid releases. We want to check if this impacts the effectiveness of the patch uplift practice by studying if the uplift caused regression(s) later in the code. To do so, we used the SZZ algorithm [121] to identify patches (these patches could be fault-fixing patches or patches related to features or improvements) that introduced faults in the system. First, we used Fischer et al.'s heuristics [40] to map each studied issue to its corresponding patch(es) (i.e., commits). This heuristic uses regular expressions to look for issue IDs in commit messages. Mercurial is a source-code management tool used by Firefox to keep track of changes to the source code. For each fault-related issue in our dataset, we used the following Mercurial command to extract the list of files that were changed to fix the issue:

```
hg log -template {commit},{file_mods},{file_dels}
```

In this step, we only considered modified and deleted lines, since added lines could not have been changed by prior commits. We denote an issue's fault-fixing file by F_{fix}. For each changed file $f_{fix} \in F_{fix}$, we used Mercurial's annotate command as follow to check which prior commits changed the lines that were modified by the fault-fixing commits. The SZZ algorithm assumes that the fault is located in these lines.

```
hg annotate commit^ -r f_fix -c -l -w -b -B
```

We refer to the obtained commits as fault-inducing candidates. Then we checked if the uplift commits are in the fault-inducing candidates to identify uplifted patches that introduced a regression in the system. We studied the evolution of the number of fault-inducing uplifts. Fig. 5.21 shows that the high number of regression inducing patches are the patches requested on *Aurora*. After the removal of *Aurora*, the number of regression is lower in the *Beta* channel, and it fluctuates between 3 and 18 regressions. The number of regressions is less after the 4-weeks switch, and the short duration we study after this switch can explain this. In this question, we did not find any observable difference in the number of regressions over time. This makes sense as in RQ2.1 we did not find any difference in the characteristics of accepted patches in the different release models for the Release channel.

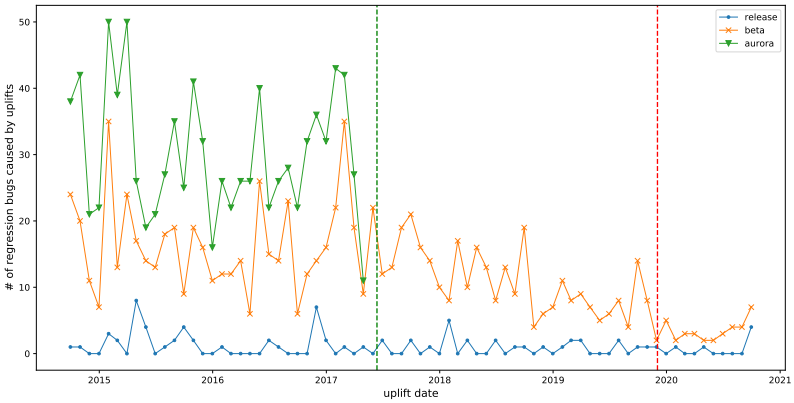


FIGURE 5.21: Number of bugs caused by accepted uplifts on each channel.

Summary: Even though patch requests take less time to review in the more rapid releases, with reviewers of similar experience as in the slower release cycle, the number of regressions caused by patch uplifts decreased slightly.

5.5 Discussion

Khomh et al. [63, 64] studied the effect of the transition to a rapid release cycle on the bug handling activity in Mozilla Firefox. Considering a period of two years of bug activity of the bug reports linked to crashes, they found this transition to lead to shorter bug fixing times. However, on the downside, fewer bugs were being fixed in a timely fashion. Different from their study, we analyzed all types of bugs in the Firefox project. Our results for the more recent Firefox switch to a 4-week release cycle partially align with these findings, as we also observed fewer bugs are fixed, and bugs are fixed faster after the transition.

Mäntylä et al. [69, 80] performed a case study on Firefox and analyzed the results of six years (2006-2012) of test execution runs. They found that the migration to the rapid release model has reduced the community participation in manual testing and made it harder to develop a larger testing community. To maintain with rapid releases, they had to increase the number of specialized testing resources through contractors. However, in our study, we found an increase in the test execution with more individuals contributing to the testing activity after the switch to more rapid releases. Adopters of rapid releases should invest more in test automation [103]. Our analysis did not study the automated test; however, we found an increase in manual testing. One possible hypothesis for this can be explained by the fact that Firefox switched to a new test case management tool. TestRail replaced Moztrap because Moztrap maintenance is slow. The instability of the testing environment can hinder teams. TestRail includes integration with Bugzilla and Github and can automatically file or comment on bugs. It is very detailed and well separates “runs" from “plans". These characteristics might be the reason to attract more testers or execute more tests. Further investigation is required to find if these changes were because of the testing tool. Changes in the software tools might impact the software development activity.

5.6 Threats to Validity

Following the structure recommended by [135], we discuss the threats that may have affected the validity of our findings and how we have tried to mitigate them.
Construct validity concerns the relation between the theory behind the experiment and the observed findings. In our case study, releases have varying lengths; thus, it is not possible to compare metrics in different releases. To enable comparison between releases of varying lengths, we normalized several metrics, such as the number of post-release bugs, by the length of the release cycle.
Threats to *internal validity* concern our selection of subject systems, tools, and analysis method. A first threat concerns the TestRail database because it only represents a part of the Firefox testing process, which is aimed at manual testing of risky regression test cases. Since we did not study the automated test infrastructure, we cannot provide a complete picture of the Firefox testing process.
A second threat stems from the fact that we ignored bugs not having a *Version* field, as well as bugs having a *Version* field that did not correspond to any of the considered releases. We excluded such bugs as it was not possible to deal with such cases automatically.
Threats to *external validity* concern whether the results and conclusions can be generalized outside the scope of this study. Our study is limited to Mozilla Firefox, so we cannot generalize our results. While the followed methodology is applicable to other systems, the obtained findings are not generalizable. The analysis and findings are likely to differ for smaller projects with different versioning, release policies and evolutionary dynamics. Therefore, further studies on different systems are desirable. However, there are few open sources, large and mature systems that switched to rapid releases that can be studied.
Threats to *conclusion validity* concern the relation between the treatment and the outcome. We reported findings based on statistical significance. We paid attention not to violate assumptions of the constructed statistical models.
We mitigate threats to *reliability validity* by providing a publicly available replication package [61] and a detailed description of the followed methodology in Section 5.2.

5.7 Conclusion

Rapid releases are becoming very popular in software projects due to several reasons, such as faster time-to-market, faster bug fixes and frequent user feedback. Keeping the high quality in this fast-paced environment requires a lot of testing before release. In this chapter, we revisited the impact of rapid releases on the software development process in Firefox. We analyze Mozilla Firefox's evolution during the period in which it shifted from a 6-weeks release model to a 4-weeks model to understand potential changes in its bug handling process. First, we analyzed the bug handling process in Firefox. We found that the number of daily reported post-release bugs in the different release models is similar. Bugs are fixed significantly faster in the more rapid release model, however, a smaller proportion of bugs is actually fixed for them.
Second, we studied the effect of switching to more rapid releases on Firefox' system testing. By analyzing data from the TestRail testing tool. We found that more test is executed per day in the more rapid releases with higher functional coverage. Also, more testers are contributing to the testing activity after the switch to more rapid releases. This shows a positive impact of the rapid release on the testing activity.
Third, we studied to which extent the patch uplifting practice is affected by switching to more rapid release cycles as the latter may impact the uplifts' acceptance rate and

effectiveness. We found that release managers tend to accept patches submitted by developers with high experience under more rapid release models. Also, the accepted patches in more rapid releases have a shorter review duration; however, not a different response delta. An investigation should be done to understand this delay in response to accept a patch. Even though patch requests are reviewed for a shorter time in the more rapid releases with reviewers of similar experience compared to the slower one, the number of regressions caused by patch uplifts decreased slightly.

Chapter 6

Conclusion

At the beginning of this dissertation, we presented the research context, research statement and research goals of the dissertation. The main goal of this thesis is to provide a deeper understanding of how rapid releases policies impact different activities of the development process of large and mature open-source software projects. In this chapter, we summarize the contributions of this thesis and sum up the other limitations of our research. Then we suggest extensions to this research. Finally, we present future research opportunities opened by the contributions of this thesis.

6.1 Contributions

> ***Thesis statement:*** By empirically studying rapid releases in large and mature open-source software projects, we provide valuable insights in the advantages and disadvantages of the adoption of rapid releases, and the release management plans and tools that are needed for a successful adoption.

The overarching goal of this thesis is to better understand the impact of rapid releases on the software development process to provide valuable insights in the advantages and disadvantages of the adoption of rapid releases and the release management plans and tools that are needed for successful adoption. To do so, we have studied two large open source projects, Eclipse and Firefox, that adopted rapid releases. To perform our studies, we leverage data recorded in Issue Tracking Systems, Version Control Systems and testing tools; and we survey team members of the Eclipse project.

In Chapter 4, we studied the switch of Eclipse core projects from a yearly release cycle to a quarterly one and its impact on the bug handling process. We carried out a survey with Eclipse maintainers. We found that the transition has allowed the Eclipse Core projects to have a more stable bug handling process as some observed differences in triaging and fixing times and rates before and after annual releases are no longer present for the rapid releases. We apply process mining to analyze the bug report history to discover Eclipse's runtime bug handling process and inconsistencies with the Bugzilla bug handling process. From our analysis, we believe that the transition to a more rapid release cycle has been beneficial to Eclipse in terms of bug handling activity. This was confirmed by feedback from five consulted Eclipse maintainers. The consulted maintainers indicated that rapid releases helped to reduce stress and increase community growth. Moreover, we found that feature freeze periods allow to spend more focused effort fixing bugs for the upcoming releases, especially in the presence of rapid releases, so it needs to be preserved.
As for the disadvantages of rapid releases, consulted maintainers mentioned that beta testing on a release decreased due to fewer milestone releases per cycle and lack of time to contribute larger features. They also found it harder to keep development environments up-to-date and expressed the need to reorder tasks so that more release engineering work can be automated. They highlighted the importance of good management of their platform, as well as the fact that instability and regressions might occur.
According to our survey results, switching to more rapid releases requires careful planning and tracking the transition and being aware of the possible pitfalls. Adopters of rapid releases should test and fix bugs as soon and as frequently as possible as recommended by consulted Eclipse maintainers. They should raise awareness to their developer community, invest in the most appropriate tooling and support processes, and automate as much as possible. Our qualitative analysis highlighted the importance of a good testing plan as part of the preparation for a more rapid release cycle. Adopters of rapid releases should carry out tests all along the release development cycle, not only during the feature freeze period. They should also heavily invest in adequate automated tooling and support processes, especially for continuous integration and deployment.
In Chapter 5, we revisited the impact of rapid releases on the software development process in Mozilla Firefox. Different from previous studies, we analyze the evolution of Firefox during the period in which it shifted from a 6-weeks release model to a 4-weeks model. We found that daily reported post-release bugs activity in the different

release models is similar. Bugs are fixed significantly faster in the more rapid release model, however, a smaller proportion of bugs is actually fixed. Moreover, we found an increase in the testing activity per day in the more rapid releases with higher functional coverage. Also, more testers are contributing to the testing activity after the switch to more rapid releases. This shows a positive impact of the rapid release on the testing activity.
Overall, we found empirical evidence of the impact of the rapid releases policy on different activities in the software development process. Software projects should watch all the aspects of the software development process to identify bottlenecks that prevent them from being more agile. They could use analytics to measure if the targeted improvements have been reached after changing the policy and highlight the unexpected trends to place appropriate mitigations. In addition, researchers need to be aware of all the changes in the release policy process when carrying out empirical studies because their analysis result will be impacted.
To be able to reproduce our analysis in Chapters 4 and 5, we have provided replication packages in [61, 62].

6.2 Limitations

In Chapters 4 and 5, we have presented the limitations related to the empirical analysis we carried out. We can generalize this to the following four main limitations of the research that we have carried out:

- One of the main limitations of this thesis is the potential lack of generalization of the results. Indeed, the empirical studies performed in Chapter 4 and 5 were only focused on Eclipse and Firefox, two large, long lived and mature open source projects. To assess whether our findings can be generalized, there is a need to study other software projects.

- We have applied software analytics to extract insights from a limited set of data sources containing historical information about large-scale OSS development: bug repositories, version control systems, and test suites. There are many other possible sources of development artifacts that we did not consider to support our analysis. These include mailing lists, configuration management systems, code review tools, etc. Analyzing these artifacts can help determine if there have been any other important changes, besides the rapid releases, that may have affected the effectiveness of software development related activities.

- In our analyses, we cannot be sure that the switch to a more rapid release model caused the changes we observed in the non-rapid and rapid release metrics. Since an observational empirical case study is not a controlled experiment, we cannot interpret our statistical findings in isolation. There may have been hidden factors that have caused or influenced the observed differences. For instance, Mäntylä et al. [80] analyzed some potential confounding factors and present a theoretical model explaining the relationship between the release model, release length and test effort. Therefore, additional studies are needed before affirmative conclusions on the effects of a release model on testing effort can be made.

- We restricted our empirical studies to the set of bugs and tests with known specified versions. Data collection challenges that are beyond the scope of this thesis need to be overcome first to allow a complete analysis of the artifacts

of a software development process of the software project. In particular, it is difficult to identify the exact software version that is being targeted by a given bug or test.

- Software tools have played a critical role in the software development process by improving software quality and productivity. The choice of tools and their usage have a strong impact on the productivity of developers and the efficiency of the development process. For instance, the choice of the bug tracker can have an impact on the bug handling process. Both case studies in our case studies used Bugzilla as a bug tracker. Zimmermann et Casanueva [142] studied the impact of switching bug trackers from Bugzilla to GitHub and found that switch to induce an increase in bug reporting, particularly from principal developers themselves, and more generally an increased engagement with the bug tracking platform. This shows that the choice of bug tracker impacts the bug handling process. Thus, our results might not be generalizable to projects using a different bug tracking tool. The impact of important changes in the environment has rarely been studied, whether it is the bug tracking tool or a policy, etc. Therefore, it is important to study the combined impact of all the used tools, policies, and changes over time to understand and improve the evolution of the software development process.

6.3 Possible Research Extensions

In Chapter 4 and Chapter 5, we studied the impact of the rapid release on bug handling activity and testing activity. However, our study did not cover the resources available for each release and its evolution over time. For instance, the number of contributors for each release and even for each product. The latter can impact the productivity of the team and thus the number of resolved bugs, for example. Also, the availability of developers who are most suitable for fixing bugs or performing tests, etc. For instance, if a key developer was on vacation during a release, that will influence the scope that is scheduled for that release. Moreover, we need to study the size of the features planned for a release and the complexity of the features implemented for a release. The size and complexity of the implemented features might impact the different activities in the software development process. Analyzing version control repositories for code churn and vulnerabilities is common in today's software engineering research. Code churn has been used in several prediction models and often appears among the stronger correlations with faults and vulnerabilities [83]. We intuitively expect that the more complex the feature is, the harder it is to test it and thus to find or fix bugs related to this feature. If more features are being implemented in a release, we expect to see more bugs and a higher bug resolution workload, possibly resulting in a longer bug fixing time.

A possible extension to our work is to study how the rapid releases have impacted other components in the ecosystem. In a large complex software system, components and plugins most likely get outdated at a certain point [139] and with the emergence of rapid releases, this might lead to components to tend to be outdated longer and more. For instance, Eclipse core follows the rapid release model; however, plugins do not follow the same regular release policy. It would be useful to study how the adoption of rapid release impacts the plugins developed for Eclipse and if this result in its plugins to become more out of date. Also, it is useful to study how quickly

technical debt will ramp-up when release cycles are so short and can end-users keep up with the faster delivered new releases.

6.4 Future Work

The studies that were performed in the context of this thesis pave the way for several future work possibilities. We outline avenues of future research below:

Expand the case studies. As future work on the selected case studies, we propose to compare the effect of changes in release policy between the Eclipse Core and other Eclipse sub-projects and plugins. One could compare the findings for Eclipse with other competing projects, such as Netbeans, that is of the same nature and share similar functionalities and maturity. Similarly, one could compare Firefox with the open-source Chrome browser. Also, surveying and interviewing members of large and diverse open-source communities would help to get deeper insights on the impact of rapid releases on their activities.

Replication. It would be useful to replicate the empirical studies performed in this thesis on other projects of different scale, type and functionalities etc. For instance, it is important to compare large software projects like Eclipse and Firefox with younger and less mature OSS projects with less established policies and practices. This will hopefully allow for reaching more generalizable conclusions about the impact of rapid releases. Our research was validated by conducting empirical studies on open-source systems. Conducting case studies of commercial systems could help understand the difference in the impact of rapid releases between open and closed source systems. However, the study of the software development process in closed-source systems depends on the available data and repositories. Also, the study requires a high level of understanding of the process and how it is recorded by tools.

Technical aspects of collaborative software development. More research is needed to better understand the impact of changes in the release cycle on other aspects of collaborative software development. For example, how do rapid releases impact the presence of technical debt [128], social debt [125], automated testing and quality, and so on? Moreover, it is necessary to study the technology that was being used by the project. For instance, the presence of automatic bug assignment tools can influence the efficiency of the bug handling activity under the rapid release model. A deeper understanding of the impact of rapid releases on these aspects will allow software organizations and developer communities to make informed decisions on whether and when to switch to a rapid release model safely.

Social aspects of collaborative software development. As a complement to the current research that focused mostly on technical aspects, it would be very useful to conduct action research to investigate the social interaction of software development teams. For example, interesting future research would be to study the impact of rapid release policies on the involvement, productivity and collaboration of developers.
There are challenges in this analysis, such as the unavailability and inaccessibility of data (e.g., privacy issues). Another possible challenge in such an analysis is that the historical data of the project's evolution is distributed over a multitude of data sources, captured by different tools [46]. The same person often uses different accounts in different tools to develop and evolve the ecosystem (e.g., bug trackers, code

reviewing systems, continuous integration tools, version control systems, etc.). There are no strict rules in many open source ecosystems that describe how the account identities must be created [46]. Thus, the merging of identities based on different names or logins and that belong to the same person is a non-trivial process. This process must be carried out to better understand the behaviour of the actual contributors involved in a software project as it can involve many contributors. Moreover, an important issue is preserving privacy as software ecosystem data typically contains information about contributors involved in development and maintenance [85].

Tooling In our work, we extracted information about several aspects of the software development process and the impact of the rapid release policy on it. It would be useful to continue to investigate how such analytics can be used to support better decision making [20]. In particular, we can investigate the ability of predictive analytics to predict future trends and patterns. For instance, we can develop a predictive model of community contributions on the collaborative development tools, recommending the best "next action" for a developer to become more productive in completing tasks such as bug fixing, feature prioritization and testing. This would make the activities more effective with the rapid release policy.

Appendix A

Online Form

Impact of quarterly release cycle on Eclipse Core

This survey is jointly conducted by the University of Lille (France) and the University of Mons (Belgium) with the goal to understand the evolution of the bug handling process in large open source ecosystems such as Eclipse. The principal contact person for this study is Zeinab Abou Khalil, PhD researcher (supervised by Prof. Tom Mens and Prof. Laurence Duchien). She can be reached by email at zeinab.aboukhalil@umons.ac.be
There are 19 questions in this survey. We estimate that it will take up to 20 minutes of your time to complete the survey. Thanks in advance for participating!

A.1 Pre-requisites

A.1.1 By checking this box, you give your explicit consent to allow us to use your responses for research purposes only. All data will be treated anonymously. The survey results will not be used in any way for commercial purposes.*

☐ I agree

A.1.2 Are you/or have you been involved in Eclipse Core?*

○ Yes

○ No

A.2 Demographics

A.2.1 For how many years have you been actively contributing to Eclipse Core?*

○ less than 1 year

○ 1 year

○ 2 years

- ○ 3 years
- ○ 4 years
- ○ 5 year or more

A.2.2 What is/are your current role(s) within Eclipse Core?*

A.2.3 To which Eclipse Core project(s) are you/have been an active contributor?*

- ☐ Platform
- ☐ JDT
- ☐ PDE
- ☐ e4
- ☐ Incubator
- ☐ Equinox

other:

A.3 Transition to quarterly release cycle

Since the 2018-09 release (Eclipse 4.9), the cadence changed from one annual main release plus 3 update/service releases to a quarterly (13-week) release cycle.

A.3.1 How beneficial did you consider the transition to a quarterly release cycle?*

- ○ Extremely beneficial
- ○ Very beneficial
- ○ Somewhat beneficial
- ○ Not so beneficial
- ○ Not at all beneficial

A.3.2 What are the most important advantages you experienced after the transition to a quarterly release cycle?*

A.3.3 What are the most important disadvantages you experienced after the transition to a quarterly release cycle?*

A.3.4 Did you face any difficulties during Eclipse's transition from a yearly to a quarterly release cycle? If yes, which one(s)?*

A.3.5 How did the Eclipse project(s) you were involved in prepare for the transition?*

A.4 Please provide your opinion on the following statements for the *annual* Eclipse releases.

(i.e., until Eclipse 4.8)

	Strongly Agree	Agree	Neither agree nor disagree	Disagree	Strongly disgree	No answer
Bugs are prioritized according to their severity	◯	◯	◯	◯	◯	◯
Pre-release bugs are prioritized over the post-release bugs	◯	◯	◯	◯	◯	◯

A.5 Please provide your opinion on the following statements for the *quarterly* Eclipse releases.

(i.e., starting from Eclipse 4.9)

	Strongly Agree	Agree	Neither agree nor disagree	Disagree	Strongly disgree	No answer
Bugs are prioritized according to their severity	◯	◯	◯	◯	◯	◯
Pre-release bugs are prioritized over the post-release bugs	◯	◯	◯	◯	◯	◯
Bugs are fixed faster than in the annual releases	◯	◯	◯	◯	◯	◯

A.6 Based on your experience

A.6.1 During the bug fixing process, there are typically two ways to resolve a bug: (1) A reported bug is explicitly assigned to someone before it gets resolved; (2) A reported bug gets resolved without explicit assignment to someone. Which of these ways are used in the Eclipse project(s) you are involved in and why?*

A.6.2 Do you think that 13 weeks is an adequate duration between releases, or should it be adapted? Why?*

A.6.3 Do you think the community needs more contributors to the bug fixing process because of the transition to quarterly releases?*

- ○ Yes
- ○ No

Make a comment on your choice here:

A.6.4 After the transition to quarterly releases, do you think there is more pressure on developers in bug fixing?*

- ○ Yes
- ○ No

Make a comment on your choice here:

A.6.5 Besides the transition to a more rapid release cycle, have there been any other important changes in the Eclipse release process and associated tooling that may have affected the effectiveness and effectivity of software development related activities? If yes, which ones and what was their intended goal?

A.6.6 In general, do you think it is beneficial for software projects to adopt a rapid release cycle? For which reasons?*

A.6.7 According to your personal opinion, is it preferable to have release cycles with fixed or with variable time intervals? Why?*

A.6.8 Do you have any advice for the new/future adopters of rapid releases?

A.6.9 Please enter your email if you would like to be informed about the results of our survey. Your information will be treated anonymously and will only be used for this purpose.

Bibliography

[1] Odd Aalen, Ornulf Borgan, and Hakon Gjessing. *Survival and event history analysis: a process point of view*. Springer Science & Business Media, 2008. DOI: 10.1007/978-0-387-68560-1.

[2] Zeinab Abou Khalil et al. "A longitudinal analysis of bug handling across Eclipse releases". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2019, pp. 1–12. DOI: 10.1109/ICSME.2019.00010.

[3] Zeinab Abou Khalil et al. "On the impact of release policies on bug handling activity: A case study of Eclipse". In: *Journal of Systems and Software* 173 (2021), p. 110882. DOI: 10.1016/j.jss.2020.110882.

[4] Bram Adams and Shane McIntosh. "Modern release engineering in a nutshell–why researchers should care". In: *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*. Vol. 5. IEEE. 2016, pp. 78–90. DOI: 10.1109/SANER.2016.108.

[5] Devdatta Akhawe and Adrienne Porter Felt. "Alice in warningland: A large-scale field study of browser security warning effectiveness". In: *22nd USENIX Security Symposium (USENIX Security 13)*. 2013, pp. 257–272.

[6] Wajdi Aljedaani and Yasir Javed. "Bug reports evolution in open source systems". In: *5th International Symposium on Data Mining Applications*. Springer. 2018, pp. 63–73. DOI: 10.1007/978-3-319-78753-4_6.

[7] John Anvik. "Automating bug report assignment". In: (2006), pp. 937–940. DOI: 10.1145/1134285.1134457.

[8] John Anvik, Lyndon Hiew, and Gail C Murphy. "Who should fix this bug?" In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 361–370. DOI: 10.1145/1134285.1134336.

[9] Dimitrios Athanasiou et al. "Test code quality and its relation to issue handling performance". In: *IEEE Transactions on Software Engineering* 40.11 (2014), pp. 1100–1125. DOI: 10.1109/TSE.2014.2342227.

[10] Boris Baldassari. "Mining software engineering data for useful knowledge". PhD thesis. 2014.

[11] Mikaël Barbero. *Simultaneous release brainstorming*. Feb. 2018. URL: https://www.eclipse.org/lists/eclipse.org-planning-council/msg02927.html.

[12] Olga Baysal, Ian Davis, and Michael W Godfrey. "A tale of two browsers". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. 2011, pp. 238–241. DOI: 10.1145/1985441.1985481.

[13] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004. ISBN: 0321278658.

[14] Kent Beck et al. "Manifesto for agile software development". In: (2001).

[15] Becta. *Release Management*. https://www.hci-itil.com/ITIL_v3/docs/fits_release.pdf. 2004.

[16] Andrew Begel and Nachiappan Nagappan. "Usage and perceptions of agile software development in an industrial context: An exploratory study". In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE. 2007, pp. 255–264. DOI: 10.1109/ESEM.2007.12.

[17] J Martin Bland and Douglas G Altman. "The logrank test". In: *Bmj* 328.7447 (2004), p. 1073. DOI: 0.1136/bmj.328.7447.1073.

[18] Jan Bosch, ed. *Continuous Software Engineering*. Springer, 2014. ISBN: 978-3-319-36470-4. DOI: 10.1007/978-3-319-11283-1.

[19] Vid Bregar. *What is the difference between canary, beta, rc and stable releases in android studio?* https://android.jlelse.eu/what-is-the-difference-between-canary-beta-rc-and-stable-releases-in-the-android-studio-bbbb77e7c3cf. 2017.

[20] Raymond PL Buse and Thomas Zimmermann. "Analytics for software development". In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 2010, pp. 77–80.

[21] John Businge, Alexander Serebrenik, and Mark van den Brand. "Survival of Eclipse third-party plug-ins". In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE. 2012, pp. 368–377. DOI: 10.1109/ICSM.2012.6405295.

[22] Marco Castelluccio, Le An, and Foutse Khomh. "An empirical study of patch uplift in rapid release development pipelines". In: *Empirical Software Engineering* 24.5 (2019), pp. 3008–3044. DOI: 10.1007/s10664-018-9665-y.

[23] Marcelo Cataldo et al. "Identification of coordination requirements: implications for the Design of collaboration and awareness tools". In: *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. 2006, pp. 353–362. DOI: 10.1145/1180875.1180929.

[24] Sandy Clark et al. "Moving targets: Security and rapid-release in Firefox". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 1256–1266. DOI: 10.1145/2660267.2660320.

[25] Norman Cliff. "Dominance statistics: Ordinal analyses to answer ordinal questions". In: *Psychological Bulletin* 114.3 (Nov. 1993), pp. 4994–509. DOI: 10.1037/0033-2909.114.3.494.

[26] MDN contributors. *Avoiding intermittent test failures*. https://developer.mozilla.org/en-US/docs/Mozilla/QA/Avoiding_intermittent_oranges. Oct. 2019.

[27] Daniel Alencar da Costa et al. "The impact of rapid release cycles on the integration delay of fixed issues". In: *Empirical Software Engineering* 23.2 (2018), pp. 835–904. DOI: 10.1007/s10664-017-9548-7.

[28] Daniel Alencar da Costa 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: *Proceedings of the 13th International Conference on Mining Software Repositories*. 2016, pp. 374–385. DOI: 10.1145/2901739.2901764.

[29] John w Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. SAGE Publications, Incorporated, 2009.

[30] Marco D'Ambros. "On the evolution of source code and software defects". PhD thesis. Università della Svizzera italiana, 2010.

[31] Alexandre Decan, Tom Mens, and Eleni Constantinou. "On the Impact of Security Vulnerabilities in the Npm Package Dependency Network". In: *International Conference on Mining Software Repositories*. ACM, 2018, pp. 181–191. DOI: 10.1145/3196398.3196401.

[32] Alex Dmitrienko and Gary G Koch. *Analysis of clinical trials using SAS: a practical guide*. SAS Institute, 2017.

[33] Marco D'Ambros et al. "Analysing software repositories to understand software evolution". In: *Software evolution*. Springer, 2008, pp. 37–67.

[34] Steve Easterbrook et al. "Selecting empirical methods for software engineering research". In: *Guide to advanced empirical software engineering*. Springer, 2008, pp. 285–311. DOI: 10.1007/978-1-84800-044-5_11.

[35] Yusuf Ebrahim. "Lesson 1.1.2 Software release life cycle (Lessons in software design architecture and development 1.0: Selby Mvusi and beyond)". In: Jan. 2019, pp. 10 –20.

[36] Moritz Eck et al. "Understanding flaky tests: the developer's perspective". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 830–840. DOI: 10.1145/3338906.3338945.

[37] Eclipse Foundation. *Eclipse Bugzilla repository*. 2019. URL: https://bugs.eclipse.org.

[38] Omar Elazhary et al. "Do as i do, not as i say: Do contribution guidelines match the github contribution process?" In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2019, pp. 286–290. DOI: 10.1109/ICSME.2019.00043.

[39] *Everything You Need to Know to Master Release Management*. https://www.smartsheet.com/release-management-process. 2020.

[40] Michael Fischer, Martin Pinzger, and Harald Gall. "Populating a release history database from version control and bug tracking systems". In: *International Conference on Software Maintenance, 2003 (ICSM)*. IEEE. 2003, pp. 23–32. DOI: 10.1109/ICSM.2003.1235403.

[41] Brian Fitzgerald. "Open source software: Lessons from and for software engineering". In: *Computer* 44.10 (2011), pp. 25–30. DOI: 10.1109/MC.2011.266.

[42] Karl Fogel. *Stabilizing a Release - Chapter 7. Packaging, Releasing, and Daily Development*. https://producingoss.com/en/stabilizing-a-release.html.

[43] Alfonso Fuggetta. "Software process: a roadmap". In: *Proceedings of the Conference on the Future of Software Engineering*. 2000, pp. 25–34.

[44] Emanuel Giger, Martin Pinzger, and Harald Gall. "Predicting the fix time of bugs". In: *International Workshop on Recommendation Systems for Software Engineering*. ACM. 2010, pp. 52–56. DOI: 10.1145/1808920.1808933.

[45] *Glossary*. https://www.lawinsider.com/documents/jAFKJMPI4St.

[46] Mathieu Goeminne. "Understanding the evolution of socio-technical aspects in open source ecosystems". In: *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. IEEE. 2014, pp. 473–476. DOI: 10.1109/CSMR-WCRE.2014.6747221.

[47] Luiz Alberto Ferreira Gomes, Ricardo da Silva Torres, and Mario Lúcio Côrtes. "Bug report severity level prediction in open source software: A survey and research opportunities". In: *Information and software technology* 115 (2019), pp. 58–78. DOI: 10.1016/j.infsof.2019.07.009.

[48] Christopher Guindon. *Core – The Eclipse Foundation*. URL: https://www.eclipse.org/eclipse/platform-core/.

[49] Anita Gupta. "The profile of software changes in reused vs. non-reused industrial software systems". PhD thesis. Norwegian University of Science and Technology, 2009.

[50] Monika Gupta and Ashish Sureka. "Nirikshan: Mining bug report history for discovering process maps, inefficiencies and inconsistencies". In: *Proceedings of the 7th India Software Engineering Conference*. 2014, pp. 1–10. DOI: 10.1145/2590748.2590749.

[51] Israel Herraiz et al. "Impact of installation counts on perceived quality: A case study on debian". In: *2011 18th Working Conference on Reverse Engineering*. IEEE. 2011, pp. 219–228. DOI: 10.1109/WCRE.2011.34.

[52] Melinda R Hess and Jeffrey D Kromrey. "Robust confidence intervals for effect sizes: A comparative study of Cohen'sd and Cliff's delta under non-normality and heterogeneous variances". In: *annual meeting of the American Educational Research Association*. Citeseer. 2004, pp. 1–30.

[53] Heike Hofmann, Hadley Wickham, and Karen Kafadar. "Letter-Value plots: Boxplots for large data". In: *Journal of Computational and Graphical Statistics* 26.3 (2017), pp. 469–477. DOI: 10.1080/10618600.2017.1305277.

[54] Myles Hollander, Douglas A Wolfe, and Eric Chicken. *Nonparametric statistical methods*. Third. Vol. 751. John Wiley & Sons, 2015. DOI: 10.1002/9781119196037.

[55] Pieter Hooimeijer and Westley Weimer. "Modeling bug report quality". In: *International Conference on Automated Software Engineering (ASE)*. ACM. 2007, pp. 34–43. DOI: 10.1145/1321631.1321639.

[56] ISO Iso. "iec/ieee international standard-systems and software engineering–vocabulary". In: *ISO/IEC/IEEE 24765: 2017 (E)* (2017).

[57] Saket Dattatray Joshi and Sridhar Chimalakonda. "RapidRelease-A dataset of projects and issues on Github with rapid releases". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 587–591. DOI: 10.1109/MSR.2019.00088.

[58] Edward L Kaplan and Paul Meier. "Nonparametric estimation from incomplete observations". In: *Journal of the American statistical association* 53.282 (1958), pp. 457–481.

[59] Mike Kaply. *Why Do Companies Stay on Old Technology?* https://mike.kaply.com/2011/06/23/why-do-companies-stay-on-old-technology/. June 2011.

[60] Noureddine Kerzazi and Foutse Khomh. "Factors impacting rapid releases: an industrial case study". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 2014, pp. 1–8. DOI: 10.1145/2652524.2652589.

[61] Zeinab Abou Khalil. *Revisiting the impact of rapid releases on software development activities [Replication Package]*. Dec. 2020. DOI: 10.5281/zenodo.4323237.

[62] Zeinab Abou Khalil, Eleni Constantinou, and Tom Mens. *On the impact of release policies on bug handing activity: A case study of Eclipse [Replication Package]*. Apr. 2020. DOI: 10.5281/zenodo.3762771.

[63] F. Khomh et al. "Do faster releases improve software quality? An empirical case study of Mozilla Firefox". In: *orking Conf. Mining Software Repositories*. IEEE, 2012, pp. 179–188. DOI: 10.1109/MSR.2012.6224279.

[64] Foutse Khomh et al. "Understanding the impact of rapid releases on software quality". In: *Empirical Software Engineering* 20.2 (2015), pp. 336–373. ISSN: 1573-7616. DOI: 10.1007/s10664-014-9308-x.

[65] Dongsun Kim et al. "Where should we fix this bug? a two-phase recommendation model". In: *IEEE transactions on software Engineering* 39.11 (2013), pp. 1597–1610. DOI: 10.1109/TSE.2013.24.

[66] Dongsun Kim et al. "Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts". In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 430–447. DOI: 10.1109/TSE.2011.20.

[67] Carsten Kolassa, Dirk Riehle, and Michel A Salim. "The empirical commit frequency distribution of open source projects". In: *Proceedings of the 9th International Symposium on Open Collaboration*. 2013, pp. 1–8. DOI: 10.1145/2491055.2491073.

[68] Sue Kong, Julie E Kendall, and Kenneth E Kendall. "The challenge of improving software quality: Developers' beliefs about the contribution of agile practices". In: *AMCIS 2009 Proceedings* (2009), p. 148.

[69] Elvan Kula et al. "Releasing fast and slow: an exploratory case study at ING". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 785–795. DOI: 10.1145/3338906.3338978.

[70] Amit Kumar and Avdhesh Gupta. "Evolution of developer social network and its impact on bug fixing process". In: *India Software Engineering Conference*. ACM. 2013, pp. 63–72. DOI: 10.1145/2442754.2442764.

[71] Antti Lahtela and Marko Jäntti. "Challenges and problems in release management process: A case study". In: *2011 IEEE 2nd International Conference on Software Engineering and Service Science*. IEEE. 2011, pp. 10–13. DOI: 10.1109/ICSESS.2011.5982242.

[72] Ahmed Lamkanfi and Serge Demeyer. "Filtering bug reports for fix-time analysis". In: *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE. 2012, pp. 379–384. DOI: 10.1109/CSMR.2012.47.

[73] Ahmed Lamkanfi, Javier Pérez, and Serge Demeyer. "The eclipse and mozilla defect tracking dataset: a genuine dataset for mining bug information". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 203–206. DOI: 10.1109/MSR.2013.6624028.

[74] Ahmed Lamkanfi et al. "Predicting the severity of a reported bug". In: *Working Conference on Mining Software Repositories*. IEEE. 2010, pp. 1–10. DOI: 10.1109/MSR.2010.5463284.

[75] Meir M Lehman. "On understanding laws, evolution, and conservation in the large-program life cycle". In: *Journal of Systems and Software* 1 (1979), pp. 213–221. DOI: 10.1016/0164-1212(79)90022-0.

[76] Jingyue Li, Nils B Moe, and Tore Dybå. "Transition from a plan-driven process to scrum: a longitudinal case study on software quality". In: *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*. 2010, pp. 1–10. DOI: 10.1145/1852786.1852804.

[77] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. "Power laws in software". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 18.1 (2008), pp. 1–26. DOI: 10.1145/1391984.1391986.

[78] Bart Luijten, Joost Visser, and Andy Zaidman. "Assessment of issue handling efficiency". In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE. 2010, pp. 94–97. DOI: 10.1109/MSR.2010.5463292.

[79] Mika V Mäntylä et al. "On rapid releases and software testing". In: *2013 IEEE International Conference on Software Maintenance*. IEEE. 2013, pp. 20–29. DOI: 10.1109/ICSM.2013.13.

[80] Mika V. Mäntylä et al. "On rapid releases and software testing: a case study and a semi-systematic literature review". In: *Empirical Software Engineering* 20.5 (Oct. 2015), pp. 1384–1425. ISSN: 1573-7616. DOI: 10.1007/s10664-014-9338-4.

[81] Lionel Marks, Ying Zou, and Ahmed E Hassan. "Studying the fix-time for bugs in large open source projects". In: *International Conference on Predictive Models in Software Engineering*. ACM. 2011. DOI: 10.1145/2020390.2020401.

[82] Robert Martignoni. "Global sourcing of software development-a review of tools and services". In: *2009 Fourth IEEE International Conference on Global Software Engineering*. IEEE. 2009, pp. 303–308. DOI: 10.1109/ICGSE.2009.47.

[83] Andrew Meneely et al. "When a patch goes bad: Exploring the properties of vulnerability-contributing commits". In: *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE. 2013, pp. 65–74. DOI: 10.1109/ESEM.2013.19.

[84] T. Mens, J. Fernandez-Ramil, and S. Degrandsart. "The evolution of Eclipse". In: *International Conference on Software Maintenance*. Sept. 2008, pp. 386–395. DOI: 10.1109/ICSM.2008.4658087.

[85] Tom Mens. "An ecosystemic and socio-technical view on software maintenance and evolution". In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 1–8. DOI: 10.1109/ICSME.2016.19.

[86] Tom Mens et al. "Challenges in software evolution". In: *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*. IEEE. 2005, pp. 13–22. DOI: 10.1109/IWPSE.2005.7.

[87] Martin Michlmayr. "Quality improvement in volunteer free and open source software projects: Exploring the Impact of Release Management". PhD thesis. University of Cambridge, Mar. 2007.

[88] Martin Michlmayr. "Quality improvement in volunteer free software projects: Exploring the impact of release management". In: *Proceedings of the First International Conference on Open Source Systems*. 2005, pp. 309–310.

[89] Martin Michlmayr and Brian Fitzgerald. "Time-based release management in free and open source (FOSS) projects". In: *International Journal of Open Source Software and Processes (IJOSSP)* 4.1 (2012), pp. 1–19.

[90] Martin Michlmayr, Brian Fitzgerald, and Klaas-Jan Stol. "Why and how should open source projects adopt time-based releases?" In: *IEEE Software* 32.2 (2015), pp. 55–63. DOI: 10.1109/MS.2015.55.

[91] Martin Monperrus. "Automatic software repair: a bibliography". In: *ACM Computing Surveys (CSUR)* 51.1 (2018), pp. 1–24. DOI: 10.1145/3105906.

[92] Joseph Morris. *Software industry accounting*. John Wiley & Sons, 2001.

[93] Mozilla. https://release.mozilla.org/firefox/release/2017/04/17/dawn-project-faq.

[94] Mozilla.github.io. *Mozilla Firefox Development Specifics*. https://mozilla.github.io/process-releases/draft/development_specifics/.

[95] A. Murgia et al. "On the distribution of bugs in the Eclipse system". In: *IEEE Transactions on Software Engineering* 37.06 (Nov. 2011), pp. 872–877. DOI: 10.1109/TSE.2011.54.

[96] Maleknaz Nayebi, Guenther Ruhe, and Thomas Zimmermann. "Mining treatment-outcome constructs from sequential software engineering data". In: *IEEE Transactions on Software Engineering* (2019). DOI: 10.1109/TSE.2019.2892956.

[97] Malanga Kennedy Ndenga et al. "Performance and cost-effectiveness of change burst metrics in predicting software faults". In: *Knowledge and Information Systems* 60.1 (2019), pp. 275–302. DOI: 10.1007/s10115-018-1241-7.

[98] RV O'Connor. "Human aspects of information technology development". In: *International Journal of Technology, Policy and Management* 8.1 (2008).

[99] Lucas D Panjer. "Predicting Eclipse bug lifetimes". In: *International Workshop on Mining Software Repositories*. IEEE Computer Society. 2007. DOI: 10.1109/MSR.2007.25.

[100] Strategic Planning. "The economic impacts of inadequate infrastructure for software testing". In: *National Institute of Standards and Technology* (2002).

[101] Adam Porter et al. "Techniques and processes for improving the quality and performance of open-source software". In: *Software Process: Improvement and Practice* 11.2 (2006), pp. 163–176. DOI: 10.1002/spip.260.

[102] Tom Preston-Werner. *Semantic Versioning 2.0.0*. https://semver.org/. 2013.

[103] Nikhil Rathod and Anil Surve. "Test orchestration a framework for continuous integration and continuous deployment". In: *2015 international conference on pervasive computing (ICPC)*. IEEE. 2015, pp. 1–5. DOI: 10.1109/PERVASIVE.2015.7087120.

[104] Margaret Rouse. *Chrome Release Channels - The Chromium Projects*. https://www.chromium.org/getting-involved/dev-channel. 2020.

[105] Margaret Rouse. *What Is Software Patch/Fix?* https://searchenterprisedesktop.techtarget.com/definition/patch. 2020.

[106] Walker Royce. *Software project management*. Pearson Education India, 1998.

[107] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical software engineering* 14.2 (2009), p. 131.

[108] Renaud Rwemalika et al. "An industrial study on the differences between pre-release and post-release bugs". In: *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2019, pp. 92–102. DOI: 10.1109/ICSME.2019.00019.

[109] R. K. Saha et al. "Are These Bugs Really "Normal"?" In: *Working Conference on Mining Software Repositories*. 2015, pp. 258–268. DOI: 10.1109/MSR.2015.31.

[110] Ripon K Saha, Sarfraz Khurshid, and Dewayne E Perry. "Understanding the triaging and fixing processes of long lived bugs". In: *Information and software technology* 65 (2015), pp. 114–128. DOI: 10.1016/j.infsof.2015.03.002.

[111] Ripon K Saha et al. "Improving bug localization using structured information retrieval". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2013, pp. 345–355. DOI: 10.1109/ASE.2013.6693093.

[112] Vishal Sahasrabuddhe. *Release Engineering vs. Release Management*. https://devops.com/release-engineering-vs-release-management/. Jan. 2016.

[113] Ioannis Samoladas, Lefteris Angelis, and Ioannis Stamelos. "Survival analysis on the duration of open source projects". In: *Information and Software Technology* 52.9 (2010), pp. 902–922. DOI: 10.1016/j.infsof.2010.05.001.

[114] Benjamin R Saville, Amy H Herring, and Gary G Koch. "A robust method for comparing two treatments in a confirmatory clinical trial via multivariate time-to-event methods that jointly incorporate information from longitudinal and time-to-event data". In: *Statistics in medicine* 29.1 (2010), pp. 75–85. DOI: 10.1002/sim.3740.

[115] Andreas Sewe. *One Year of Automated Error Reporting*. June 2016. URL: https://www.eclipse.org/community/eclipse_newsletter/2016/july/article3.php.

[116] Stephen Shankland. *Rapid-release Firefox meets corporate backlash*. http://cnet.co/ktBsUU. June 2011.

[117] Stephen Shankland. *Rapid-release Firefox meets corporate backlash*. https://www.cnet.com/news/rapid-release-firefox-meets-corporate-backlash/. June 2011.

[118] *Sheriffing*. https://wiki.mozilla.org/Sheriffing. Sept. 2020.

[119] Ramin Shokripour et al. "Why so complicated? simple term filtering and weighting for location-based bug report assignment recommendation". In: *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE. 2013, pp. 2–11. DOI: 10.1109/MSR.2013.6623997.

[120] Forrest Shull, Janice Singer, and Dag IK Sjøberg. *Guide to advanced empirical software engineering*. Springer, 2007.

[121] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?" In: *ACM sigsoft software engineering notes* 30.4 (2005), pp. 1–5. DOI: 10.1145/1082983.1083147.

[122] *Software processes*. http://moodle.autolab.uni-pannon.hu/Mecha_tananyag/szoftverfejlesztesi_folyamatok_angol/ch03.html.

[123] Rodrigo Rocha Gomes Souza. "Inappropriate software changes: Rejection and rework". In: (2015).

[124] *Stockwell*. https://wiki.mozilla.org/Auto-tools/Projects/Stockwell. Feb. 2019.

[125] Damian A Tamburri et al. "Social debt in software engineering: insights from industry". In: *Journal of Internet Services and Applications* 6.1 (2015), pp. 1–17. DOI: 10.1186/s13174-015-0024-6.

[126] *The essential guide to release management*. https://www.smartsheet.com/release-management-process. 2020.

[127] Feifei Tu et al. "Be careful of when: an empirical study on time-related misuse of issue tracking data". In: *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM. 2018, pp. 307–318. DOI: 10.1145/3236024.3236054.

[128] Michele Tufano et al. "When and why your code starts to smell bad". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 403–414.

[129] Wil Van Der Aalst. *Process mining: discovery, conformance and enhancement of business processes*. Vol. 2. Springer, 2011.

[130] Rini Van Solingen et al. "Goal question metric (gqm) approach". In: *Encyclopedia of software engineering* (2002). DOI: 10.1002/0471028959.sof142.

[131] VersionOne. *7th annual state of agile survey*. 2013. URL: https://www.stateofagile.com/#ufh-i-338592786-7th-annual-state-of-agile-report/473508.

[132] Arun Vijayaraghavan. *Software Releases*. https://focustesting.wordpress.com/2010/01/10/software-releases/. Jan. 2010.

[133] Shaowei Wang and David Lo. "Version history, similar report, and structure: Putting them together for improved bug localization". In: *Proceedings of the 22nd International Conference on Program Comprehension*. ACM. 2014, pp. 53–63. DOI: 10.1145/2597008.2597148.

[134] Shaowei Wang, David Lo, and Julia Lawall. "Compositional vector space models for improved bug localization". In: *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE. 2014, pp. 171–180. DOI: 10.1109/ICSME.2014.39.

[135] C. Wohlin et al. *Experimentation in Software Engineering - An Introduction*. Kluwer, 2000. DOI: 10.1007/978-1-4615-4625-2.

[136] Xin Xia et al. "Dual analysis for recommending developers to resolve bugs". In: *Journal of Software: Evolution and Process* 27.3 (2015), pp. 195–220. DOI: 10.1002/smr.1706.

[137] Sami Zahran. *Software process improvement: practical guidelines for business success*. Addison-wesley, 1998.

[138] Andy Zaidman et al. “On how developers test open source software systems”. In: *arXiv preprint arXiv:0705.3616* (2007).

[139] Ahmed Zerouali et al. “A formal framework for measuring technical lag in component repositories—and its application to npm”. In: *Journal of Software: Evolution and Process* 31.8 (2019), e2157. DOI: 10.1002/smr.2157.

[140] Feng Zhang et al. “An empirical study on factors impacting bug fixing time”. In: *Working Conference on Reverse Engineering*. IEEE. 2012, pp. 225–234. DOI: 10.1109/WCRE.2012.32.

[141] Tao Zhang et al. “A literature review of research in bug resolution: Tasks, challenges and future directions”. In: *The Computer Journal* 59.5 (2016), pp. 741–773. DOI: 10.1093/comjnl/bxv114.

[142] Théo Zimmermann and Annalí Casanueva Artís. “Impact of switching bug trackers: a case study on a medium-sized open source project”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, pp. 13–23. DOI: 10.1109/ICSME.2019.00011.

[143] Weiqin Zou et al. “An empirical study of bug fixing rate”. In: *Computer Software and Applications Conference (COMPSAC)*. IEEE. 2015, pp. 254–263. DOI: 10.1109/COMPSAC.2015.57.