Empir Software Eng
DOI 10.1007/s10664-014-9308-x

Understanding the impact of rapid releases on software quality

The case of firefox

Foutse Khomh · Bram Adams · Tejinder Dhaliwal · Ying Zou

Abstract Many software companies are shifting from the traditional multi-month release cycle to shorter release cycles. For example, Google Chrome and Mozilla Firefox release new versions every 6 weeks. These shorter release cycles reduce the users' waiting time for a new release and offer better feedback and marketing opportunities to companies, but it is unclear if the quality of the software product improves as well, since developers and testers are under more pressure. In this paper, we extend our previous empirical study of Mozilla Firefox on the impact of rapid releases on quality assurance with feedback by Mozilla project members. The study compares crash rates, median uptime, and the proportion of pre- and post-release bugs in traditional releases with those in rapid releases, and we also analyze the source code changes made by developers to identify potential changes in the development process. We found that (1) with shorter release cycles, users do not experience significantly more pre- or post-release bugs (percentage-wise) and (2) bugs are fixed faster, yet (3) users experience these bugs earlier during software execution (the program crashes earlier). Increased integration activity and propagation of harder bugs to later versions account for some of these findings. Overall, our case study suggests that a clear release engineering process with thorough automation is one of the major challenges when switching to rapid releases.

Communicated by: Maximillano di Penta and Tao Xie

F. Khomh (✉)
SWAT, Polytechnique Montréal, Québec, Canada
e-mail: foutse.khomh@polymtl.ca

B. Adams
MCIS, Polytechnique Montréal, Québec, Canada
e-mail: bram.adams@polymtl.ca

T. Dhaliwal · Y. Zou
Department of Electrical and Computer Engineering, Queen's University, Kingston, ON, Canada

T. Dhaliwal
e-mail: tejinder.dhaliwal@queensu.ca

Y. Zou
e-mail: ying.zou@queensu.ca

Published online: 31 May 2014

Keywords Software release · Release cycle · Software quality · Testing · Bugs

1 Introduction

In today's fast changing business environment, many software companies are aggressively shortening their release cycles (i.e., the time in between successive releases) to speed up the delivery of their latest innovations to customers (HP 2011). Instead of typically working 18 months on a new release containing hundreds of new features and bug fixes, companies reduce this period to, say, 3 months by limiting the scope of the release to the new features and fixing only the most crucial bugs. For example, with a rapid release model (ie., a development model with a shorter release cycle), Mozilla could release over 1,000 improvements and performance enhancements with Firefox 5.0 in approximately 3 months (Mozilla 2011). Under the traditional release model (i.e., a development model with a long release cycle), Firefox users used to wait for a year to get some major improvements or new features.

The concept of rapid release cycle was introduced by agile methodologies like XP (Beck and Andres 2004), which claim that shorter release cycles offer various benefits to both companies and end users. Companies get faster feedback about new features and bug fixes, and releases become slightly easier to plan (short-term vs. long-term planning). Developers are not rushed to complete features because of an approaching release date, and can focus on quality assurance every 6 weeks instead of every couple of months. Furthermore, the higher number of releases provides more marketing opportunities for the companies. Customers benefit as well, since they have faster access to new features, bug fixes and security updates. On the downside, enterprise customers no longer have enough time to stabilize their platforms (Shankland 2011) and customer support costs are increasing because of the frequent upgrades (Kaply 2011). To address these issues, organisations like Mozilla and Google have initiated parallel versions of their products for companies, which are released at a slower schedule (Shankland 2011; Vaughan-Nichols 2012).

Of all these claims, one of the most contested ones is the relation between rapid release models and software quality. Even though proponents claim that a shorter release cycle improves the quality of the released software, this claim has not yet been empirically validated. Baysal et al. (2011) found that bugs were fixed faster (although not statistically significantly) in versions of Firefox using a traditional release model than in Chrome, which uses a rapid release model. Porter et al. reported that shorter release cycles make it impossible to test all possible configurations of a released product (Porter et al. 2006). Furthermore, anecdotal evidence suggests that shorter release cycles do not allow enough time to triage bugs from previous versions, and hence hurt the developers' chances of catching persistent bugs (Downer 2011). Those sources attribute Firefox's current high number of unconfirmed bugs to the adoption of the 6 week-release cycle. In August 2011, Firefox had about 2, 600 bugs that had not been touched since the release of Firefox 4 five months earlier. The number of Firefox bugs that were touched, but not triaged or worked on was even higher and continues to grow everyday (Downer 2011).

To understand whether and how transitioning to a rapid release model can affect the quality of a software system, we empirically analyze the historical bug fixing and field usage (i.e., post-release Li et al. 2012) data of Mozilla Firefox. Firefox is a hugely popular web browser that has shifted from the traditional development model to a rapid release model. This allows us to compare the field quality of traditional releases (in terms of crash reports) to that of rapid releases, as well as the effectiveness of the fixing process of bugs linked to the reported crashes, all in the relatively controlled setting of one organization. Field quality

provides a user's perspective on software quality, whereas the bug fixing process provides a developer's perspective. We also analyze whether the development process (in particular coding activity) changed during the switch to rapid release, partly as a sanity check of the study, and partly to understand possible consequences of rapid releases.

We studied the following three research questions:

RQ1) Is there a correlation between the release cycle model and the observed field quality?

Opponents of the rapid release model are concerned that it does not provide enough time for quality assurance, possibly leaking major bugs to the end user that crash their software. To analyze this phenomenon, we study crash reports automatically gathered through Mozilla's Socorro crash report server, as well as post-release bug reports linked to crash reports.

We found that there is only a negligible difference in the number of post-release bugs when we control for the time interval between subsequent release dates. However, versions developed in short release cycles crash significantly faster at run-time. One possible explanation is that, although short release cycles leave less time for introducing bugs, they involve more integration of changes across branches.

RQ2) Is there a correlation between the release cycle model and the fixing of pre- and post-release bugs?

A second concern about rapid releases is that developers have less time to fix bugs, either bugs identified during development as well as bugs identified by the end user and linked to crash reports (see RQ1). This would mean that bugs would persist longer than they would for traditional release models. To analyze this, we study the characteristics of bug fixing as documented in pre- and post-release bug reports.

We found that bugs are fixed significantly faster for versions developed in a rapid release model. However, proportionally less bugs are being fixed, as harder bugs are being propagated to later releases.

RQ3) Is there a correlation between the release cycle model and coding activity?

During RQ1 and RQ2, we tacitly assumed that the changes observed in field quality and bug fixing were correlated to the changes in release cycle model. However, since the development process of software projects evolves in many other ways as well, there could be confounding factors that we are ignoring. One such factor is the coding activity, i.e., the way in which actual code for new features or bug fixes is produced by developers. To analyze this, we studied the version control data of Mozilla, looking both at characteristics of developers as well as of the developed code.

We found that more developers are working on each rapid release version, resulting in less changes per developer, but those changes are more invasive (touching more files). Similar to traditional minor releases, focus has shifted to continuous maintenance, i.e., the majority of changes fix bugs instead of adding new features. However, some of these phenomena are independent of the switch to rapid releases.

Overall, we identify a number of challenges that decision makers in software companies should be aware of to find the right balance between the delivery speed (release cycle) of new features and the quality of their software. Perhaps unsurprisingly, setting up a stable release engineering process, supported by heavy automation, turns out to be a must when switching to rapid release models, whereas there are no major changes to coding activities.

This paper extends our previous work (Khomh et al. 2012) in three ways. First, we contacted employees at Mozilla to discuss our findings and validate different hypotheses that could explain our findings. Second, we have added RQ3, which studies the version control history of Firefox, to verify whether there are confounding factors related to the development process that might explain our findings, as well as to identify any other consequences of the switch for development. Third, our research question about the rate at which new releases are picked up by users has been rendered obsolete, since Firefox (and its competitors) have been moving towards automated updates to new releases. Hence, we do not consider this question here.

The rest of the paper is organized as follows. Section 2 provides some background on Mozilla Firefox. Section 3 describes the design of our study and Section 4 discusses the results. Section 5 discusses rapid releases in general as well as threats to the validity of our study. Section 6 discusses the related literature on release cycles and software quality. Finally, Section 7 concludes the paper and outlines future work.

2 Mozilla Firefox

Firefox is an open source web browser developed by the Mozilla Corporation. It is currently the third most widely used browser, with approximately 25 % usage share worldwide (Ltd. 2012). Figure 1a shows the release dates of major Firefox versions up until Firefox 9 (the last release used in our crash report study). Firefox 1.0 was released in November 2004 and followed a traditional release (TR) model until version 4.0 (March 2011). Afterwards, Firefox adopted a rapid release (RR) model to speed up the delivery of its new features. This was partly done to compete with Google Chrome's rapid release model (Shankland 2011, 2010), which was eroding Firefox's user base. The next subsections discuss the Firefox development and quality control processes.

2.1 Development Process

Before March 2011, Firefox supported multiple releases in parallel, not only the last major release. Every major version of Firefox (such as 3.6) was preceded by alpha and beta versions (not shown in Fig. 1a) and followed by a series of minor versions, each containing bug

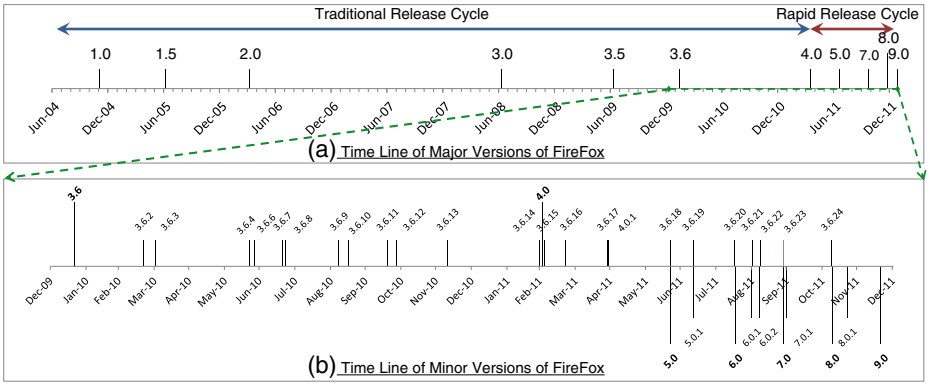


Fig. 1 Timeline of FireFox versions

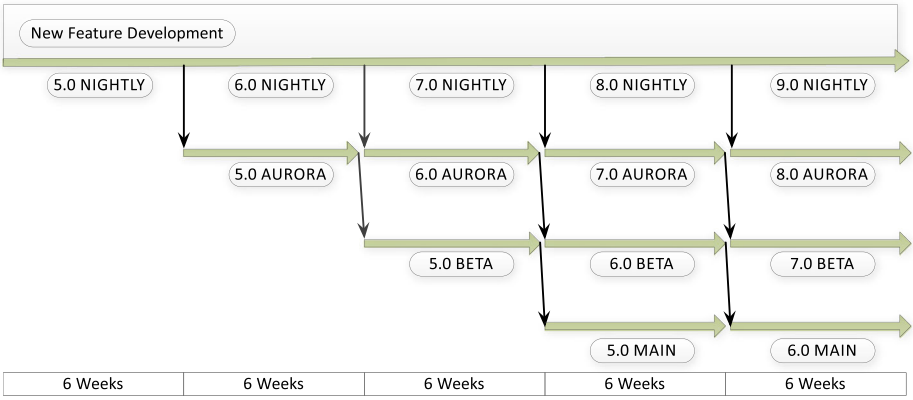


Fig. 2 Development and release process of Mozilla Firefox

fixes or minor updates over the previous version. These minor versions continued even after a new major release was made. Figure 1b shows the release dates of the minor versions of Firefox 3.6. New minor releases of the 3.x series were still being released when Firefox 4 came out, and continued until Firefox 8.

With the advent of shorter release cycles in March 2011, new features needed to be tested and delivered to users faster. To achieve this goal, Firefox changed its whole release process. First, versions are no longer supported in parallel, i.e., a new version supersedes the previous ones. Second, every Firefox version now flows through four release channels: NIGHTLY, AURORA, BETA and MAIN. The versions move from one channel to the next every 6 weeks (Sicore 2011). Major version 5.0 was the first version to have completed the new rapid release model.

Firefox now basically follows a pipelined development and release process (see Fig. 2). The NIGHTLY channel integrates new features from the developers' source code repositories as soon as the features are ready. The AURORA channel inherits new features from NIGHTLY at regular intervals (i.e., every 6 weeks). The features that need more work are disabled and left for the next import cycle into AURORA. The BETA channel receives only new AURORA features that are scheduled by management for the next Firefox release. Finally, mature BETA features make it into MAIN. When the source code of one release is imported from the NIGHTLY channel into the AURORA channels, the source code of the next release is imported into the NIGHTLY channel. Consequently, four consecutive releases of Firefox migrate through Mozilla's NIGHTLY, AURORA, BETA, and MAIN channels at any given time. To address critical security or stability issues, unscheduled releases may be performed independent from the 6-week release schedule.

From this discussion, it becomes clear that the term "rapid release" does not necessarily mean that each release only takes 6 weeks to develop, since each release at least takes three times six weeks to arrive in MAIN (i.e., official public release), and likely much longer if one would incorporate the time used for requirements gathering and mock-ups (discussed below). Instead, "rapid release" refers to the concept of a "release train" where every 6 weeks end users see a new major release and (behind the scenes) the upcoming releases shift one channel up. As such, rapid releases are not a magical new concept, but rather a consistent application of existing concepts to foster user feedback and provide the ability to change the project strategy in an agile manner.

Thus far, we have only mentioned the development and release process. However, what about requirements and design? Mozilla has official teams responsible for activities like identifying and prioritizing user needs and requirements, as well as for mocking up and evaluating designs before the real implementation work in the release channels starts (Rouget 2012). These activities are not tied to a specific release style, but work both for TR and RRs. RRs just provide a smaller, more regular vehicle for delivering new features to the end user, whereas TRs stocked up on many new features before delivering them to the end user. For this reason, this paper does not consider those activities.

2.2 Quality Control Process

One of the main reasons for splitting Firefox' development process into pipelined channels is to enable incremental quality control. First of all, Firefox has an elaborate automated testing infrastructure, which has evolved through a number of incarnations, but currently consists of three test harnesses (rendering, UI and Firefox Desktop) and thousands of automated tests (Mozilla 2013). These tests are enabled on each check-in, and can even be run on-demand, for example before checking in a change. Since we did not have access to this data, analysis of automated tests is outside the scope of this paper.

The second foundation of quality control, is user testing. This consists of a mixture of script-driven manual tests (Litmus and Moztrap frameworks Mäntylä 2013) and free-form beta testing. As changes make their way through the release process, each development channel makes the source code available for such testing to a ten-fold larger group of users. The estimated number of contributors and end users on the channels are respectively 100,000 for NIGHTLY, 1 million for AURORA, 10 million for BETA and 100+ million for a major Firefox version (Paul 2011). NIGHTLY reaches Firefox developers and contributors, while other channels (i.e., AURORA and BETA) recruit external users for testing. The source code on AURORA is tested by web developers who are interested in the latest standards, and by Firefox add-on developers who are willing to experiment with new browser APIs. The BETA channel is tested by Firefox's regular beta testers.

The results of script-driven tests need to be reported using a specific web application, while the free-form beta testing results are reported as bugs in the official bug repository. A bug report contains detailed semantic information about a bug, such as the bug description and bug comments, and metadata such as the bug open date, the last modification date, and the bug status. The bugs are triaged by bug triaging developers and assigned for fixing. Apart from bugs identified during testing, developers also use Bugzilla to submit patches that fix a bug. Once approved, the patch code is integrated into the source code of Firefox on the corresponding channel and migrated through the other channels for release. Bugs that take too long to get fixed and hence miss a scheduled release are picked up by the next release's channel.

The third foundation of quality control are crash reports generated automatically when some user is running Firefox. Each version of Firefox in any channel embeds an automated crash reporting tool, i.e., the Mozilla Crash Reporter, to monitor the quality of Firefox across all four channels. Whenever Firefox crashes on a user's machine, the Mozilla Crash Reporter (Mozilla 2011) collects information about the event and sends a detailed crash report to the Socorro crash report server. Such a crash report includes the stack trace of the failing thread and other information about a user environment, such as the operating system, the version of Firefox, the installation time, the amount of time since system start-up ("up-time") and a list of plug-ins installed.

Socorro groups similar crash reports into crash-types. These crash-types are then ranked by their frequency of occurrence by the Mozilla quality assurance teams. For the top crash-types, testers file bug reports in Bugzilla and link them to the corresponding crash-type in the Socorro server. Multiple bugs can be filed for a single crash-type and multiple crash-types can be associated with the same bug. For each crash-type, the Socorro server provides a crash-type summary, i.e., a list of the crash reports of the crash-type and a set of bug reports that have been filed for the crash-type.

3 Study Design

This section presents the design of our case study, which aims to address three research questions. The first two questions aim to understand the link between rapid release models and respectively (1) the field (i.e., post-release) quality of the software product in terms of major defects noticed by the end user, and (2) how effectively such major bugs are fixed during testing (before release) and maintenance (after release). In other words, the first question considers software quality from the user's perspective, while the second one considers the developers' perspective. The third question on the other hand analyzes the Firefox development process to identify possible confounding factors that could explain our findings for the first two research questions.

RQ1) Is there a correlation between the release cycle model and the observed field quality?

By field quality, we refer to the degree to which a software release is free of major software defects that can be noticed by the end user. Since it is hard to automatically determine whether or not a defect has a major impact on users (the priority or severity fields of bug repositories are known to be unreliable (Khomh et al. 2011)), we approximated this concept through bugs associated with run-time crashes, since those halt the program and hence annoy the user. As such, this question requires analysis of crash reports gathered after ("post") a particular release and associated with a bug report. In other words, for this question we will compare post-release bug reports generated from crash reports between major/minor TR and RR releases, as well as the characteristics of these crashes, in particular how fast the crashes occurred at run-time.

RQ2) Is there a correlation between the release cycle model and the fixing of pre- and post-release bugs?

While RQ1 focused on the occurrence of major defects from the user's perspective, this question focuses on the effectiveness of how such defects are dealt with by the developers. We focus both on defects identified during testing (i.e., before the release) as well as on defects identified by end users (i.e., after the release, see RQ1). For TR releases, testing starts with the first alpha release and ends with the release of a major version. For RR releases, testing for a particular major version happened on the NIGHTLY, AURORA and BETA channel for that release. As mentioned in Section 2, pre- and post-release bugs of TR or RR release X are independent of those of release Y.

RQ3) Is there a correlation between the release cycle model and coding activity?

A major assumption in the previous two RQs is that the development process has not changed during the switch to RRs, in the sense that we assume that we

are comparing the same developers working in the same way on the same project for both TRs and RRs . However, although several elements hint at this assumption, there is no explicit proof. If the assumption would not hold, certain findings could be explained by other factors than the switch to RRs. Since the development process comprises many different activities, many of which are not extensively documented (e.g., requirements or even design), we focus on the actual coding activity, which is heavily tied to development of new features and to bug fixing, and is thoroughly documented in the form of the development history stored in the version control system. We use this data to analyze common code complexity measures on the TR and RR releases and patches, as well as the number of active developers.

We now explain our general approach to collect, process and validate the data. Question-specific metrics and analyses are discussed in the results section of each question, such that the reader does not need to browse back and forth all the time.

3.1 Data Collection

For this study, we obtained crash report data for all versions of Firefox that were released from January 1, 2010 to December 21, 2011, i.e., 25 alpha versions, 25 beta versions, 29 minor versions and 7 major versions that were released within a period of one year before and after the move to the rapid release model. For RQ3, we also added version control data up to release 13, since such data (in contrast to crash report data) can be obtained easily (see below). Firefox 3.6, Firefox 4 and their subsequent minor versions were developed following a traditional release cycle with a mean cycle time of 52 weeks between the major version releases and 4 weeks between the minor version releases. Firefox 5 until 13 and their subsequent minor versions followed a rapid release model with a mean release time interval of 6 weeks between the major releases and 2 weeks between the minor releases. The sole exception is Firefox 10, which was the first extended support release (ESR), i.e., a release meant to be maintained during 9 regular RR releases (with a dedicated ESR channel through which patches are sent). ESR releases are meant to make integration of RR versions in corporate or education settings easier. Table 1 shows additional descriptive statistics of the studied Firefox versions.

3.2 Data Processing

Figure 3 shows an overview of our approach. First, we check the release notes of Firefox and classify the versions based on their release model (i.e., traditional release model and rapid release model). Then, for each version, we extract the necessary data from the crash repository (i.e., Socorro), the bug repository (i.e., Bugzilla), and the source code repository (i.e., Mercurial). Using this data, we compute and plot several metrics, then statistically compare these metrics between the traditional release (TR) model group and the rapid release (RR) model group. The remainder of this section elaborates on each of these steps.

3.2.1 Analyzing the Mozilla Wiki

For each version, we extract the starting date of the development phase and the release date from the release notes on the Mozilla Wiki. The release cycle is the time period between

Table 1 Statistics from the analyzed Firefox versions (the cycle time is given in days)

	Version	Release date	Cycle Time	Alpha Versions (#)	Beta Versions (#)	Minor Versions (#)
Trad.	3.6	21-01-2010	425	3.6a1pre – 3.6b6pre (8)	3.6b1 – 3.6b6 (6)	3.6.2–3.6.24 (22)
	4.0	22-03-2011	91	4.0.b1pre – 4.0.b12pre (12)	4.0.b1beta – 4.0.1beta (14)	4.0.1 (1)
Rapid	5.0	21-06-2011	56	5.0Aurora	5.0Beta	5.0.1 (1)
	6.0	16-08-2011	42	6.0Aurora	6.0Beta	6.0.1, 6.0.2 (2)
	7.0	27-09-2011	42	7.0Aurora	7.0Beta	7.0.1 (1)
	8.0	08-11-2011	42	8.0Aurora	8.0Beta	8.0.1 (1)
	9.0	20-12-2011	42	9.0Aurora	9.0Beta	9.0.1 (1)
	10.0	31-01-2012	42	10.0Aurora	10.0Beta	10.0.1 – 10.0.12 (12)
	11.0	13-03-2012	42	11.0Aurora	11.0Beta	N/A
	12.0	24-04-2012	42	12.0Aurora	12.0Beta	N/A
	13.0	05-06-2012	42	13.0Aurora	13.0Beta	13.0.1 (1)

For the last four releases (below the double line), we only study version control data, while for the other releases we studied both version control and crash report data

the release dates of two consecutive versions. We also compute the development time of the version by calculating the difference between the release date and the starting date of the development phase. As discussed in Section 2.1, the development time is generally longer than the release cycle because the development of a new version is started before the release of the previous one.

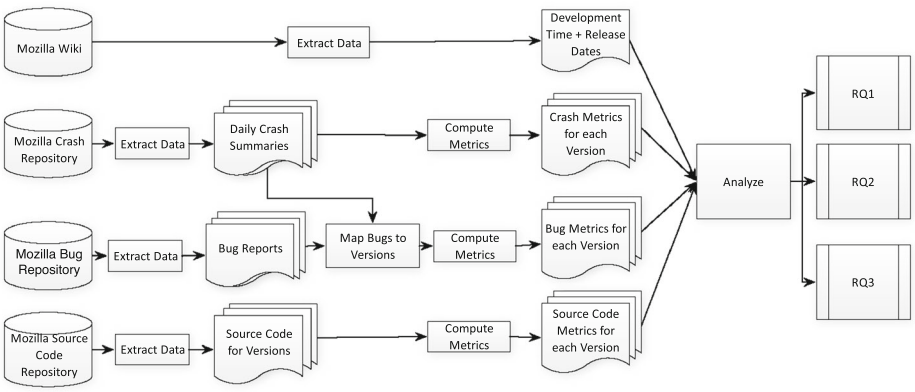


Fig. 3 Overview of our approach to study the impact of release cycle time on software quality

3.2.2 Mining the Mozilla Crash Repository

We downloaded the summaries of crash reports for all versions of Firefox that were released between January 21, 2010 and December 21, 2011. From these summaries, we extracted the date of the crash, the version of Firefox that was running during the crash, the list of related bugs, and the uptime (i.e., the duration in seconds for which Firefox was running before it crashed). To avoid confidentiality leaks, the crash reports do not contain information about the person who experienced a crash or other sensitive data.

Crash report data is not easy to obtain, since it is huge in volume (i.e., 479 GB), requires custom scripts to screen-scrape the data (with the corresponding data format changing regularly), as well as a lot of computation to analyze. For this reason, our crash data is limited to December 2011, whereas the version control data used in RQ3 provides data up until Firefox 13.

3.2.3 Analyzing the Mozilla Bug Repository

We downloaded all Firefox bug reports related to the Firefox crashes in order to determine for each release its post-release bugs associated with crashes. We parse each of the bug reports to extract information about the bug status (e.g., UNCONFIRMED, FIXED), the bug open and modification dates, the priority of the bug and the severity of the bug. We ignored bugs tagged as DUPLICATE, INVALID, INCOMPLETE or UNCONFIRMED, and only focused on FIXED and OPEN bugs. However, we cannot directly identify the version of Firefox for which the bug was filed, since this is not recorded.

Instead, since the analyzed bug reports' recorded crashes are linked to specific versions, we use this indirect mapping to link the bugs to Firefox versions. In particular, for each bug, we check the crash-types for which the bug is filed. Then, we look at the crash reports of the corresponding crash-type(s) to identify the version that produces the crash-type, and we link the bug to that version. When the same crash-type contains crash reports from users on different versions, we consider that the crash-type is generated by the oldest version. At the end of this process, we know for each analyzed post-release bug the oldest version in which the bug occurs.

We can use the obtained post-release data to derive pre-release bugs. In particular, since each major version is preceded by alpha and beta versions (see Section 2.1), we can interpret the post-release bugs of the alpha and beta versions as pre-release bugs of the upcoming major version . In particular, we consider the testing period of a version v_i to be the period between the release date of the first alpha version of v_i and the release date of v_i, such that the post-release bugs of the alpha or beta versions of v_i in this period correspond to the pre-release bugs of v_i. For RR releases, the alpha version corresponds to the AURORA channel and the beta version to the BETA channel.

Since minor versions do not have alpha or beta versions, we cannot calculate pre-release bugs for them, and hence we only analyze pre-release bugs for major versions. Furthermore, none of our research questions analyzes alpha or beta versions by themselves, since otherwise we would be counting their post-release bugs double (once as post-release for themselves, once as pre-release for the upcoming major release). Finally, although the beta version of the next major release (e.g., 6.0 BETA in Fig. 2) is being tested in parallel with regular usage of the preceding major release (e.g., 5.0 MAIN in Fig. 2), the crash reports and hence bug reports of both releases are independent.

3.2.4 Mining the Mozilla Source Code Repository

To compute the number of Total Lines of Code and the Mean Complexity of the source code of each downloaded version for RQ3, we use the source code measurement tool SourceMonitor. SourceMonitor[1] can be applied on $C++$, C, $C\sharp$, $VB.NET$, $Java$, $Delphi$, $VisualBasic(VB6)$, and $HTML$ source code files. Such a polyvalent tool is necessary, given the diverse set of programming languages used by Firefox.

We also parse the version control history of Firefox to extract information about the developers who commit changes, and the number and size of the commits that they have made.

3.3 Data Validation

In order to validate our findings for the three research questions, we contacted 6 Mozilla employees that are involved with Mozilla Firefox, each having a different job profile. We provided them a draft of our paper with the answers to the three research questions, and asked them a couple of background questions, their opinion about our main findings, and their thoughts about possible confounding factors.

In particular, we asked the following background questions:

1. How long have you been working on the Firefox project and what is your role?
2. According to you, what are the two biggest advantages of rapid releases?
3. According to you, what are the two biggest drawbacks of rapid releases?

We then provided our main findings (see next section), and asked the employees' opinion about them. Finally, we also asked the following control questions:

1. Apart from the move towards rapid releases, did the actual Firefox development process change in other ways? Can some of our findings be explained by those changes?
2. What advice would you provide to other organizations planning on moving towards a rapid release model?

We explicitly asked people to not discuss these questions or their answers with other people, to avoid introducing bias. Eventually, we received two replies out of six. Both employees have been working at Mozilla for 8 years, and hence actively experienced the switch to rapid releases. Both have had various roles in release engineering, development and QA. We will refer to both respondents as "QA engineers".

We analyzed the answers of the QA engineers, asked them some clarification questions, then analyzed the replies to those questions. We will discuss the engineers' opinions about our findings at the end of each research question, while the general ideas about rapid releases will be discussed in Section 5.1.

[1] http://www.campwoodsw.com/

4 Case Study Results

This section presents and discusses the results of our three research questions. For each research question, we present the motivation behind the question, the analysis approach and a discussion of our findings.

4.1 RQ1: Is There a Correlation Between the Release Cycle Model and the Observed Field Quality?

4.1.1 Motivation

Despite the benefits of speeding up the delivery of new features to users, shorter release cycles could have a negative impact on the field quality of software systems, since there seems to be less time for testing. Many 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. On the other hand, with fast release trains (e.g., every 6 weeks), developers are less pressured to rush half-baked features into the software repository to meet the deadline, since the scope of a release typically will be smaller. Hence, a rapid release model could actually introduce less bugs compared to traditional release models. Clearing up the interaction between both factors is important to help decision makers in software organizations find the right balance between the speed of delivery of new features and maintaining software quality.

4.1.2 Approach

We measure the quality of a software system using the following three well-known metrics (we use median values to neutralize the impact of outliers):

- **Post-Release Bugs:** the number of bugs linked to a crash report and reported after the release date of a particular minor or major version (lower values are better).
- **Median Daily Crash Count:** the median of the number of crashes per day for a particular minor or major version (lower values are better).
- **Median Uptime:** the median uptime values of all the crashes that are reported for a specific minor or major version (higher values are better).

We answer this research question in three steps. First, we compare the number of post-release bugs linked to crash reports between the traditional release (i.e., TR) and rapid release (i.e., RR) groups. Note that we cannot perform this comparison directly. Herraiz et al. (2011) have shown that the number of reported post-release bugs of a software system is related to the number of deployments (installations), 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 version to control for the usage time.

One way to control for this, is by dividing the number of reported post-release bugs of each minor or major version by the time between its release date and the next major or minor version's release date. For example, if release 3.6.14 is released 50 days after release 3.6.13, we divide the number of post-release bugs of release 3.6.13 by 50. This makes sense, since in earlier work (Khomh et al. 2012), we found that the number of crash reports for an

old version quickly drops once a new version becomes available. We then test the following null hypothesis for major and minor versions:

H_{01}^1: *There is no significant difference between the daily number of post-release bugs of RR versions and TR versions.*

Second, we analyze the distribution of the median daily crash counts for RR and TR major and minor versions, and test the following null hypothesis:

H_{02}^1: *There is no significant difference between the median daily crash count of RR versions and TR versions.*

Third, we compare the median uptime across the post-release bugs of RR and TR major and minor versions. We test the following null hypothesis:

H_{03}^1: *There is no significant difference between the median uptime values of RR versions and TR versions.*

We use the Wilcoxon rank sum test (Hollander and Wolfe 1999) to test H_{01}^1, H_{02}^1, and H_{03}^1. The Wilcoxon rank sum test is a non-parametric statistical test used for assessing whether two independent distributions have equally large values. Non-parametric statistical methods make no assumptions about the distributions of the assessed variables, which is necessary since the values of our metrics are not normally distributed.

4.1.3 Findings

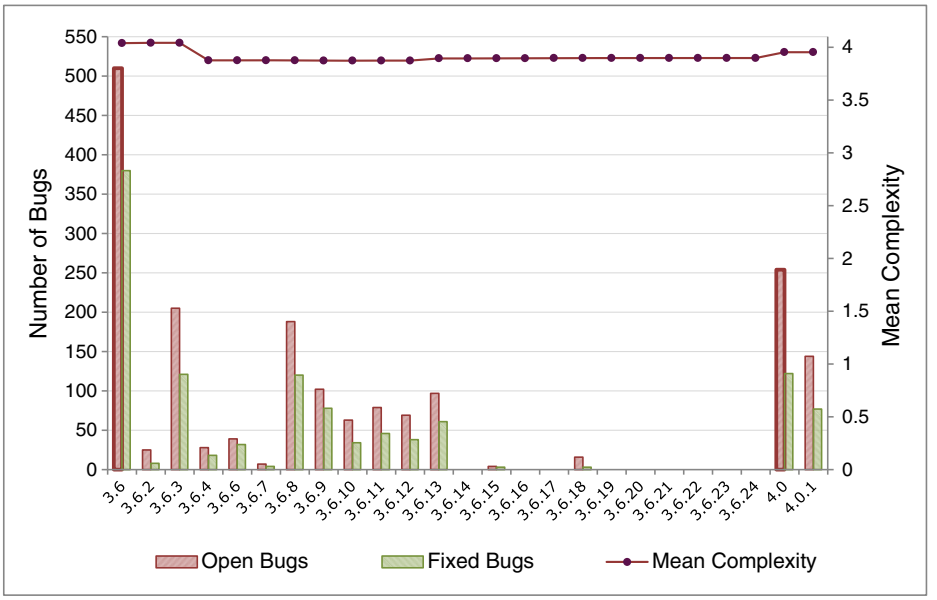
When controlled for the time in between subsequent releases, there is no significant difference between the number of post-release bugs of rapid release and traditional release versions. Figure 4 (left axis) shows the absolute number of post-release bugs linked to crash reports for major/minor TR versions and the RR versions (red-dashed bars). We can see that the last major version of the 3.x series (3.6) has almost twice the number of post-release bugs as versions 4.0, 5.0 and 8.0. In between versions 3.6 and 4.0.1, the maximum number of bugs is between 150 and 200 for versions 3.6.3, 3.6.8, and 4.0.1. The other TR versions typically have similar numbers of post-release bugs as the remaining RR versions.

However, when we normalize by the time in between subsequent releases, Fig. 5a shows that version 8.0 has the highest number of post-release bugs reported per day. Also, the number of post-release bugs reported per day for 3.6.9 and 3.6.11 are the highest of the 3.x series. Overall, the number of post-release bugs of major RR releases is similar to the number of post-release bugs for minor TR releases. Finally, Fig. 5 shows that the distribution of the normalized number of post-release bugs for TR and RR versions is very similar to each other.

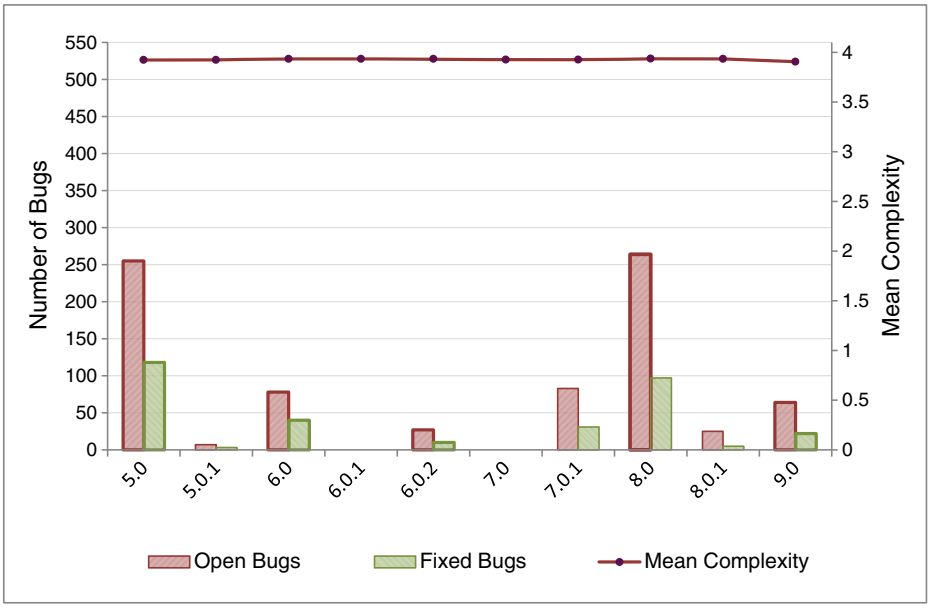
The Wilcoxon rank sum test confirms this observation ($p - value = 0.3$), therefore we cannot reject H_{01}^1.

There is no significant difference between the median daily crash count of rapid release versions and traditional release versions. The Wilcoxon rank sum test yielded a p-value of 0.73. Again, we cannot reject H_{02}^1. Further analysis, however, shows that this test does not tell the whole story.

In reality, the RR versions behave somewhere in the middle between the most crash-prone and the least crash-prone TR versions. Figure 6 shows the distribution (vertical bars) of crash counts for TR and RR versions. The RR versions either have around 125,000 crash reports per day, or around 50,000 reports. For TR releases, the variance is much larger, with

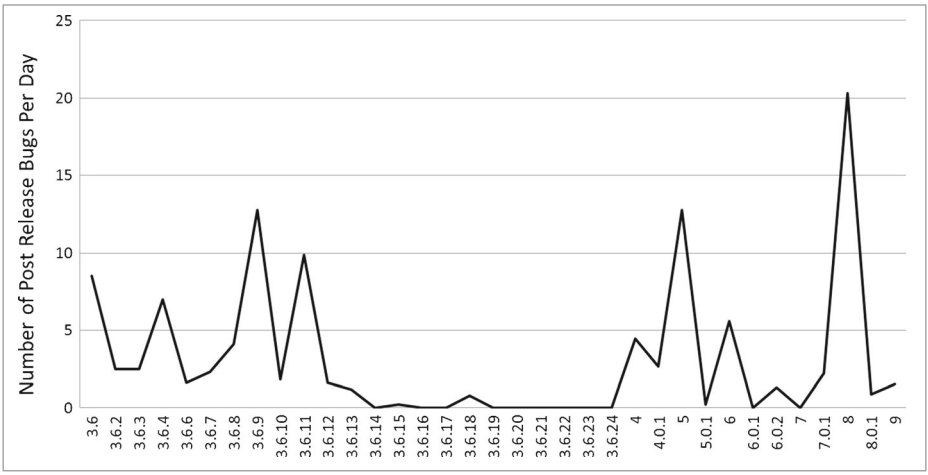


(a)

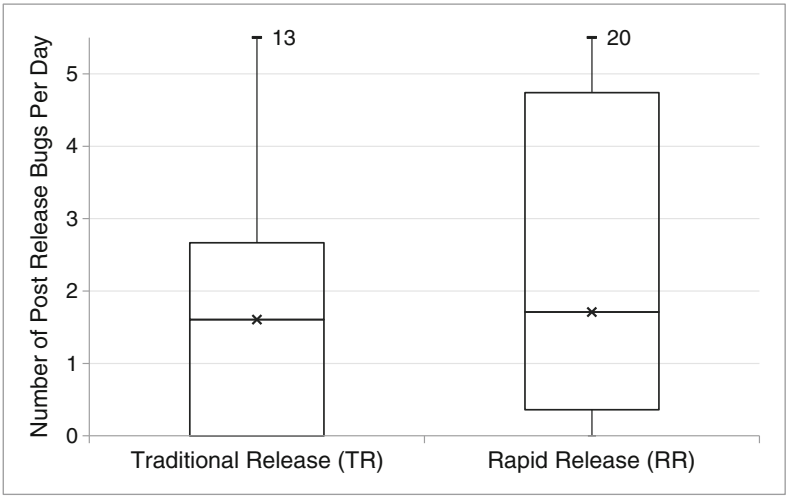


(b)

Fig. 4 Distribution of the number of post-release bugs and mean complexity for (**a**) TR versions and (**b**) RR versions



(a)

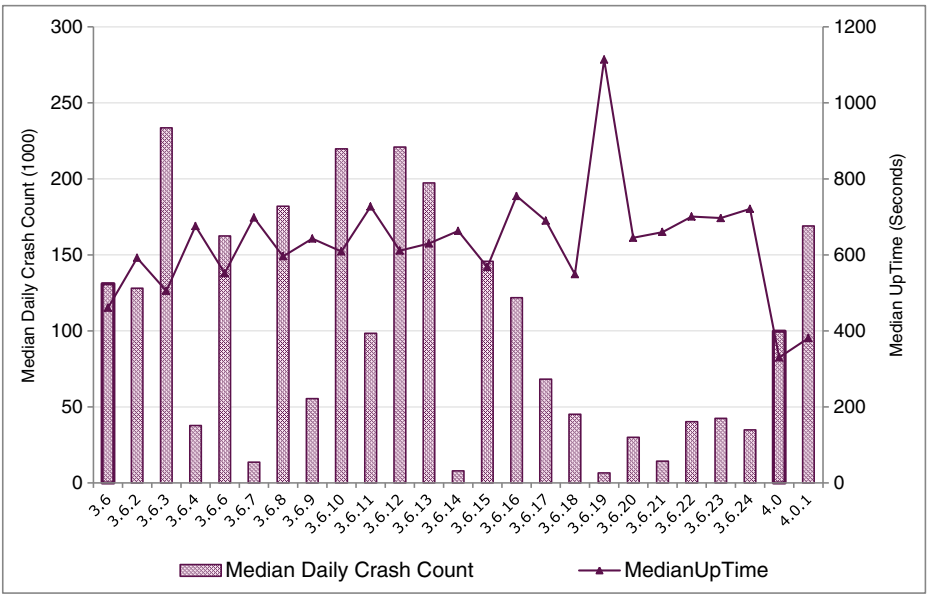


(b)

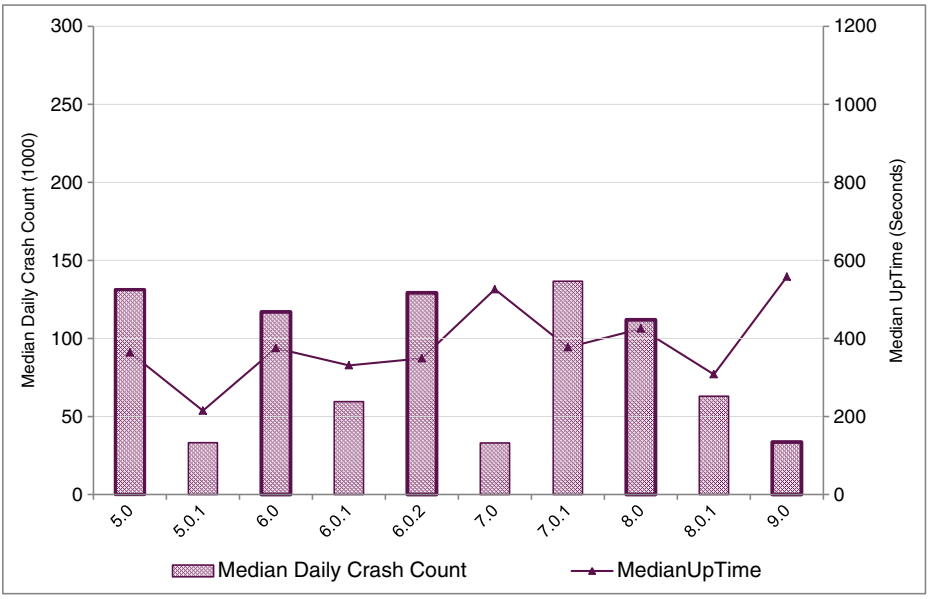
Fig. 5 Distribution (**a**) and Boxplot (**b**) of the number of post-release bugs reported per day

four releases reaching up to 200,000 and four others up to 150,000, while at the other side of the spectrum four releases hardly reach 10,000 reports. Starting from 3.6.18, the number of crash reports drops below 50,000 and remains there until the 4.x series. Since the crash-prone TR and RR versions on average have similar numbers of crash reports, and the least crash-prone TR and RR versions as well, the Wilcoxon test could not reject H_{02}^{1}.

Feedback QA Engineers When asked about the lack of significant difference between the number of post-release bugs/day and the number of crashes/day of TR and RR releases,



(a)



(b)

Fig. 6 Median daily crash counts for (**a**) TR and (**b**) RR versions

the QA engineers explained that on the one hand TR releases took a much longer time and hence accumulated more bugs, while on the other hand there was a lower chance of integration issues, since changes were integrated directly into the main trunk of the version

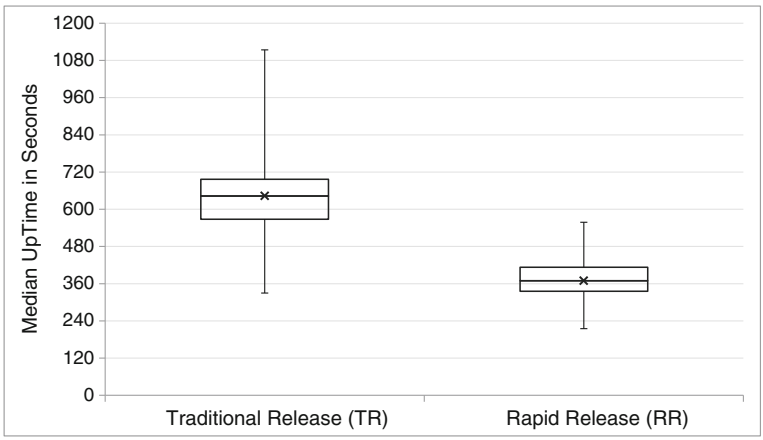


Fig. 7 Boxplot of the median uptime

control system. RR releases, although they accumulate less bugs due to the shorter release cycle, experience a lot more integration *"by having lots of projects branches, each of which runs the same tests as the integration branches. We have about 30 of these project branches active at any one time. That means as a company we can have 30 or more experiments safely running in parallel....I'm guessing that most of the bugs end up being integration issues [of these branches into the main trunk] otherwise the project branch wouldn't have merged back in with mainstream devel."*.

To deal with those integration issues *"For community contributors and developers without access to a project branch, we also introduced the Try Server. The Try Server allows a contributor to submit a speculative patch and have it run the same battery of builds and tests that a normal check-in to an integration branch would trigger. If the results look good, the patch can then be safely landed onto the integration branch. If there are problems, the contributor gets feedback and can try again, all without inconveniencing other developers. The Try Server is generally used for less-invasive patches, or when developers want to target just a single platform or configuration."*

The median uptime is significantly lower for rapid release versions. Figure 7 shows the distribution of the median uptime across TR and RR versions, respectively. We can observe that the median uptime is lower for RR versions. Indeed, if we look at the uptime in more detail (Fig. 6, right axis), we can see that for both TR and RR versions, the median uptime for a version remains constant at 660 and 360 seconds, respectively. Theoretically, TR version 3.6.19 was the best release, since the program ran a median time of 1,100 seconds before a crash would occur, and this release also has one of the lowest numbers of crash reports. As one would expect from the plots, the Wilcoxon rank sum test to decide if the observed difference is statistically significant or not, is able to reject H_{03}^{1} ($p - value$ of $6.11e^{-06}$).

In general, we can conclude that although the median number of daily crash counts and the number of post-release bugs are comparable for RR versions and TR versions, the median uptime of RR versions is lower. In other words, although rapid releases do not seem to impact software quality directly, end users do get crashes earlier during execution $\left(H_{03}^{1}\right)$, i.e., the bugs of RR versions seem to be more annoying for users than the bugs of TR versions.

Feedback QA Engineers One QA engineer noted that the large decrease in median uptime that happened around the release time of Firefox 4, was likely due to some systematic changes in that release, which were unrelated to rapid release and just happened to coincide with it. He said *"we had several big projects after Firefox 4, including reducing memory usage (memshrink), improving perceived performance (snappy), and focusing on crash stability (crashkill). I don't know whether any of those can be directly related to rapid release, or just happened to coincide with it...Firefox 4 was a* ***crisis time*** *for Firefox and Mozilla, since it took so long to complete and was a painful engineering effort. We decided to make our development more agile by switching to rapid release at the same time as we were making a bunch of other changes such as the projects above, and starting Firefox OS development."* The other engineer suspected our findings to be related to the integration issues mentioned earlier. Hence, our findings for median uptime seem to be partly due to rapid releases, and partly due to other factors.

> *Users experience crashes earlier during the execution of rapid release versions, but otherwise there does not seem to be a difference in the number of post-release bugs or the number of crashes.*

4.2 RQ2: Is there a Correlation Between the Release Cycle Model and the Fixing of Pre- and Post-Release Bugs?

4.2.1 Motivation

For **RQ1**, we found, amongst other things, that when one controls for the time in between releases, there is no significant difference between the number of post-release bugs reported per day of traditional release and rapid release versions. 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 pre- and post-release bugs fixed and the speed with which post-release bugs are fixed in the rapid release model compared to the traditional release model. Fixing speed is only analyzed for post-release bugs, because, unlike pre-release bugs, post-release bugs impact regular users of Firefox.

4.2.2 Approach

For each official (i.e., major or minor) version, we compute the following metrics:

- **Fixed Bugs:** the number of pre-release and post-release bugs that are closed with the status field set to FIXED (higher numbers are better).
- **Fix Time:** the duration of the fixing period of a FIXED post-release bug, i.e., the difference between the bug open time and the last modification time (lower value is better)

Similar to the number of reported bugs linked to crash reports in RQ1, the number of closed bugs also depends on the length of the release cycle. Hence, we again need to normalize this metric. Since we are mainly interested in how many post- or pre-release bugs

are fixed relative to all reported post- or pre-release bugs, we use a proportion (i.e., percentage) in this case. Hence, we test the following null hypothesis to compare the efficiency of pre-release bug fixing activities of official traditional and rapid release versions:

H_{01}^2: *There is no significant difference between the proportion of pre-release bugs fixed of a major RR version and the proportion of pre-release bugs fixed of a major official TR version.*

We have an analogous null hypothesis for the post-release bugs of official versions:

H_{02}^2: *There is no significant difference between the proportion of post-release bugs fixed of a official RR version and the proportion of post-release bugs fixed of an official TR version.*

To compute the proportion of pre-release bugs fixed of a major version (minor releases do not have pre-release bugs, as explained in Section 3.2.3), we divided the number of fixed post-release bugs of the version's alpha and beta versions by the total number of post-release bugs reported for the alpha and beta versions. To compute the proportion of post-release bugs fixed of a minor or major version, we used the version's own post-release bugs.

To assess and compare the speed at which post-release bugs of an official version are fixed under traditional and rapid release models, we test the following null hypothesis:

H_{03}^2: *There is no significant difference between the distribution of Fix Time values for post-release bugs related to TR versions and post-release bugs related to RR versions.*

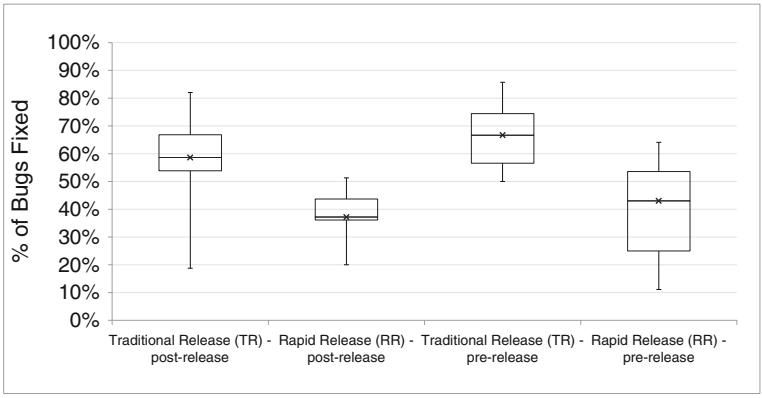
Similar to **RQ1**, hypotheses H_{01}^2, H_{02}^2 and H_{03}^2 are two-tailed. We perform a Wilcoxon rank sum test to accept or refute them.

4.2.3 Findings

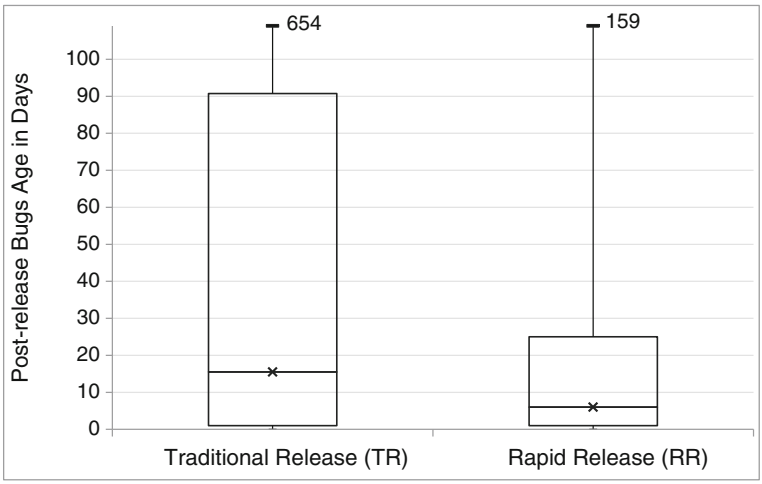
When following a rapid release model, the proportion of pre-release and post-release bugs fixed is lower than the proportion of pre-release and post-release bugs fixed under the traditional release model. Figure 8a shows the distribution of the proportion of (respectively) fixed post-release and pre-release bugs of official versions for TR and RR versions. We can observe that in both cases, the proportion of bugs fixed is higher under the traditional release model. The Wilcoxon rank sum test returned a significant $p - value$ of 0.003 for pre-release bugs. Therefore, we reject H_{01}^2. Similarly, we rejected the corresponding null hypothesis H_{02}^2 for the post-release bugs.

Looking into more detail at the proportion of post-release bugs fixed in each version (green bars in Fig. 4), we can see that almost all TR versions fix at least 50 % of the reported post-release bugs (i.e., Open Bugs in Fig. 4). Conversely, no RR version fixes 50 % of its post-release bugs. This suggests that as TR releases have more time until the next release, and will receive more bug reports, developers also have more time to fix these bugs. RR releases have less time, both for bug reporting and fixing, but seemingly the time for fixing drops faster than the time for bug reporting (more bugs are reported than fixed) (Fig. 9).

Post-release Bugs are fixed faster under a rapid release model. Figure 8b shows the distributions of fix time values for post-release bugs of TR and RR versions, respectively. We can see that developers take almost three times longer to fix post-release bugs under the traditional release cycle. The medians of post-release bug fixing times under traditional release and rapid release models are respectively 16 days and 6 days. The result



(a)

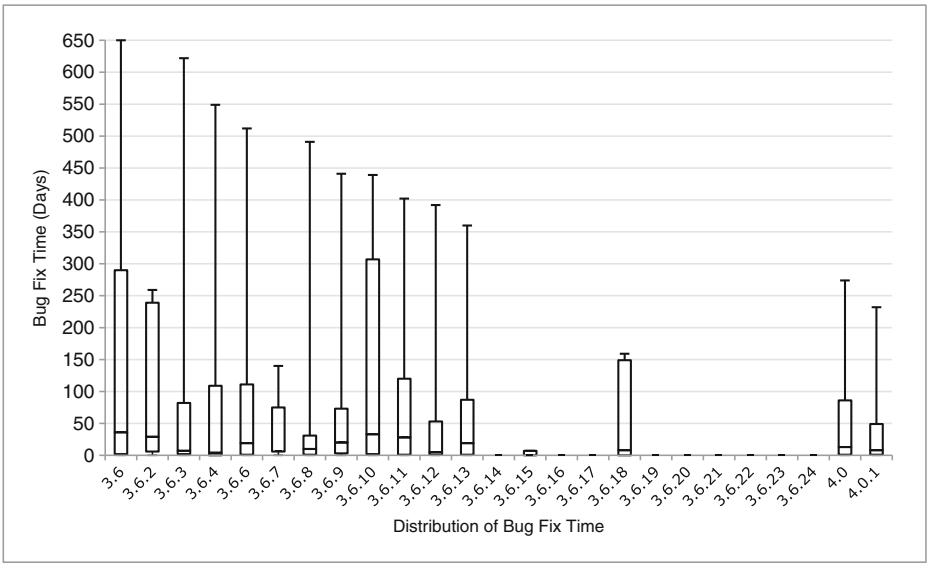


(b)

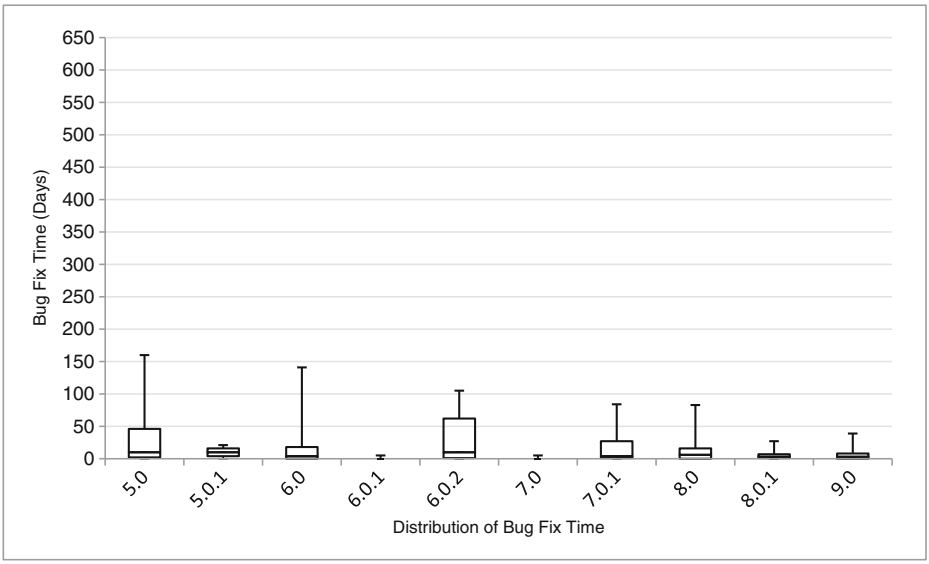
Fig. 8 Boxplots of (**a**) the proportion of pre-/post-release bugs fixed and (**b**) post-release bug fixing time

of the Wilcoxon rank sum test shows that the observed difference is statistically significant ($p - value = 5.22e - 08$). Therefore, we reject H_{03}^2.

Figure 14c shows the distribution of post-release bug fixing times for each TR and RR version. For the TR versions, the fixing time of post-release bugs shows a downward trend, with the bug fixing time falling almost to zero for versions 3.6.14 to 3.6.24 (except for 3.6.18), since those versions hardly had any post-release bugs reported (see Fig. 4). The 75th percentile of bug fixing time of all but one TR version before 3.6.14 is at least 50 days (median of 16 days). When considering the RR versions, two versions have 50 days as 75th percentile of bug fixing time, while all others have a 75th percentile below 25 days (median is even 6 days). For minor RR versions, bug fixing time tends to be lower than for TR versions. This indeed confirms the results of the statistical test.



(a)



(b)

Fig. 9 Distribution of bug fixing time of post-release bugs for (**a**) TR and (**b**) RR versions

In summary, we found that although post-release bugs are fixed faster during a shorter release cycle, a smaller proportion of bugs is actually fixed compared to the traditional release model . We analyzed the reported pre-release bugs, and found that for a rapid release

model, bugs are reported at a slightly higher rate compared to the traditional model, i.e., the project gets more feedback. The mean (respectively median) number of pre-release bugs reported per day for a rapid release model is 10.7 bugs (respectively 1.8 bugs), while the mean (respectively median) number of pre-release bugs reported per day for the traditional release model is 2.6 bugs (respectively 1.6 bugs). Similar to other projects with shorter release cycles (Marschall 2007), Firefox seems to experience a flood of user feedback that, given the limited length of the release cycle, cannot be triaged and fixed in timely fashion.

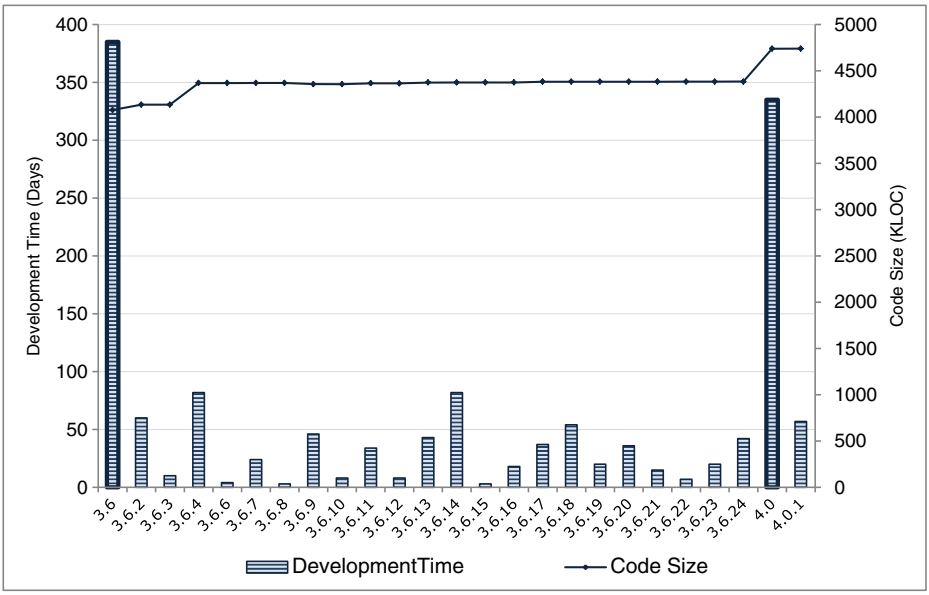
Feedback QA Engineers The QA engineers that we interviewed were not surprised by these findings. One of them stated that: *"I'm not surprised that fewer bugs are fixed: some defects are trivial and some will take weeks of careful code inspection to find and fix. All those bugs would have fallen in the same bucket for a 12+ month release cycle, but now the harder ones get pushed out to later releases"*. Since those releases follow each other in rapid succession, this propagation does not inflate the bug fixing time. The other QA engineer mentioned that these findings *"can be better interpreted that we are being less effective at triaging bugs with rapid release, or that we have more beta users and so the incoming bug rates are larger"*. This resonates with earlier claims about sub-optimal triaging of bugs for RR versions (Downer 2011) where QA people are unable to handle the higher daily traffic of bug reports coming in, leaving many bugs unassigned (nobody working on them) and many duplicate bugs undiscovered, effectively inflating the number of reports. Since our data set did not include un-assigned bug reports, we were unable to verify these claims.

> *The Firefox rapid release model fixes post-release bugs faster than using the traditional model, but fixes proportionally less pre- and post-release bugs, since difficult bugs tend to be postponed to the next release.*

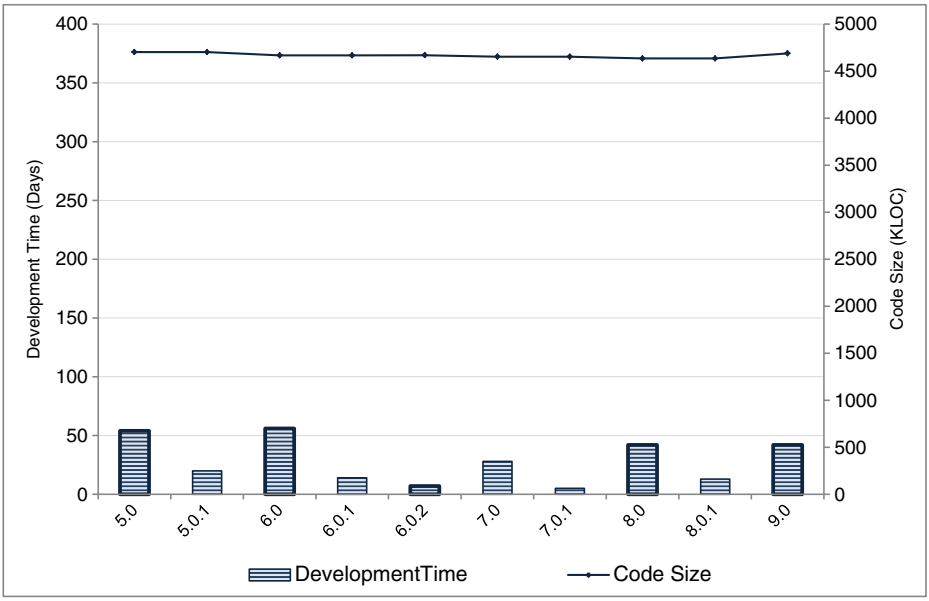
4.3 RQ3: Is there a Correlation Between the Release Cycle Model and Coding Activity?

4.3.1 Motivation

Up until now we have assumed that there were no substantial changes in the Firefox development process other than switching to the rapid release model. However, it is possible that some workload redistributions were necessary to cope with short release cycles, or that other agile techniques have been introduced, for example to determine which bugs should be fixed first. Such a redistribution of work could have large repercussions on software organizations interested in adopting rapid release models. Furthermore, there could have been changes to the development process unrelated to the switch to rapid releases, such as a thorough re-engineering of a major component. Such invasive changes could explain some of our findings. Hence, this question analyzes the development process before and after the switch to rapid releases to identify development process changes related or unrelated to rapid releases. In particular, given its importance and the availability of historical data, we focus on coding activities. In other work, we have already studied changes in system testing (Mäntylä et al. 2013), while other activities of the development process, such as requirements and design, are left as future work.



(a)



(b)

Fig. 10 Development time and code size for (**a**) TR and (**b**) RR versions

4.3.2 Approach

To analyze changes to coding activity, we computed two groups of coding-related metrics: (1) complexity metrics on the source code of the TR and RR major/minor versions, and (2) metrics on the changes made to RR and TR versions, based on the actual patches (code changes) and commit log information extracted from source code repositories.

We computed the following complexity metrics:

- **Total Lines of Code:** the total number of lines of code of all files contained in a version.
- **Mean Complexity:** the mean of the McCabe Cyclomatic Complexity of all files contained in a version. The McCabe Cyclomatic Complexity of a file is the count of the number of linearly independent paths through the source code contained in the file.
- **Development Time:** the duration in days of the development phase of a version (which is not identical to the time in between releases, see Section 2.1). Although not a complexity metric per se, we use this metric to analyze potential correlation between having less time and obtaining higher or lower values for the other metrics.

We also computed the following change metrics:

- **Monthly Active Developers:** the mean number of developers changing the code per month.
- **Developer Contribution:** the mean number of commits authored per developer per month.
- **Code Churn:** the mean number of files modified in a commit per month.
- **Rate of New Code:** the total number of new lines of code added in the version (i.e., the delta of Total Lines of Code of a new version compared to Total Lines of Code of the previous version) divided by the Development Time.

These metrics all measure to some extent the rate of activity of developers, and how much they contribute to the project. We test the following null hypotheses to compare the coding activity under traditional and rapid release models:

H^3_{01}: *The total number of lines of code of a TR version is not significantly different from the total number of lines of code of an RR version.*

H^3_{02}: *The mean McCabe Cyclomatic Complexity of a TR version is not significantly different from the mean McCabe Cyclomatic Complexity of an RR version.*

H^3_{03}: *The mean number of developers per month actively involved in source code editing is not significantly different for TR and RR models.*

H^3_{04}: *The mean number of monthly commits authored per developer is not significantly different for TR and RR models.*

H^3_{05}: *There is no significant difference in the code churn of TR and RR models.*

H^3_{06}: *The mean rate of new code is not significantly different for TR and RR models.*

For each commit, we also record whether it is a bug fixing commit, or a feature enhancement, to analyze whether more or less effort is spent on bug fixing vs. regular development. We used traditional regular expressions that look for bug report identifiers in the commit logs as well as terms like "crash" and "defect" to distinguish between feature enhancement changes and bug fix changes (Śliwerski et al. 2005). We also record the name of the developers responsible for each change. This leads to the following hypothesis:

H^3_{07}: *The proportion of commits that fix bugs for TRs is not significantly different from the corresponding proportion for RRs.*

Similar to **RQ1** and **RQ2**, we perform a Wilcoxon rank sum test to accept or refute these seven hypotheses, which are two-tailed. In addition, we also build multivariate linear regression models (Fox 2008) to analyze the combined correlation of the different analyzed metrics with the median daily number of crash reports, the median uptime, the daily number of post-release bugs and the percentage of post-release bugs fixed. For each dependent variable, we build one model that explains the data of all TR and RR releases together, using the Total Lines of Code, Code Churn and Mean Complexity metrics, as well as the Development Time and Cycle Time as independent variables.

To determine the metrics with the highest impact in these models, we use the concept of relative weights (Johnson 2000). The importance (weight) of a metric in a multivariate linear regression model is determined by building all possible models on a subset of all metrics and evaluating the role of each metric across all models in which it features (we use Johnson et al.'s algorithm for that). The higher the weight of a metric, the more it explains the data. Note that we checked the metrics for multi-collinearity before building the models, but all metrics had a small enough variance inflation factor (VIF) value (below 5), hence we did not have to remove any metric. Also, when adding the developer-related metrics as independent variables, no metric was statistically significant, hence we discarded those models.

4.3.3 Findings

We found no significant difference between the size and code complexity of traditional and rapid releases. The curves in Fig. 10a and b show that the size of the RR versions hardly changes. This is similar to the observations for the size of the minor TR versions, which remained mostly constant. The only exception is minor version 3.6.4, which (together with major version 4.0) showed a sudden increase in code size. Interestingly, this increase was accompanied by a substantial decrease of mean complexity, as is shown in Fig. 4a and 4b. Except for version 4.0, other TR or RR versions did not see any change in mean complexity. These observations seem to be due to an overhaul of Firefox in version 3.6.4, implementing a crash protection feature for "out-of-process plugins" that isolates third party plugins from the browser process (in order to prevent the browser from crashing when a plugin crashes). Version 4.0, as explained in RQ1, featured many changes, including a completely revised user interface. These major changes could explain the drop in complexity observed in Fig. 4a. The Wilcoxon rank sum test could not reject H_{01}^3 or H_{02}^3.

The number of active developers per month and the number of files modified per commit for RRs is significantly higher than for TRs, however the mean number of commits authored per developer per month is lower.

Figure 11a shows a boxplot of the mean number of developers per month that participated actively in the development of Firefox before and after the adoption of the rapid release model. On average, under the traditional release model, 145 developers were involved in the development of Firefox each month. This number increased to 227 after the adoption of the rapid release model. The median numbers of active developers under traditional and rapid release models are 151 and 220, respectively. This difference in the number of monthly active developers over the development history of Firefox is statistically significant with the Wilcoxon rank sum test showing a $p - value = 1.53e - 07$. We therefore reject H_{03}^3.

If we study these numbers more in detail (Fig. 12a), we can see that the number of developers per month has been growing significantly according to a concave curve. The mean number of developers per month has more than doubled compared to release 3 from around 100 to around 250. From versions 9 and 10 on, the number has reached a plateau.

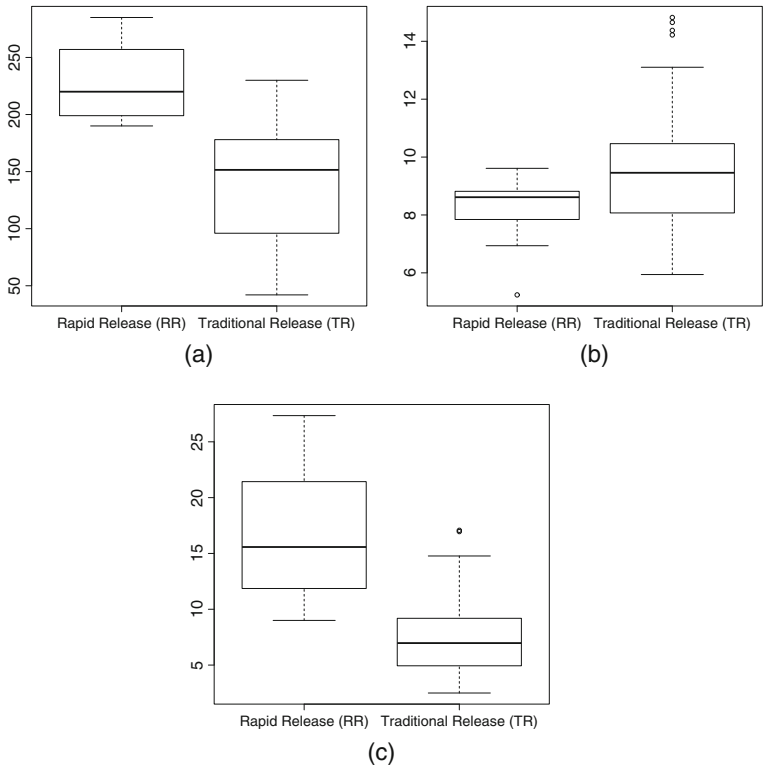


Fig. 11 Boxplots of (**a**) the mean number of active developers per month, (**b**) the mean number of monthly commits per developer and (**c**) the mean number of files modified per commit

On the other hand, the mean number of commits per developer (Fig. 11b) is significantly lower under rapid release with a $p - value = 0.02837$. This means that while (proportionally to the cycle time) more people are participating, most of them perform less changes. Figure 12b shows indeed a decrease of around 11 commits per developer on average to a number between 5 and 6.

The number of files modified per commit (Fig. 11c) is significantly higher under the rapid release model with a $p - value = 3.059e - 09$. This seems at odds with the idea of rapid releases, since the goal is to avoid large, risky changes and instead select small increments to evolve a system. However, a look at the detailed distribution (Fig. 12c) shows that the number of changed files has been fluctuating a lot, from a low value around 5 to as high as 23 (version 6), back to 12 and up to 19 again.

Overall, the amount of new code in major RR versions is similar to the amount of new code in minor TR versions. Since we found earlier that the development time of major RR versions is similar to the development time of minor TR versions, this finding is not that surprising. On the other hand, it also means that we cannot explain our findings regarding bug fixing in RQ2 through changes in code characteristics.

RRs are under continuous maintenance, i.e., bug fixes amount to more than 90 % of all code changes. Figure 13a and b show the mean number and percentage of commits per

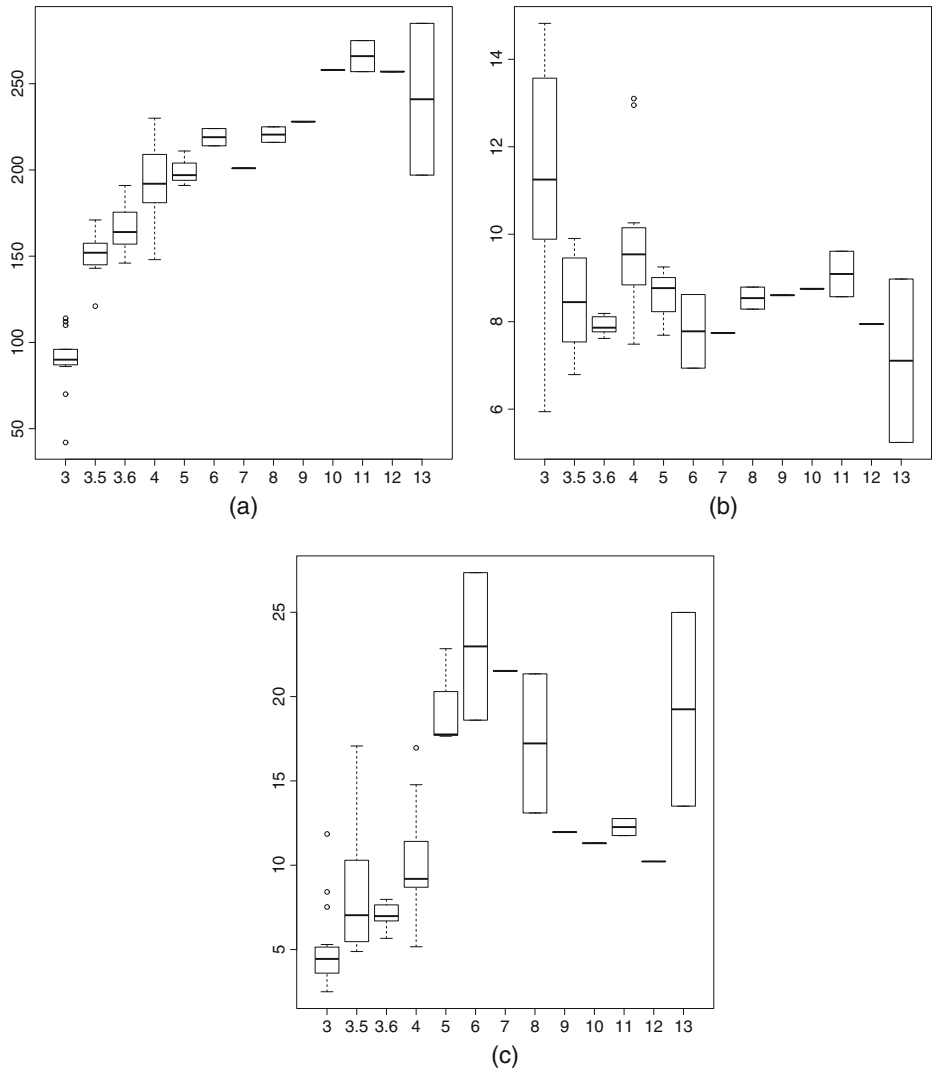


Fig. 12 Distribution of (**a**) the number of active developers per month, (**b**) the number of monthly commits per developer and (**c**) the number of files modified per commit monthly, for TR and RR versions

day that fix bugs for each version. The ever increasing mean number of commits per day is due to the ever increasing number of bug fix changes, which corresponds to more than 90 % of all commits (and is still growing). It is as if the RRs are under continuous maintenance, i.e., instead of focusing on enhancement changes with maintenance activities, development focuses especially on bug fixing changes. We can also observe an increase of 10 % between the percentage of bug fixing changes per day for TR releases and RR releases (Fig. 13b). The median percentage for TR releases is 0.76, while it is 0.88 for RR releases, leading the Wilcoxon rank sum test to reject H_{07}^{3} with a p-value of 0.002797.

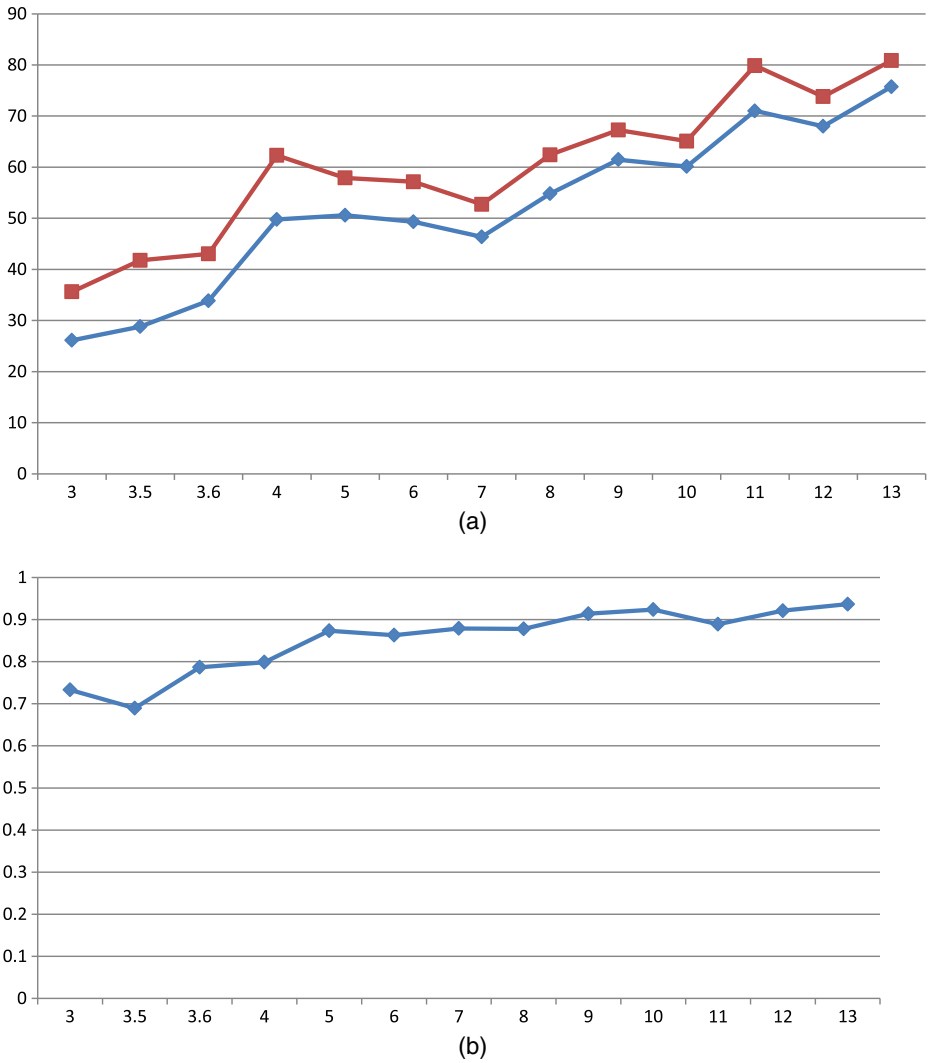


Fig. 13 The mean (**a**) number and (**b**) percentage of Firefox commits that fix bugs per day. The red curve shows the total number of commits per day

Feedback QA Engineers To explain the continuous maintenance observed for RR, a QA engineer stated that "*Not all **major** releases these days have a whiz-bang new feature...while we were still following the traditional release model, Mozilla would still put out dot-releases (security bug fix releases) [i.e., minor releases] at about the same cadence as rapid releases now. That's pretty similar to the continuous maintenance you mention*".

Size, churn and complexity have a significantly weaker correlation with the analyzed crash and bug measures than development and cycle time. Figure 14 shows the relative importance of the five metrics used in the multivariate regression models with which we tried to explain the crash and bug measures of RQ1 and RQ2. The model for median

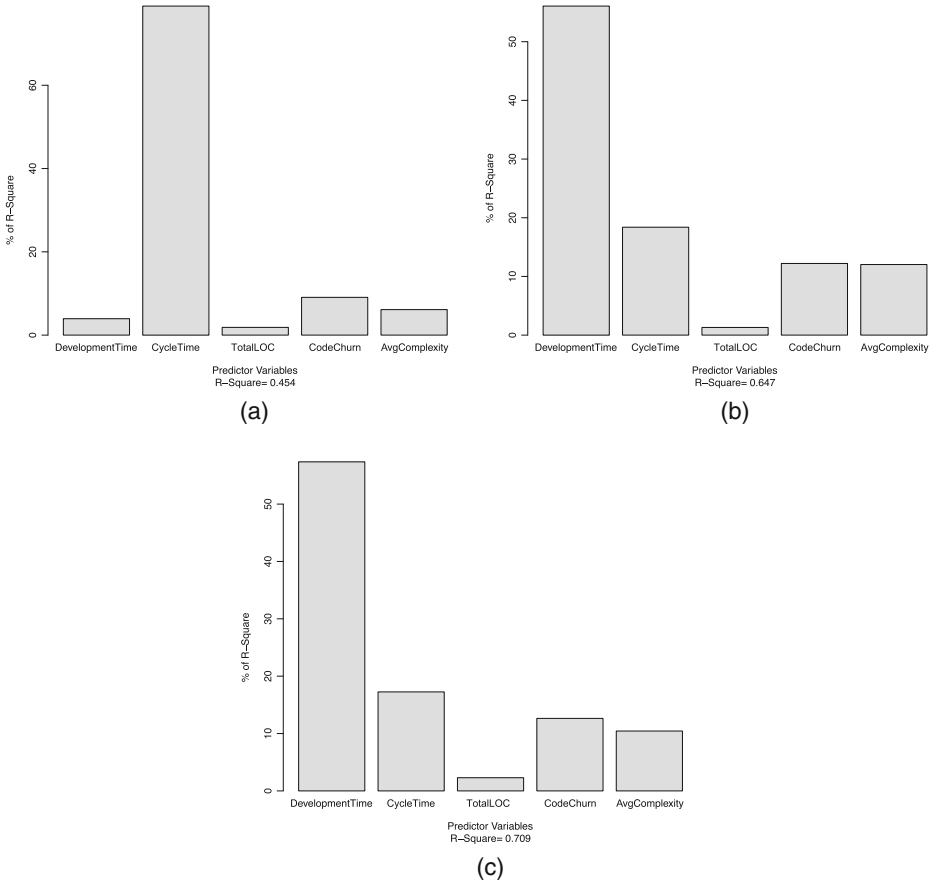


Fig. 14 Percentage of R^2 of the multivariate regression models for (**a**) number of median daily crash count, (**b**) median daily number of post-release bugs reported and (**c**) percentage of post-release bugs fixed, for each of the confounding metrics

uptime did not have any significant metric, hence we discarded the model. We can see that the bug-related models (Fig. 14b and c) have very high R^2 values of 0.647 and 0.709, respectively. This means that at least 64.7 % of all observations (releases) can be explained using those models. The crash-related model (Fig. 14a) obtains a slightly lower R^2 of 0.454.

We can see that in the bug-related models, the development time explains most of the models' R^2, while for the crash-related model it is the cycle time. This means that the median daily crash count changes almost exclusively are correlated to the switch in release model (TR to RR), whereas for the bug-related models it is actually changes in development time that mostly explain the findings. For TR releases, development could start arbitrarily early during development of the previous releases, while for RR releases development time is almost always identical to cycle time, i.e., development starts when the previous release cycle starts. Hence, again, the choice of release model plays an important role. In other words, the confounding factors related to size, churn and complexity do not explain the observed phenomena well, it is the choice of release model that counts.

More developers are working on each rapid release version, resulting in less change per developer. However, those changes touch more files and (similar to TR minor releases) the majority of them fix bugs instead of adding new features.

5 Discussion

Here, we reflect on rapid releases in general, as well as on the threats to validity of our empirical study.

5.1 Rapid Releases

Given that Mozilla Firefox has been one of the pioneering adopters of rapid release, we asked the two QA engineers on their personal views about the pros and cons of rapid releases, as well as concrete advice for future adopters.

Both engineers unanimously mentioned predictability and pace of innovation as the top advantages. Rapid releases make releases more predictable: *"a regular cadence of releases lets features ship when they are ready. A single questionable feature won't derail other features that are already done"*. This predictability and repeated execution of the same release engineering flow motivated and enabled the team to fully automate the release engineering process, whereas before some parts were not executed often enough to warrant the effort. This automation in turn enables the whole organization, in particular release engineering management, to better estimate the timing of the upcoming release.

The pace of innovation refers to the fact that *"new features don't languish for 12-18 months waiting for a release vehicle. Any change is always within 18 weeks of being released (and can be fast-tracked given its importance, riskiness, extra QA). This has important knock-on effects for developer morale and industry competitiveness"*. This means that there is *"less incentive to cram features which aren't finished into a release, and it is more palatable to disable a feature on the testing channels if issues are found"*. This in turn improves the user experience, since missing features might show up in the next (near) release. Overall, the QA engineers note that they *"get new features and fixes out to users faster"*.

The QA engineers provided four disadvantages of rapid releases. First, the switch to rapid releases requires thorough preparation. In the case of Firefox, *"The first few releases were painful. Automation was cobbled together quickly, and required much manual attention and intervention. We weren't given much time to prepare for the transition from 12+ month release cycles to 6 weeks"*. As mentioned earlier, Firefox 4 was a kind of crisis period, with too many product and process changes at once.

A second disadvantage is the lack of oversight: *"If you spend most of each 6 week cycle trying to figure out the current status of the release and can't manage hand-offs between teams, you're not being very agile and you end up shipping less new code than you could otherwise"*. Mozilla realized that there was a *"lack of a single team or person driving the release process for each release"*. By creating a dedicated release management team for this task, they were able to overcome this problem.

A third disadvantage is the danger that *"more frequent changes break some extensions more often, especially the extensions that use compiled code"*. Extensions are third party plugins running on top of Firefox that provide additional functionality like search bars or debugging tools. A thriving ecosystem of thousands of ecosystems have made Firefox one of the most popular browsers in the world, since everyone can easily tweak and customize his or her browser by installing different extensions. Hence, a higher risk of breaking extensions is indeed a major issue for the project.

The fourth disadvantage is the fact that *"some users don't like the frequent changes and want their product to be more "stable""*. As mentioned earlier, enterprise customers have trouble stabilizing their platforms in time for the next release (Shankland 2011), while customer support costs are increasing because of the frequent upgrades (Kaply 2011). Measures like long-term support releases have been taken to reduce the impact of this problem.

Lastly, we asked the QA engineers for advice for future adopters of rapid releases. Two pieces of advice were related to the disadvantages above. In particular *"Fight through the initial pain caused by lack of automation and lack of process. The payoff is worth it."* and *"End-to-end oversight is the quickest way to make your release process more rapid. Hand-offs go through a single person or group and someone always knows what state the release is in"*.

The final piece of advice deals with the role of testing in the multi-stage release process of Firefox. One of the QA engineers noted that *"Rapid release requires a fairly sizeable testing population. We have 25k nightly users and 1.2M beta users. Rapid release would not be possible without these. If you are a small project, consider whether you can get enough testing in a short cycle to ensure quality"*. In related work, we started to verify this interesting claim (Mäntylä et al. 2013).

5.2 Threats to Validity

We now discuss the threats to validity of our study following common guidelines for empirical studies (Yin 2002).

Construct Validity Threats concern the relation between theory and observation. In this work, these threats are mainly due to measurement errors. We compute source code metrics using the open source tool SourceMonitor. We extract commit, crash and bug information by parsing their corresponding TXT (mercurial log), HTML (crash reports) and XML (bug reports) files. To normalize the impact of the release cycle, we divided several metrics, such as the number of post-release bugs, by the length of the release cycle. Although this enables comparison across release cycle models, there might be better ways to normalize the data.

Threats to Internal Validity concern our selection of subject systems, tools, and analysis method. Although we selected Firefox to control for development process and other changes before and after the migration to a rapid release cycle, some of the findings might still be specific to Firefox's development process, as we saw in RQ3. However, the feedback that we received for RQ1 and RQ2, as well as for the general (dis)advantages of rapid release, show that most of our findings are indeed related to rapid release. Furthermore, we only considered coding activities as a proxy for the software development process. In related work, we have been studying the impact of system testing on rapid release (Mäntylä et al. 2013), but other software development activities still need to be analyzed. Finally, we

only obtained feedback from two Mozilla QA engineers. This might have introduced bias in our qualitative results, however in many cases both engineers confirmed each other's statements.

Conclusion Validity Threats concern the relation between the treatment and the outcome. We paid attention not to violate assumptions of the constructed statistical models.

Reliability Validity Threats concern the possibility of replicating this study. We attempt to provide all the necessary details to replicate our study. The Mercurial repository of Firefox is publicly available to obtain the source code and the commit history of Mozilla Firefox. Both the Socorro crash repository and Bugzilla repository are also publicly available. SourceMonitor is an open source code measurement tool (Software 2012).

Threats to External Validity concern the possibility to generalize our results. Although this study is limited to Mozilla Firefox, our previous findings (Khomh et al. 2012) on the time it takes users to adopt a new version of Firefox are consistent with the findings of previous studies on Google Chrome, which has been following a rapid release model for a much longer time (Baysal et al. 2011). Nevertheless, further studies on different systems are desirable. Also, we only studied bug reports that were linked to crashes. Further studies on all bug reports are needed.

6 Related Work

To the best of our knowledge, this study, together with our previous work (Khomh et al. 2012), is the first attempt to empirically quantify the link between release cycle models and software quality in a controlled setting.

Since open source projects have been using agile methods for a long time , many projects adopted short release cycles. Ten years ago, Zhao et al. found that 54 % of the open source apps released at least once per month. Five years later, Otte et al. (2008) found slightly contrasting numbers (on a different set of apps), i.e., 49.3 % released at least once per 3 months. Although this is still relatively rapid, it is not clear why this shift has happened. In any case, modern commercial software projects (Brown 2011; Jenkins 2011) and open source projects backed by a company (Shankland 2010; Gamma 2005) have embraced shorter release cycles.

A lot of work has focused on enabling consistent, short release cycles. For example, van der Storm (2005) and Dolstra et al. (2004) developed infrastructure to automatically build, package and deploy individual code changes . Mainstream continuous integration servers (Duvall et al. 2007) automatically run sets of unit, integration or even acceptance tests after each check-in, while more advanced environments are able to run such tests in a massively parallel way in the shorter time in between releases (Porter et al. 2006). The combination of these ideas and practices have led to the concept of continuous delivery (Humble and Farley 2010), which uses highly automated infrastructure to deploy new releases in record time. Amazon, for example, deploys on average every 11.6 seconds (Jenkins 2011), achieving more than 1,000 deployments per hour.

Despite all this work on achieving and pushing for shorter release cycles, there is hardly any empirical evidence that it really improves product quality, except for various developer surveys (Kong et al. 2009; VersionOne 2009). Escrow.com reduced its release cycle to iterations of 2 weeks (Hodgetts and Phillips 2002), resulting in a reduction of the number of

defects by 70 %. However, since many agile techniques and team restructurings were introduced at once, this improvement in quality cannot be related to shorter release cycles alone. Marschall (2007) found that short release cycles require a steady flow of releases in order to control the number of reported bugs.

Kuppuswami et al. (2003) built a simulation model to analyze the effects of each XP practice on development effort. Small, incremental releases reduce the development effort needed by 2.67%, but no link with software quality was made. Stewart et al. (2005) tried to relate code quality to release frequency, number of releases and the change of size across releases, but could not derive any conclusions.

Releasing too frequently not only decreases the time to run tests, but it also might make customers weary of yet another update to install (Porter et al. 2006; Jansen and Brinkkemper 2006). For this reason, many projects do not automatically distribute each release to their customers. For example, although the Eclipse project uses 6-week release cycles, the resulting milestone releases are only available to interested users and developers (Gamma 2005), similar to how the NIGHTLY, AURORA and BETA Firefox channels are only targeted at specific groups of users. Clear communication about each channel/release is necessary to make sure that the intended user group deploys the new release and provides feedback about it (Gamma 2005; Jansen and Brinkkemper 2006).

The work that is most closely related to ours is that of Baysal et al. (2011). It compares the release and bug fix strategies of Mozilla Firefox and Google Chrome based on browser usage data from web logs. At that time, Firefox was still in the 3.x series, i.e., before its transition to a shorter release cycle, whereas Chrome had been following a short release cycle since its birth. Although the different profiles of both systems made it hard to compare things, the median time to fix a bug in the TR system (Firefox) seemed to be 16 days faster than in the RR system (Chrome), but this difference was not significant. We found the opposite, i.e., Firefox RR fixes bugs faster than Firefox TR. However, the findings about staleness in the TR system confirm our findings (Khomh et al. 2012).

Our paper eliminates the inconsistency between the two compared systems, by focusing on one project (Firefox). We believe that this allows to make more accurate claims regarding RR versus TR models. Furthermore, we use actual field crash data to assess quality from the customers' perspective, and we contacted members of the project to validate our findings.

More recently, we performed a study on the Litmus system testing infrastructure of Firefox, which manages system tests (e.g., "open a tab and browse to this website") (Mäntylä et al. 2013). We found that, although the density of system test execution increased after the switch to rapid release, a smaller team of test volunteers was available for performing the manual system tests, dropping from more than thousand human testers to thirty testers. Those testers ran a smaller variety of test cases than before, consisting of a fixed core of test cases and a variable, risk-based set of additional tests. Tests also targeted less platforms, but more thoroughly. These findings complement the findings of the current paper, which considers defects that slipped through the different test stages as well as how effective they were addressed.

Finally, some papers have studied the characteristics of specific kinds of bugs, such as performance bugs (Zaman et al. 2012), concurrency bugs (Lu et al. 2008), bugs in machine learning systems (Thung et al. 2012) and server bugs (Sahoo et al. 2010). All those studies primarily focus on data from bug repositories and version control systems, whereas we study crash report data. Li et al. (2012) on the other hand investigated the use of pre- and post-release crash data for bug predictions. Furthermore, we study bugs in order to understand the implications of a rapid release model on software quality.

7 Conclusion

The increased competitiveness of today's business environment has prompted many companies to adopt shorter release cycles. Among the various benefits and risks of rapid releases, the impact of this adoption on software quality has not been empirically studied before. In this paper, we analyze the evolution of Mozilla Firefox during the period in which it shifted from a traditional release model to a rapid release model in order to understand potential changes in field quality (users) and bug fixing (developers).

We find that similar amounts of crashes occur before and after the switch to RR (due to increased integration), yet users seem to experience crashes earlier during run-time (partly explained by simultaneous changes in the software architecture). Furthermore, bugs are fixed faster under rapid release models, but proportionally less bugs are fixed compared to the traditional release model (hard bugs are postponed to later, still nearby releases).

From a coding activity point of view (as a proxy for the development process), on the one hand more developers are working on each rapid release version, resulting in less change per developer, but those changes touch more files. Overall, the complexity of the source code developed under both models did not differ significantly. Focus has shifted to continuous maintenance, i.e., the majority of changes fix bugs instead of adding new features.

Feedback from two QA engineers suggests that most of the coding activity changes were unrelated to the switch to rapid release, whereas the real challenges were related to the need to automate the release engineering process as well as the need for clear ownership and processes for producing a release. More case studies are needed, especially on the latter factors, to better understand the impact of changes in release cycle on the quality of software systems.

Acknowledgments We would like to thank the two Mozilla QA engineers who provided feedback on our findings. Their statements are accounts of personal experience and opinion, and are in no means whatsoever an official statement from Mozilla.

References

HP (2011) Shorten release cycles by bringing developers to application lifecycle management. HP Applications Handbook, Retrieved on Febuary 08, 2012

Mozilla (2011) Mozilla puts out firefox 5.0 web browser which carries over 1,000 improvements in just about 3 months of development. InvestmentWatch on June 25th, 2011. Retrieved on January 12, 2012

Beck K, Andres C (2004) Extreme programming explained: embrace change 2nd edn. Addison-Wesley

Shankland S (2011) Rapid-release firefox meets corporate backlash. http://cnet.co/ktBsUU

Kaply M (2011) Why do companies stay on old technology? Retrieved on January 12, 2012

Shankland S (2011) Mozilla proposes not-so-rapid-release firefox. CNET. Retrieved on February 08, 2012

Vaughan-Nichols SJ (2012) The truth about goobuntu: Google's in-house desktop ubuntu linux. http://www.zdnet.com/the-truth-about-goobuntu-googles-in-house-desktop-ubuntu-linux-7000003462/

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

Porter A, Yilmaz C, Memon AM, Krishna AS, Schmidt DC, Gokhale A (2006) Techniques and processes for improving the quality and performance of open-source software. Softw Process Improv Pract 11:163–176

Downer T (2011) Some clarification and musings. Accessed 6 Jan 2012

Li PL, Kivett R, Zhan Z, Jeon Se, Nagappan N, Murphy B, Ko AJ (2012) Characterizing the differences between pre- and post- release versions of software. In: Proceedings of the 33rd international conference on software engineering (ICSE). pp 716–725

Khomh F, Dhaliwal T, Zou Y, Adams B (2012) Do faster releases improve software quality? an empirical case study of mozilla firefox. In: Proceedings of the 9th working conference on mining software repositories (MSR). pp. 179–188
Ltd. RS (2012) Web browsers (global marketshare). Roxr Software Ltd. Retrieved on January 12, 2012
Shankland S (2010) Google ethos speeds up chrome release cycle. http://cnet.co/wlS24U
Sicore D (2011) New channels for firefox rapid releases. The Mozilla Blog. 2011-04-13. Retrieved on January 12, 2012
Rouget P (2012) Shaping a firefox feature - how does it work? http://paulrouget.com/e/featuredesign/
Mozilla (2013) Auto-tools/automation development. https://wiki.mozilla.org/QA/Automation_Services
Mäntylä M, Khomh F, Adams B, Engstrom E, Petersen K (2013) On rapid releases and software testing. In: Proceedings of the 29th IEEE international conference on software maintenance (ICSM). Eindhoven, The Netherlands. To appear
Paul R (2011) Mozilla outlines 16-week firefox development cycle. Accessed 6 Jan 2012
Mozilla (2011) Socorro: Mozilla's crash reporting system. https://crash-stats.mozilla.com/home/products/firefox. Accessed 29 March 2011
Khomh F, Chan B, Zou Y, Hassan AE (2011) An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox. In: Proceedings of the 18th working conference on reverse engineering (WCRE)
Herraiz I, Shihab E, Nguyen THD, Hassan AE (2011) Impact of installation counts on perceived quality: a case study on debian. In: Proceedings of the 18th working conference on reverse engineering (WCRE). pp 219–228
Hollander M, Wolfe DA (1999) Nonparametric statistical methods, 2nd edn. Wiley
Marschall M (2007) Transforming a six month release cycle to continuous flow. In: Proceedings of the conference on AGILE. pp 395–400
Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? In: Proceedings of the 2005 international workshop on mining software repositories (MSR). pp 1–5
Fox J (2008) Applied regression analysis and generalized linear, 2nd edn. Sage Publications
Johnson JW (2000) A heuristic method for estimating the relative weight of predictor variables in multiple regression. Multivar Behav Res 35:1–19
Yin RK (2002) Case study research: design and methods, 3rd edn. SAGE Publications
Software C. (2012) Sourcemonitor. Accessed 12 Jan 2012
Otte T, Moreton R, Knoell HD (2008) Applied quality assurance methods under the open source development model. In: Proceedings of the 32nd Annual IEEE international computer software and applications conference (COMPSAC). pp 1247–1252
Brown AW (2011) A case study in agile-at-scale delivery. In: Proceedings of the 12th international conference on agile processes in software engineering and extreme programming (XP), vol 77. pp 266–281
Jenkins J (2011) Velocity culture (the unmet challenge in ops). Presentation at O'Reilly velocity conference
Gamma E (2005) Agile, open source, distributed, and on-time – inside the eclipse development process. Keynote at the 27th international conference on software engineering (ICSE)
van der Storm T (2005) Continuous release and upgrade of component-based software. In: Proceedings of the 12th international workshop on software configuration management (SCM). pp 43–57
Dolstra E, de Jonge M, Visser E (2004) Nix: a safe and policy-free system for software deployment. In: Proceedings of the 18th USENIX conference on system admin. pp 79–92
Duvall P, Matyas SM, Glover A (2007) Continuous Integration: improving software quality and reducing risk. Addison-Wesley Professional
Humble J, Farley D (2010) Continuous delivery: reliable software releases through build, test, and deployment automation, 1st edn. Addison-Wesley Professional
Kong S, Kendall JE, Kendall KE (2009) The challenge of improving software quality: developers' beliefs about the contribution of agile practices. In: Proceedings of the Americas conference on information systems (AMCIS). 12p
VersionOne (2009) 4th annual state of agile survey. http://bit.ly/6BPw5
Hodgetts P., Phillips D. (2002) 30. in: extreme adoption experiences of a B2B start up. Addison-Wesley Longman Publishing Co. Inc. Extreme programming perspectives
Kuppuswami S, Vivekanandan K, Ramaswamy P, Rodrigues P (2003) The effects of individual xp practices on software development effort. SIGSOFT Softw Eng Notes 28:6
Stewart KJ, Darcy DP, Daniel SL (2005) Observations on patterns of development in open source software projects. SIGSOFT Softw Eng Notes 30:1–5
Jansen S, Brinkkemper S (2006) Ten misconceptions about product software release management explained using update cost/value functions. In: Proceedings of the international workshop on software product management. pp 44–50

Zaman S, Adams B, Hassan AE (2012) A qualitative study on performance bugs. In: Proceedings of the 9th IEEE working conference on mining software repositories (MSR). Zurich, pp 199–208

Lu S, Park S, Seo E, Zhou Y (2008) Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th international conference on architectural support for programming languages and operating systems (ASPLOS). pp. 329–339

Thung F, Wang S, Lo D, Jiang L (2012) An empirical study of bugs in machine learning systems. In: Proceedings of the 23rd IEEE international symposium on software reliability engineering (ISSRE). 271–280

Sahoo SK, Criswell J, Adve V (2010) An empirical study of reported bugs in server software with implications for automated bug diagnosis. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering (ICSE), vol 1. pp 485–494

Foutse Khomh is an assistant professor at the École Polytechnique de Montréal, where he heads the SWAT Lab on software analytics and cloud engineering research (http://swat.polymtl.ca/). He received a Ph.D in Software Engineering from the University of Montreal in 2010. His research interests include software maintenance and evolution, cloud engineering, service-centric software engineering, empirical software engineering, and software analytic. He has published several papers in international conferences and journals, including ICSM, MSR, WCRE, ICWS, JSS, JSP, and EMSE. He has served on the program committees of several international conferences including ICSM, WCRE, MSR, ICPC, SCAM, and has reviewed for top international journals such as SQJ, EMSE, TSE and TOSEM. He is program co-chair of the Workshops track at WCRE 2013, program chair of the Tool track at SCAM 2013, and program chair for Satellite Events at SANER 2015. He is one of the organizers of the RELENG workshop series (http://releng.polymtl.ca) and guest editor for a special issue on Release Engineering in the IEEE Software magazine.

Bram Adams is an assistant professor at the École Polytechnique de Montréal, where he heads the MCIS lab on Maintenance, Construction and Intelligence of Software (http://mcis.polymtl.ca). He obtained his PhD at Ghent University (Belgium). His research interests include software release engineering in general, and software integration, software build systems, software modularity and software maintenance in particular. His work has been published at premier venues like ICSE, FSE, ASE, MSR and ICSM, as well as in major journals like TSE, EMSE, IST and JSS. Bram has been program co-chair of the 2013 International Working Conference on Source Code Analysis and Manipulation (SCAM) and 2015 International Conference on Software Analysis, Evolution and Reengineering (SANER), and he is one of the organizers of the International Workshop on Release Engineering (RELENG), see http://releng.polymtl.ca.

Tejinder Dhaliwal is a software engineer in the Software transformation group at Cisco Systems, Ottawa. His works includes improving software development process and increasing developer's performance. He has also worked with Ericsson Inc, Research in Motion, and Huawei Technologies Ltd as a software engineer. He received his MASc in software engineering from Queen's University in 2012 and B.Eng from University of Rajasthan in 2006.

Ying Zou is an associate professor in the Department of Electrical and Computer Engineering at Queen's University in Canada. She is a Canada research chair in Software Evolution. She is a visiting scientist of IBM Centre for Advanced Studies, IBM Canada Lab. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service-oriented architecture.