

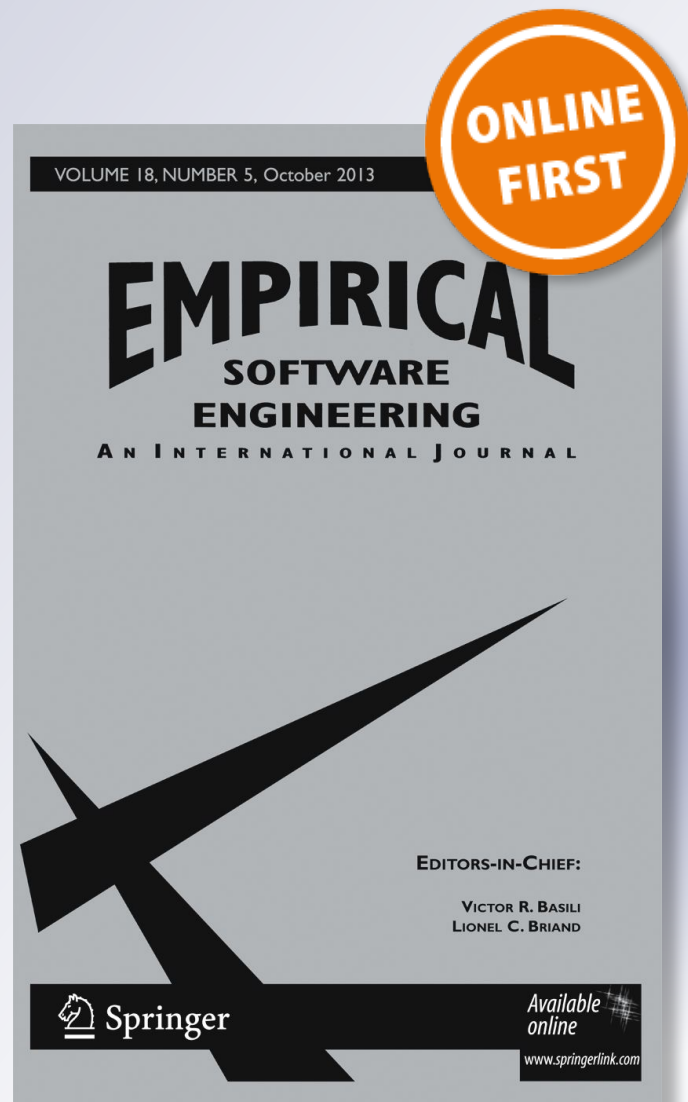
# *An empirical study of the integration time of fixed issues*

**Daniel Alencar da Costa, Shane McIntosh, Uirá Kulesza, Ahmed E. Hassan & Surafel Lemma Abebe**

**Empirical Software Engineering**  
An International Journal


ISSN 1382-3256

Empir Software Eng  
DOI 10.1007/s10664-017-9520-6



**Your article is protected by copyright and all rights are held exclusively by Springer Science +Business Media New York. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at [link.springer.com](http://link.springer.com)".**

# An empirical study of the integration time of fixed issues

Daniel Alencar da Costa<sup>1</sup>  · Shane McIntosh<sup>2</sup> ·  
Uirá Kulesza<sup>1</sup> · Ahmed E. Hassan<sup>3</sup> ·  
Surafel Lemma Abebe<sup>4</sup>

© Springer Science+Business Media New York 2017

**Abstract** Predicting the required time to fix an issue (i.e., a new feature, bug fix, or enhancement) has long been the goal of many software engineering researchers. However, after an issue has been fixed, it must be integrated into an official release to become visible to users. In theory, issues should be quickly integrated into releases after they are fixed. However, in practice, the integration of a fixed issue might be prevented in one or more releases before reaching users. For example, a fixed issue might be prevented from integration in order to assess the impact that this fixed issue may have on the system as a whole. While one can often speculate, it is not always clear why some fixed issues are integrated immediately, while others are prevented from integration. In this paper, we empirically study the integration of 20,995 fixed issues from the ArgoUML, Eclipse, and Firefox projects.

---

Communicated by: Maurizio Morisio

---

✉ Daniel Alencar da Costa  
danielcosta@ppgsc.ufrn.br

Shane McIntosh  
shane.mcintosh@mcgill.ca

Uirá Kulesza  
uira@dimap.ufrn.br

Ahmed E. Hassan  
ahmed@cs.queensu.ca

Surafel Lemma Abebe  
surafel.lemma@aait.edu.et

<sup>1</sup> Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte, Natal, RN, Brazil

<sup>2</sup> Department of Electrical and Computer Engineering, McGill University, Montreal, Quebec, Canada

<sup>3</sup> Software Analysis and Intelligence Lab (SAIL), Queen's University, Kingston, Ontario, Canada

<sup>4</sup> Addis Ababa Institute of Technology, Addis Ababa University, Addis Ababa, Ethiopia

Our results indicate that: (i) despite being fixed well before the release date, the integration of 34% to 60% of fixed issues in projects with traditional release cycle (the Eclipse and ArgoUML projects), and 98% of fixed issues in a project with a rapid release cycle (the Firefox project) was prevented in one or more releases; (ii) using information that we derive from fixed issues, our models are able to accurately predict the release in which a fixed issue will be integrated, achieving Areas Under the Curve (AUC) values of 0.62 to 0.93; and (iii) heuristics that estimate the effort that the team invests to fix issues is one of the most influential factors in our models. Furthermore, we fit models to study fixed issues that suffer from a long integration time. Such models, (iv) obtain AUC values of 0.82 to 0.96 and (v) derive much of their explanatory power from metrics that are related to the release cycle. Finally, we train regression models to study integration time in terms of number of days. Our models achieve  $R^2$  values of 0.39 to 0.65, and indicate that the time at which an issue is fixed and the resolver of the issue have a large impact on the number of days that a fixed issue requires for integration. Our results indicate that, in addition to the backlog of issues that need to be fixed, the backlog of issues that need to be released introduces a software development overhead, which may lead to a longer integration time. Therefore, in addition to studying the triaging and fixing stages of the issue lifecycle, the integration stage should also be the target of future research and tooling efforts in order to reduce the time-to-delivery of fixed issues.

**Keywords** Integration time · Integration delay · Software maintenance · Mining software repositories

## 1 Introduction

Prior studies have explored several approaches that help developers to estimate the time that is needed to fix issues (Anvik et al. 2005; Anbalagan and Vouk 2009; Giger et al. 2010; Kim and Whitehead 2006; Marks et al. 2011; Weiß et al. 2007; Zhang et al. 2013). We use the term *issues* to broadly refer to bugs, enhancements, and new features. Such studies are useful for project managers who need to allocate development resources effectively in order to deliver new releases on time without exceeding budgets.

On the other hand, users and contributors care most about when an official release of a software project will include a fixed issue rather than the time needed to fix that issue. Although an issue may have been fixed, it might take some time before shipping that fix through an official release. For instance, Jiang et al. (2013) find that a reviewed code change might take an additional 1–3 months to be integrated into the Linux kernel. In this paper, we use the term *integration time* to refer to the time that a fixed issue takes to be officially released.

Although one can often speculate, it is not always clear why a fixed issue would not be integrated into an upcoming official release. When the reasons for a long integration time are unclear, users and contributors may become frustrated. For example, on a Firefox issue report, a stakeholder asks: “*So when does this stuff get added? Will it be applied to the next FF23 beta? A 22.01 release? Otherwise?*”<sup>1</sup>

To investigate the integration time of fixed issues, we perform an empirical study of 20,995 issues that are collected from the ArgoUML, Eclipse, and Firefox projects. We

<sup>1</sup>[https://bugzilla.mozilla.org/show\\_bug.cgi?id=883554](https://bugzilla.mozilla.org/show_bug.cgi?id=883554).

investigate (1) how long the integration time of fixed issues typically is—in terms of number of releases and number of days—and (2) which issues suffer from a long integration time in a given project. To that end, this paper addresses six research questions that are structured along two dimensions of integration time as described below.

## 1.1 Concrete Integration Time

- **RQ1: How often are fixed issues prevented from being released?** 34% to 60% of fixed issues within traditional release cycles (the ArgoUML and Eclipse projects) skip at least one release. Furthermore, the delivery of 98% of the fixed issues skip at least one release in the rapidly released Firefox project.
- **RQ2: Does the stage of the release cycle impact integration time?** We observe that issues that are fixed during more stable stages of a release cycle tend to have a shorter integration time. We also observe that fixed issues are unlikely to skip releases solely because they were fixed near a code freeze period.
- **RQ3: How well can we model the integration time of fixed issues?** Our models that are fit to study the integration time in terms of number of releases obtain AUC values of 0.62 to 0.93. Our models that are fit to study the integration time in terms of number of days obtain  $R^2$  values of 0.39 to 0.65.
- **RQ4: What are the most influential attributes for modeling integration time?** We find that the total fixing time that is spent per resolver in the release cycle plays an influential role in modeling the integration time in terms of releases of a fixed issue. On the other hand, we find that the time at which an issue is fixed and the resolver of the issue have a large influence on the integration time in terms of days. Moreover, attributes that are related to the state of the project are the most influential in both kinds of integration time.

## 1.2 Prolonged Integration Time

- **RQ5: How well can we identify the fixed issues that will suffer from a long integration time?** Our models outperform naïve models like random guessing, achieving AUC values of 0.82 to 0.96.
- **RQ6: What are the most influential attributes for identifying the issues that will suffer from a long integration time?** Attributes that are related to the state of the project, such as the integration workload, the period during which issues are fixed, and the fixing time that is spent per resolver are the most influential attributes for identifying the issues that will suffer from a long integration time.

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

## 1.3 Paper Organization

We report our work based on the guidelines that are provided by Singer (1999). In Section 2, we describe the necessary concepts to understand our research. In Section 3, we present the methodology of our empirical study. In Sections 4 and 5, we present the results with

respect to the *integration time* and *prolonged integration time* dimensions. In Section 6, we discuss and relate our observations along the studied integration time dimensions. We perform an exploratory analysis on the backlog of issues of each studied project in Section 7. In Section 8, we discuss the threats to the validity of our conclusions, while we position our work with respect to previous studies (including our own prior work (Costa et al. 2014), which is a precursor to this extended study) in Section 9. Finally, we draw conclusions and proposes avenues for future work in Section 10.

## 2 Background & Definitions

In this section, we present the main concepts necessary to understand our study.

### 2.1 Issue Reports

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

We use the term *issue* to broadly refer to bugs, enhancements, and feature requests. Issues can be filed by users, developers, or quality assurance personnel. Software teams use Issue Tracking Systems (ITS) to track development progress. Examples of ITSs are Bugzilla<sup>2</sup> and JIRA.<sup>3</sup>

Each issue in an ITS has a unique identifier, a brief description of the nature of the issue, and a variety of other metadata. Large software projects receive plenty of issue reports every day. For example, the Eclipse and Firefox projects respectively received an average of 65 and 89 issue reports daily (from January to October 2016) on their respective ITSs.<sup>4,5</sup> The number of filed issues is usually greater than the size of the development team. After an issue has been filed, project managers and team leaders *triage* them, i.e., assign them to developers, while denoting the urgency of the issue using priority and severity fields (Anvik et al. 2006).

After being triaged, issues are then *fixed*, i.e., solutions to the described issues are provided by developers. Generally speaking, an issue may be in an open or closed state. An issue is marked as open when a solution has not yet been found. We consider UNCONFIRMED, CONFIRMED, and IN\_PROGRESS as open states. An issue is considered closed when a solution has been found. Usually, a *resolution* is provided with a closed issue. For example, if a developer made code changes to fix an issue, the state and resolution combination should be RESOLVED-FIXED. However, if the developer could not reproduce the bug, then the state and resolution may be RESOLVED-WORKSFORME.<sup>6</sup> The lifecycle of an issue is documented in detail on the Bugzilla website.<sup>7</sup>

Finally, fixed issues must be integrated into an official release (i.e., releases that are intended for end users) to make them available. The releases that contain such fixed issues

<sup>2</sup><https://www.bugzilla.org/>.

<sup>3</sup><https://www.atlassian.com/software/jira>.

<sup>4</sup><https://bugs.eclipse.org/bugs>.

<sup>5</sup><https://bugzilla.mozilla.org/>.

<sup>6</sup><https://bugzilla.mozilla.org/page.cgi?id=fields.html>.

<sup>7</sup><https://www.bugzilla.org/docs/4.2/en/html/lifecycle.html>.

could be made available every few weeks or months, depending on the project release policy. Releasing every few weeks is typically referred to as a *rapid release* cycle, while releasing monthly or yearly is typically referred to as a *traditional release* cycle (Mantyla et al. 2013).

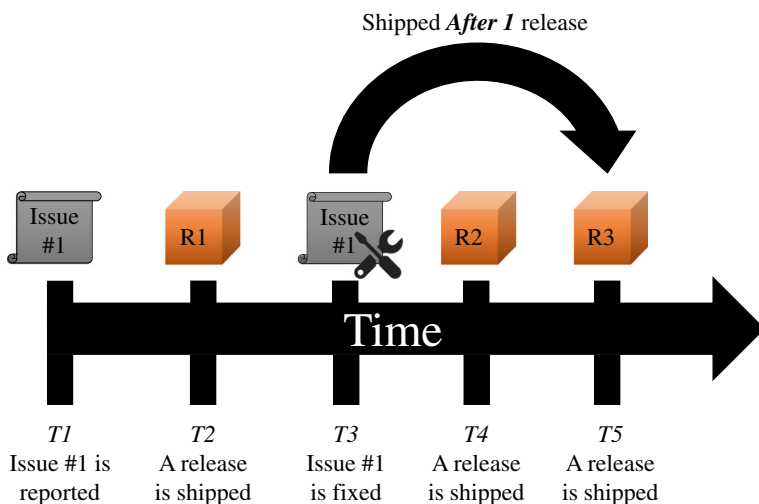
## 2.2 Integration Time

*Integration time* refers to the time between the moment at which an issue is fixed (i.e., changed to the RESOLVED-FIXED status) to the time at which such a fix is shipped to end users. In this study, we analyze two *dimensions* of integration time. The first dimension is comprised of two kinds of integration time, which are: (i) integration time in terms of number of releases and (ii) integration time in terms of days. As for the second dimension, we study (iii) prolonged integration time.

**Definition 1—Integration time in terms of releases.** Figure 1 provides an example of how we measure delivery delay. To compute the delivery delay in terms of number of releases, we count the number of releases that a given addressed issue is prevented from integration. In Fig. 1, Issue #1 is reported at time  $t_1$ , addressed at  $t_3$ , and shipped at time  $t_5$ . The delivery delay in terms of releases for Issue #1 is the number of official releases that are shipped between  $t_3$  and  $t_5$ . Therefore, Issue #1 has a delivery delay of one release.

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

**Definition 1—Prolonged integration time.** Prolonged integration time occurs when the integration time in terms of days (see Definition 2) for a given fixed issue is above one *Median Absolute Deviation* (MAD) of the median integration time of a studied project. MAD is the median of the *absolute deviations* from one distribution's median. The higher the MAD, the greater is the variation of a distribution with respect to its median (Howell 2005; Leys et al. 2013).



**Fig. 1** An illustrative example of how we compute integration time



### 3 Method

In this section, we describe the studied projects, explain how the data was collected and how we study the kinds of integration time that are presented in Section 2.2.

#### 3.1 Subjects

To study integration time, we analyze three subject projects: Firefox, ArgoUML, and Eclipse, which are from different domains and sizes. ArgoUML is a UML modeling tool that includes support for all standard UML 1.4 diagrams.<sup>8</sup> Eclipse is a popular open-source IDE, of which we study the JDT core subproject.<sup>9</sup> Firefox is a popular Web browser.<sup>10</sup>

Figure 2 shows an exploratory analysis of our studied projects. We plot the proportion of issues per priority and severity level, as well as the proportion of issues that were fixed and not fixed (e.g., resolution is *WONTFIX* or *WORKSFORME*). We observe that for the majority of the issues, the priority and severity levels remain at the default value. For example, the vast majority of the priority values are set to P3 (in the Eclipse and ArgoUML projects) or “-” (in the Firefox project). We also observe that Firefox is the project with the highest proportion of fixed issues.

Table 1 shows the studied period and range of releases, as well as the number of releases and issue reports. We focus our study on the releases for which we could recover a list of issue IDs from the release notes. We collected a total of 20,995 issue reports from the three studied projects. Each issue report corresponds to an issue that was fixed and could be mapped directly to a release.

In our analyses regarding integration time of fixed issues, we also investigate the release cycle stages of our subject projects in which a given issue is fixed. To perform this investigation, we study the release engineering process of each subject project. In the following subsections, we provide an overview of the release engineering processes of our subject systems.

##### 3.1.1 Eclipse Release Engineering

The release engineering process of the Eclipse project is composed by *nightly/integration* builds followed by *milestones* builds and *release candidate* builds. Nightly or integration builds are the least stable builds and are tested by the early adopters who are following the eclipse developer mailing lists. For example, integration builds are not supposed to be announced through links, blogs, or wikis that are related to the respective Eclipse project.<sup>11</sup>

*Milestone* and *release candidate* builds are more stable and can be announced by external links. The goal is to reach external early-adopters from outside the developer mailing lists. However, the external links that refer to such builds should warn that they are not as stable as official releases. The main difference between a release candidate build and a milestone build is that a release candidate is followed by a rigorous *test pass*.<sup>12</sup>

<sup>8</sup><http://argouml.tigris.org/>.

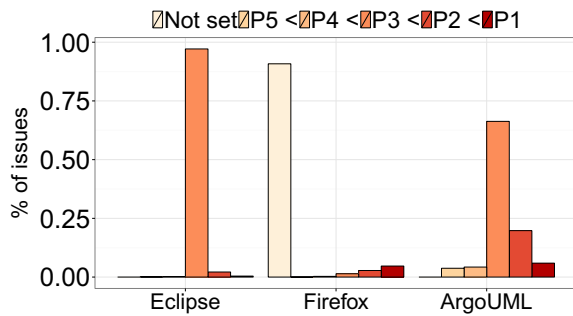
<sup>9</sup><https://www.eclipse.org/>.

<sup>10</sup><https://www.mozilla.org>.

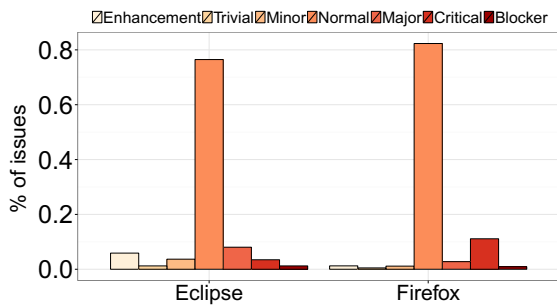
<sup>11</sup>[https://eclipse.org/projects/dev\\_process/development\\_process.php#6\\_Development\\_Process](https://eclipse.org/projects/dev_process/development_process.php#6_Development_Process).

<sup>12</sup>[https://www.eclipse.org/eclipse/development/plans/freeze\\_plan\\_4.4.php](https://www.eclipse.org/eclipse/development/plans/freeze_plan_4.4.php).

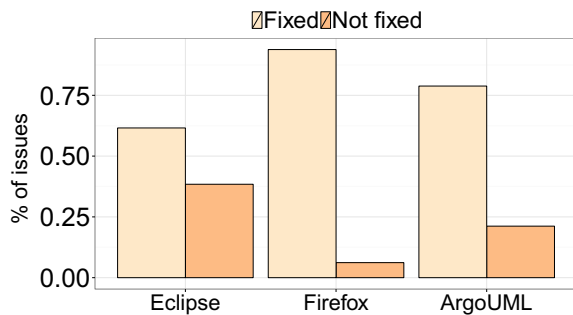




(a) Priority values



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



(c) Fixed issues vs. Not fixed yet

**Fig. 2** Exploratory analysis of the studied projects. We present the ratio of fixed issues per priority, severity, and the ratio of fixed vs. not fixed yet issues (e.g., *WONTFIX* or *WORKSFORME*)

The test process consists of intensive testing activities that are performed by the development team and community to find regression and *stop-ship* bugs. If stop-ship bugs are found late in the process, the release schedule may be slipped to accommodate the fixes for such bugs.<sup>12</sup>

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

**Table 1** Overview of the studied projects

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

We present the number of studied releases, issues, the studied period and the median time between releases

The last release candidate is submitted to a *code freeze* stage. The *code freeze* is a period during which the rules to integrate changes in the software project becomes more strict. For example, new changes may be integrated only if they are solving special requirements such as translations or documentation fixing.<sup>12</sup> Such a period is important because it helps the development team to stabilize the project just before creating an official release.

Official releases are categorized as *major*, *minor*, and *service* releases.<sup>13</sup> Major releases include API changes. Minor releases add new functionalities but are compatible with the API of prior versions. Finally, service releases include bug fixes only (i.e., without significant addition of new functionality). Both major and minor releases have to pass through a *release review* process. A release review aims at getting feedback about the release cycle that was performed. The main goal is to find areas of improvement and if the development process is being open and transparent.<sup>14</sup>

### 3.1.2 Firefox Release Engineering

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

The pipelining process consists on developing releases through several channels. As the release progresses through these channels, the stability of the release increases and less severe bugs are likely to be found. The Firefox project team uses four channels to develop releases: *NIGHTLY*, *AURORA*, *BETA*, and *RELEASE* channels.<sup>15</sup>

The *NIGHTLY* channel produces a release every night (i.e., as soon as features are ready). This nightly release is built from the *mozilla-central* repository and has the lowest stability of the channels.<sup>16</sup> The *AURORA* channel produces a release every six weeks. However, some new features may be disabled if they are not stable enough. At the end of the cycle of the *AURORA* channel (the sixth week), the release management team decides which issues that were further stabilized are good enough to migrate to the *BETA* channel. Again, the goal of the *BETA* channel is to stabilize the new features and disable the features that are not stable enough by the end of the cycle. Finally, the features that are stable

<sup>13</sup><https://www.eclipse.org/projects/handbook/#release>.

<sup>14</sup><https://www.eclipse.org/projects/handbook/#release-review>.

<sup>15</sup><http://mozilla.github.io/process-releases/draft/>.

<sup>16</sup><https://hg.mozilla.org/mozilla-central/>.

enough to survive at the BETA channel are moved further to the RELEASE channel, from which an official major release is produced.<sup>15</sup>

In the Firefox release engineering process, the release schedule is not changed to accommodate issues that are not stable enough by the end of the release cycle. Instead, the development team holds such issues back to be shipped in future releases when a greater degree of stability is achieved.<sup>15</sup> Also, an issue may be integrated directly into the AURORA or BETA channels (i.e., the issue is *uplifted*), but such cases are exceptions (e.g., a very critical security issue should be solved).<sup>15</sup>

The Firefox project also ships *Extended Support Releases* (ESR) that are based on prior official Firefox releases. ESRs are meant for institutions such as business organizations, schools, and universities that must manage their Firefox desktop client. ESRs provide one year of support for security and bug fixes of prior Firefox official releases. ESRs are important for organizations that cannot follow the pace that the Firefox major release evolves.<sup>17</sup>

### 3.1.3 ArgoUML Release Engineering

In the ArgoUML release engineering process, there are five kinds of releases: *development*, *alpha*, *beta*, *stable*, and *stable patch* releases. *Development* releases are the least stable, while *stable* releases are the official releases intended to the users.<sup>18</sup>

*Development* releases are generated during the *development* stage. The *development* stage may take from one to several months. During this stage, the development team strives to produce a *development* release each month. *Development releases* are not supposed to be used by end users. Such releases are only advertised to users if there is a purpose of recruiting new developers to implement and test new features.<sup>18</sup>

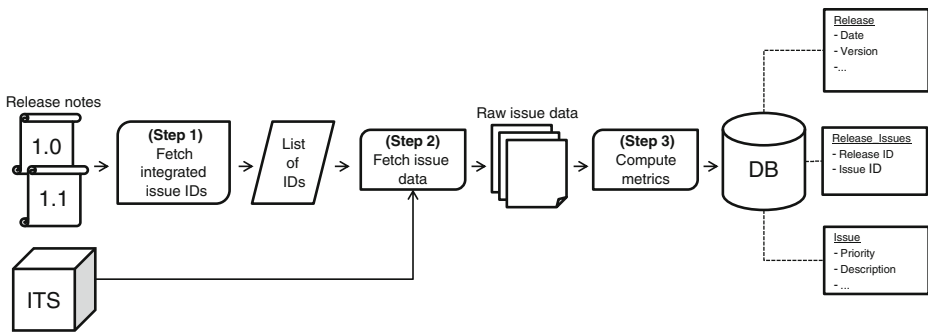
After the *development* stage, the *alpha* stage starts. The *alpha* stage is also referred to as the *enhancement freeze* point. All of the enhancements that are not stable enough before the start of the *alpha* stage are not included into the *stable* release. According to the ArgoUML documentation, the *alpha* stage usually takes a “*couple of weeks*” and the development team strives to make a release each week.<sup>18</sup>

The *alpha* stage is followed by the *beta* stage. The *beta* stage is also referred to as the *bug-fix freeze* point, i.e., all of the (less severe) bug-fixes that could not be completed before the start of the *beta* stage are omitted from the *stable* release. Such remaining bugs are listed on the “*known problems*” document that is to be published along with the *stable* release. *Beta* releases are more stable than *alpha* releases and are also referred as *release candidates*. For example, *beta* releases should not contain high priority bugs (i.e., issues for which the priority is either P1 or P2). The *beta* stage is supposed to last for a couple of weeks with a *beta* release being generated each week. Finally, the *beta* stage is marked by intense testing activities after each release candidate. When the team is confident that the *beta* release is stable enough, the official *stable* release is generated with no code changes from the last *beta* release.<sup>18</sup>

The last kind of ArgoUML release is the *stable patch* release. *Stable patch* releases are generated if critical bugs are found after the publication of the *stable* release. The *stable*

<sup>17</sup><https://www.mozilla.org/en-US/firefox/organizations/faq/>.

<sup>18</sup>[http://argouml.tigris.org/wiki/How\\_to\\_Create\\_a\\_Stable\\_Release](http://argouml.tigris.org/wiki/How_to_Create_a_Stable_Release).



**Fig. 3** Data collection. An overview of our approach to collect the needed data for studying integration time

*patch* release contains the fixes for the eventual critical bugs that are found upon *stable* releases.<sup>18</sup> The ArgoUML team strives to ship a *stable* release every 8 months.<sup>19</sup>

### 3.2 Study Procedures

Figure 3 provides an overview of our data collection approach—how we collect and organize the data to perform our empirical study. We create a relational database that describes the integration of fixed issues in the studied projects. We briefly describe our data sources, and each step involved in the database construction process.

#### 3.2.1 Step 1: Fetch Integrated Issue IDs

In Step 1, we consult the release notes of each studied project to identify the release into which an fixed issue was integrated. A release note is a document that describes the content of a release. For example, a release note might provide information about the improvements that are included in a release (with respect to prior releases), the new features, the fixed issues, and the known problems. The Eclipse, ArgoUML, and Firefox projects publish their release notes on their respective websites.<sup>20</sup>

Unfortunately, release notes may not mention all of the fixed issues that have been integrated into a release. This limitation hinders the possibility of studying issues that were fixed but have not been integrated because we cannot claim that an issue that is not listed in a release note was not integrated (e.g., the development team may forget to list some integrated fixed issues). However, the fixed issues that are listed in a release note are more likely to have been shipped to the end users (i.e., it is unlikely that a release note would mention a fixed issue that was not integrated). Hence, we choose to use release notes as a means of linking fixed issues to releases in our database, despite the incompleteness of such release notes—the release where we claim that an issue has been integrated is more likely to be correct (we elaborate more on this point in Section 8).

The output of Step 1 is a list of the issue IDs that have been fixed and integrated. To retrieve such a list for the Eclipse and Firefox projects, we wrote a script to extract the listed issue IDs from all the release notes and insert them into our database. The retrieved

<sup>19</sup> [http://argouml.tigris.org/wiki/Strategic\\_Planning](http://argouml.tigris.org/wiki/Strategic_Planning).

<sup>20</sup> <https://www.mozilla.org/en-US/firefox/releases/>.

issue IDs are used to fetch the issue report meta-data from the corresponding ITSs. In our database, we also store the dates and version number of each release.

### 3.2.2 Step 2: Fetch Issue Data

We use the collected issue IDs from Step 1 to retrieve information from their corresponding issue reports, which are recorded in the ITSs. Not all release notes of the ArgoUML project list the fixed issues of an official release. When they do, only a few issues are listed (e.g., 1–4).<sup>21</sup> To increase our sample of fixed issues for the ArgoUML project, we rely on its ITS. We use the milestone field of the issue reports to approximate the release into which an issue was integrated. Development milestones are counted towards the next official releases. For example, the development milestone 0.33.7<sup>22</sup> is counted towards the official release 0.34. The output of Step 2 is the raw issue report data that is collected from ITSs.

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

### 3.2.3 Step 3: Compute Metrics

After collecting the release date for each fixed issue, we compute all of the attributes that may share a relationship with the kinds of integration time that are presented in Section 2.2.

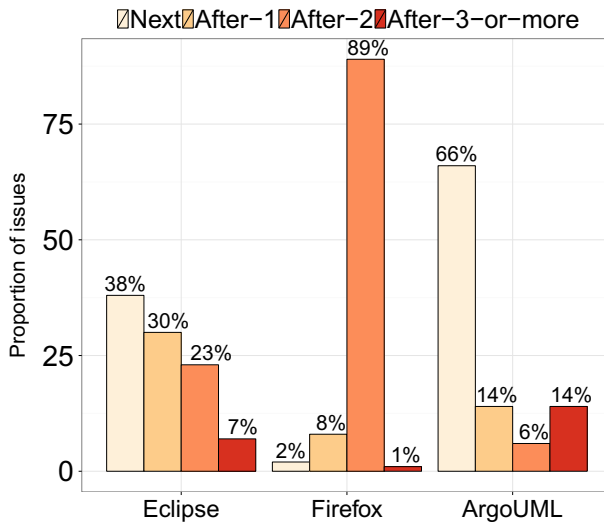
We first compute the integration time of fixed issues in terms of number of releases (see Definition 1). We group this kind of integration time into four buckets: *next*, *after-1*, *after-2*, and *after-3-or-more*. The *next* bucket contains fixed issues that are integrated immediately. The *after-1*, *after-2*, and *after-3-or-more* buckets contain fixed issues for which integration is skipped by one, two, or three or more releases, respectively. Figure 4 shows the distribution of the fixed issues among buckets for each studied project. The ArgoUML project has the highest percentage of fixed issues that fall into the *next* bucket (66%), whereas *next* accounts for only 2% and 38% of fixed issues in the Firefox and Eclipse projects, respectively.

Next, we compute the integration time in terms of number of days (see Definition 2). Figure 5 shows the distribution of integration time in terms of days for each studied project. The Firefox project has the least skewed distribution of integration time. We use both Definitions 1 and 2 of integration time to address RQ1–RQ4.

Finally, we identify issues that have a long integration time in each studied project (see Definition 3). We group fixed issues into *long time* and *normal time* buckets. Fixed issues, of which integration time is at least one MAD above the median integration time of a subject project, fall into the *long time* bucket. Figure 5 shows that a long integration time in one

<sup>21</sup>[http://argouml.tigris.org/wiki/ReleaseSchedule/Past\\_Releases\\_in\\_Detail](http://argouml.tigris.org/wiki/ReleaseSchedule/Past_Releases_in_Detail).

<sup>22</sup>[http://argouml.tigris.org/issues/show\\_bug.cgi?id=4914](http://argouml.tigris.org/issues/show_bug.cgi?id=4914).



**Fig. 4** Distribution of fixed issues per bucket. The issues are grouped into *next*, *after-1*, *after-2*, and *after-3-or-more* buckets

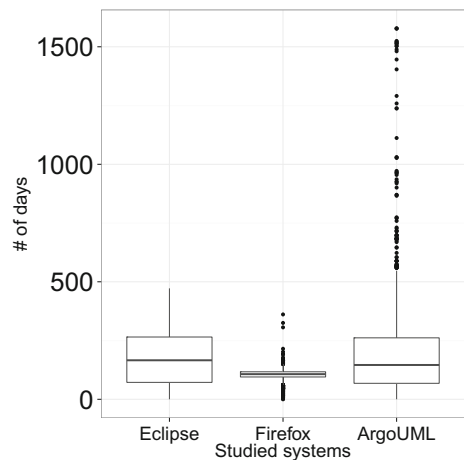
project may be a normal integration time in another project (e.g., the ArgoUML project vs. the Firefox project). This figure highlights the importance of performing this analysis for each project individually. We use this data to address RQ5 and RQ6.

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

### 3.3 Research Questions

In this subsection, we present the research questions of our study. We present the motivation and research approach for each RQ.

**Fig. 5** Integration time in terms of days. The medians are 166, 107, and 146 days for the Eclipse, Firefox, and ArgoUML projects, respectively



### 3.3.1 RQ1: How Often are Fixed Issues Prevented from Being Released?

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

**RQ1: Approach** We compute the integration time of fixed issues in terms of number of releases and number of days (as shown in Definitions 1 and 2). Next, we analyze if fixed issues are being prevented from being released solely because their fix occurs in the end of their release cycle. For example, Rahman and Rigby (2015) observe a rush-to-release in which many issues are fixed near the release date. For each fixed issue, we compute the *fix timing* metric, which is the ratio between (i) the remaining number of days—after an issue is fixed—for an upcoming release over (ii) the duration in terms of days of its respective release cycle (see (1)). The *fix timing* values range from 0 to 1. A *fix timing* value close to 1 indicates that an issue is fixed early in the release cycle because the numerator and denominator of (1) would be close to each other.

$$\frac{\text{\# days that is remaining for a release}}{\text{release cycle duration}} \quad (1)$$

### 3.3.2 RQ2: Does the Stage of the Release Cycle Impact Integration Time?

**RQ2: Motivation** An issue that is *fixed* before the production of a *release candidate* may receive more attention, which may lead to a shorter integration time. Analysis of the impact of integration phase may help researchers and practitioners to reflect on how to reduce integration time or to increase awareness about it.

**RQ2: Approach** For each studied project, we tag fixed issues according to the stage during which they were fixed. For example, if an issue was fixed during the *beta* stage of the Firefox project (i.e., at the BETA channel), we tag such issue as being “*fixed during beta*”. We then compare the distributions of integration time in terms of days (Definition 2) among the different stages of a release cycle. For example, in the Firefox project, we compare the distributions of integration time between the *NIGHTLY*, *ALPHA*, and *BETA* stages, since the *RELEASE* stage corresponds to the official release itself.

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

Finally, we use Cliff’s delta to check the magnitude of the observed differences (Cliff 1993). For example, two distributions may be statistically different, but the magnitude of such a difference may be negligible. The higher the value of the Cliff’s delta, the greater the magnitude of the difference between distributions. We use the thresholds provided by



Romano et al. (2006) to perform our comparisons:  $\delta < 0.147$  (*negligible*),  $\delta < 0.33$  (*small*),  $\delta < 0.474$  (*medium*), and  $\delta \geq 0.474$  (*large*).

We also compute the *fix timing* metric (as in RQ1). However, this time we check whether fixed issues are being prevented mostly because they were performed near a code freeze date—rather than the upcoming release date. Equation (2) shows how we adapt (1) to compute the *fix timing* metric to account for the code freeze date. For the Eclipse project, we consider the date of the last *release candidate* as the code freeze stage, while we consider the date of a *beta stage* as the code freeze stage in the ArgoUML project.

$$\frac{\text{\# days that is remaining for a code freeze}}{\text{release cycle duration}} \quad (2)$$

### 3.3.3 RQ3: How Well Can We Model the Integration Time of Fixed Issues?

**RQ3: Motivation** Several studies have proposed approaches to investigate the time that is required to fix an issue (Anbalagan and Vouk 2009; Giger et al. 2010; Kim and Whitehead 2006; Marks et al. 2011; Weiß et al. 2007; Zhang et al. 2013). These studies could help to estimate when an issue will be fixed. However, we find that a fixed issue may be prevented from integration before reaching users. Even though most issues are fixed well before the next release date, many of them are not integrated until a future release. For users and contributors, however, knowing the integration time of fixed issues is of great interest. In RQ3, we investigate if we can accurately model integration time in terms of number of releases and days (i.e., Definitions 1 and 2 of integration time). Our explanatory models are important to understand which attributes may impact integration time of fixed issues. Moreover, such kind of models could be used by practitioners to estimate when a fixed issue will likely be integrated.

**RQ3: Approach** To study when a fixed issue is integrated, we collect information from both the ITSs and VCSs of the studied systems. We train models using attributes that are grouped in the following families: *reporter*, *resolver*, *issue*, *project*, and *process*.

- **Reporter:** refers to the attributes regarding an issue reporter. Issues that are reported by a reporter who is known to report important issues may receive more attention from the integration team.
- **Resolver:** refers to team members that fix issues. Issues that are fixed by experienced resolvers may be easier to integrate and ship to end users.
- **Issue:** refers to the attributes of issues reports. Project teams use this information to triage, fix, and integrate issues. For example, integrators may not be able to properly assess the importance and impact of poorly described issues, which may increase integration time.
- **Project:** refers to the status of the project when a specific issue is fixed. If the project team has a heavy integration workload, i.e., many fixed issues waiting to be integrated, the integration of newly fixed issues are likely to have a longer integration time.
- **Process:** refers to the process of fixing an issue. A fixed issue that involved a complex process (e.g., long comment threads, large code changes) could be more difficult to understand and integrate.

Tables 2 and 3 describe the attributes that we compute for each family. For each attribute, Tables 2 and 3 present our rationale for using it in our models. We choose these families of attributes because (i) we intend to study a variety of perspectives that might influence integration time and (ii) they are simple to compute using the publicly available data sources (e.g., ITSs and VCs) from our studied systems.

We train exploratory models to study how many releases a fixed issue is likely to be prevented from integration (Definition 1). To study integration time in terms of releases, we use the *random forest* classification technique (Breiman 2001). Random forest is an *ensemble learning* technique that operates by combining a multitude of decision trees at the training stage. Each decision tree uses a random subset of the attributes that are used to explain one phenomenon (e.g., integration time). Next, each decision tree votes for the response bucket (e.g., *next* or *after-1* release(s)) of a given instance. The majority of the votes for a given bucket will be the actual response of the random forest. We choose random forests because they are known to have a good overall accuracy and to be robust to outliers as well as noisy data. Model robustness is important for our study because the data in the ITSs tend to be noisy (Herraiz et al. 2008). In our study, we use the *random forest* implementation provided by the *bigrf* R package.<sup>23</sup>

Because our data has a temporal order, i.e., the values of the attributes for each instance depends on the time at which the issue was fixed, we evaluate our models by adapting the *Leave One Out Cross Validation* (LOOCV) technique. In our LOOCV variation, we first sort the data by the date at which the issues were fixed. Then, we train models to predict each next instance of the data. For example, if issue *A* is fixed before issue *B*, we train a model using *A* and test it using *B*. Furthermore, if issues *A* and *B* are fixed before *C*, we train a model using *A* and *B* and test it using *C*. This process is repeated until we test a model by using the last fixed issue in our data.

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

*Precision* (P) measures the correctness of our models in estimating the number of releases that are necessary to ship a fixed issue. An estimation is considered correct if the estimated integration time is the same as the actual integration time of a fixed issue. Precision is computed as the proportion of correctly estimated integration time for each studied integration bucket (e.g., *next*, *after-1*).

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

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

*Area Under the Curve* (AUC) is used to evaluate the degree of discrimination achieved by the model (Hanley and McNeil 1982). For instance, AUC can be used to evaluate how well our models can distinguish between fixed issues that are prevented from integration into one or two releases. The AUC is the area below the curve plotting the true positive rate against false positive rate. The value of AUC ranges between 0 (worst) and 1 (best). An area greater than 0.5 indicates that the explanatory model outperforms a random predictor. We

<sup>23</sup> Bigrf package <https://cran.r-project.org/src/contrib/Archive/bigrf/>.

**Table 2** Reporter, Resolver and Issue families

Family	Attributes	Value	Definition (d) Rationale (r)
Reporter	Experience	Numeric	<p><b>d:</b> Experience in filing reports for the project. It is measured by the number of previously reported issues of a reporter.</p> <p><b>r:</b> An issue reported by an experienced reporter might be integrated quickly.</p>
	Integration speed	Numeric	<p><b>d:</b> Measured by the median integration time of prior issues that were reported by a given reporter.</p> <p><b>r:</b> If issues that are reported by a given reporter are integrated quickly, future issues reported by the same reporter may also be integrated quickly.</p>
Resolver	Experience <sup>a</sup>	Numeric	<p><b>d:</b> Experience in fixing issues for the project. It is measured by the number of prior issues that were fixed by a given resolver.</p> <p><b>r:</b> An issue that is fixed by an experienced resolver may be easier to integrate.</p>
	Integration speed <sup>a</sup>	Numeric	<p><b>d:</b> Measured by the median integration time of prior fixed issues.</p> <p><b>r:</b> If the prior fixed issues of a particular resolver were quickly integrated, future issues that are fixed by the same resolver may also be quickly integrated.</p>
Issue	Component	Nominal	<p><b>d:</b> The component to which an issue is being reported.</p> <p><b>r:</b> Issues that are related to a given component (e.g., authentication) might be more important, and thus, might be integrated more quickly than issues that are reported to less important components.</p>
	Platform	Nominal	<p><b>d:</b> The platform specified in the issue report.</p> <p><b>r:</b> Issues regarding one platform (e.g., MS Windows) might be integrated more quickly than issues that are reported to less important platforms.</p>
	Severity	Nominal	<p><b>d:</b> The severity level that is recorded in the issue report.</p> <p><b>r:</b> severe issues (e.g., blocking) might be integrated faster than other issues. Panjer observed that the severity of an issue has a large effect on its lifetime for the Eclipse project (Panjer 2007).</p>
	Priority	Nominal	<p><b>d:</b> The priority that is assigned to the issue report.</p> <p><b>r:</b> High priority issues will likely be integrated before low priority issues.</p>
	Stack trace attached <sup>a</sup>	Boolean	<p><b>d:</b> We check whether the issue report has a stack trace attached in its description.</p> <p><b>r:</b> A stack trace attached in the description of the issue report may provide useful information with respect to the cause of the issue, which may quicken the integration of that fixed issue (Schroter et al. 2010).</p>

**Table 2** (continued)

Family	Attributes	Value	Definition (d) Rationale (r)
	Description size	Numeric	<b>d:</b> Description of the issue measured by the number of words in its description. <b>r:</b> Issues that are well-described might be easier to integrate than issues that are difficult to understand.

Attributes of the Reporter, Resolver and Issue families that are used to model the integration time of fixed issues

<sup>a</sup>New attributes that did not appear in our previous work (Costa et al. 2014)

computed the AUC value for a given bucket  $b$  (e.g., *next*) on a binary basis. In other words, the probabilities of the instances were analyzed as pertaining to a given bucket  $b$  or not. For example, when computing the AUC value for the *next* bucket, the AUC value is computed by verifying if an instance belongs to the *next* bucket or not. This process is repeated for each bucket. Therefore, each bucket has its own AUC value.

*Zero-R models* are naïve models that always select the bucket with the highest number of instances. For example, a Zero-R model trained with the Firefox project data would always select *after-2* as the response for each instance.

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

We use the guidelines that are provided by Harrell (2001) to fit our regression models. Figure 6 provides an overview of our model fitting approach. In Step 1, we compute the budget of degrees of freedom that our data can accommodate while keeping the risk of overfitting low. We compute this budget by using the formula  $\frac{n}{15}$ , where  $n$  is the number of issues in our dataset and 15 is a denominator that is recommended by Harrell (2001). In Step 2, we verify the normality assumption of *ordinary least squares*, i.e., it assumes that the response variable  $y$  should follow a normal distribution. Through analysis of the integration time values (i.e., the  $y$  variable), we find that it does not follow a normal distribution, and hence, we apply a log transformation  $[\ln(y + 1)]$  to mitigate such skewness.

In Step 3, we use a variable clustering analysis (Sarle 1990) to remove high correlated variables. For variables within a cluster that have a correlation of  $|\rho| > 0.7$ , we choose only one of them to include in our models. In Step 4, we check the redundancy of the surviving explanatory variables. Redundant variables do not add explanatory power to the models and can distort the relationship between explanatory and response variables. To remove redundant variables we use the `redun` function from the `rms` R package, which fits models

**Table 3** Project and Process families

Family	Attributes	Value	Definition (d) Rationale (r)
Project	Backlog of issues	Numeric	<p><b>d:</b> The number of issues in the RESOLVED-FIXED state at a given time.</p> <p><b>r:</b> Having a large number of fixed issues at a given time might create a high workload on team members, which may affect the number of fixed issues that are integrated.</p>
	Queue position <sup>a</sup>	Numeric	<p><b>d:</b> <math>\frac{\text{rank of the issue}}{\text{all fixed issues}}</math>, where the rank is the position in time at which an issue was fixed in relation to others in the current release cycle. The rank is divided by all of the issues that were fixed until the end of the current release cycle.</p> <p><b>r:</b> An issue that is near the front of the queue is more likely to be quickly integrated.</p>
	Fixing time per resolver <sup>a</sup>	Numeric	<p><b>d:</b> <math>\frac{\sum_{\text{issue}=1}^{\text{total}} \text{fixing time}}{\text{\# of resolvers}}</math>, the sum of the time (measured in terms of days) to fix the issues of the current release cycle over the number of resolvers that worked in that release cycle (Brooks 1975).</p> <p><b>r:</b> The higher the total fixing time that is spent per resolver in fixing issues the less the likelihood of a fixed issue experiencing a large integration time.</p>
	Backlog of issues per resolver <sup>a</sup>	Numeric	<p><b>d:</b> The number of issues in the RESOLVED-FIXED state at a given time for each resolver of the development team.</p> <p><b>r:</b> Having a large number of fixed issues per resolver might create a workload on that resolver to integrate the issue.</p>
Process	Number of impacted files	Numeric	<p><b>d:</b> The number of files that are linked to an issue report.</p> <p><b>r:</b> Integration time might be related to a high number of impacted files because more effort would be required to properly integrate code modifications (Jiang et al. 2013).</p>
	Number of activities	Numeric	<p><b>d:</b> An activity is an entry in the issue's history.</p> <p><b>r:</b> A high number of activities might indicate that much work was necessary to fix the issue, which can impact the integration time of an issue. (Jiang et al. 2013).</p>
	Number of comments	Numeric	<p><b>d:</b> The number of comments of an issue report.</p> <p><b>r:</b> A large number of comments might indicate the importance of an issue or the difficulty to understand it (Giger et al. 2010), which might impact integration time (Jiang et al. 2013).</p>
	Number of tosses	Numeric	<p><b>d:</b> The number of times that the issue's assignee has changed.</p> <p><b>r:</b> The number of changes in the issue assignee might indicate a complex issue to fix or a difficulty in understanding such an issue, which can impact integration time. One of the reasons for changing the</p>

**Table 3** (continued)

Family	Attributes	Value	Definition (d) Rationale (r)
	Comment interval	Numeric	<p>assigned developer is because additional expertise may be required to fix an issue (Jiang et al. 2013; Jeong et al. 2009).</p> <p><b>d:</b> The sum of all of the time intervals between comments (measured in hours) divided by the total number of comments.</p> <p><b>r:</b> A short comment time interval indicates that an active discussion took place, which suggests that the issue is important. (Jiang et al. 2013).</p>
	Churn	Numeric	<p><b>d:</b> The sum of the added lines and removed lines in the code repository.</p> <p><b>r:</b> A higher churn suggests that a great amount of work was required to fix the issue, and hence, verifying the impact of integrating the modifications may also be difficult (Nagappan and Ball 2005; Jiang et al. 2013).</p>
	Fixing time <sup>a</sup>	Numeric	<p><b>d:</b> The number of days between the moment at which an issue is opened the moment at which the issue is fixed (i.e., the issue reaches the RESOLVED-FIXED status) (Giger et al. 2010).</p> <p><b>r:</b> Issues that are fixed quickly might indicate that the necessary code changes are easy to integrate, which may quicken integration time.</p>

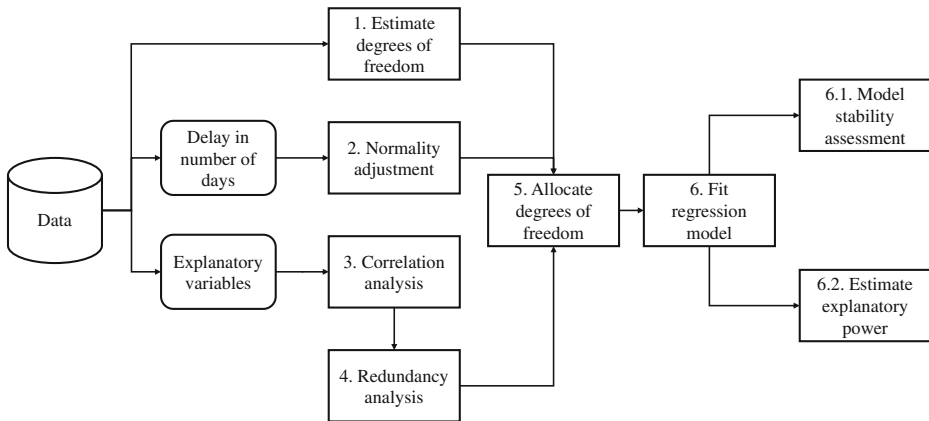
Attributes of the Project and Process families that are used to model the integration time of a fixed issue

<sup>a</sup>New attributes that did not appear in our previous work (Costa et al. 2014)

to explain each explanatory variable using the other explanatory variables. We then discard those explanatory variables that could be estimated with an  $R^2 \geq 0.9$  (the default threshold of the `redun` function).

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

In Step 6, we fit our regression models. To assess the fit of our models (Step 6.1) we use the  $R^2$  metric. The  $R^2$  measures the “*variability explained*” of the dependent variable that is analyzed (Steel and James 1960). For example, a  $R^2$  of 0.4 indicates that 40% of the variability of the dependent variable is being modeled (“*explained*”) by the explanatory variables—the remaining 60% of the variability may be due to external factors that are not being modeled or cannot be controlled. The interpretation of  $R^2$  values depends on the analysis that is being performed. For example, when prediction is the main goal, the  $R^2$  values should be very high (e.g., around 0.7 to 0.9) (Choi and Varian 2012). Low  $R^2$  values (e.g., around 0.20) may also generate important insights in fields such as psychology or social sciences (Bersani et al. 2016).



**Fig. 6** Training regression models. We follow the guidelines that are provided by Harrell (2001) to train regression models, which involves nine activities, from data collection to model validation. The results of Steps 6.2 and is presented in RQ4

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

### 3.3.4 RQ4: What are the Most Influential Attributes for Modeling Integration Time?

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

**RQ4: Approach** To identify the most influential attributes for estimating the integration time in terms of releases (Definition 1), we compute the *variable importance* score for each attribute of our models. The *variable importance* implementation that we use in our study is available within the *bigrf* R package. This implementation computes the importance score based on *Out Of the Bag* (OOB) estimates. Each attribute of the dataset is randomly permuted in the OOB data. Then, the average  $a$  of the differences between the votes for the correct bucket in the permuted OOB and the original OOB is computed. The result of  $a$  is the importance of an attribute.



The final output of the variable importance is a rank of the attributes indicating their importance for the model. Hence, if a specific attribute has the highest rank, then it is the most influential attribute that our explanatory model is using to estimate integration time. Finally, we use the models with the largest training corpus when performing the LOOCV to compute the variable importance scores.

We perform Step 6.2 of Fig. 6 to identify the most influential attributes in our models that we fit to study the integration time in terms of number of days (Definition 2). We evaluate the explanatory power of each attribute by using the Wald  $\chi^2$  maximum likelihood test (Step 6.2). The larger the  $\chi^2$  value, the greater the power that a particular attribute has to model the variability of integration time in terms of days. We use the `anova` function of the `rms` R package.

### 3.3.5 RQ5: How Well Can We Identify the Fixed Issues that Will Suffer from a Long Integration Time?

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

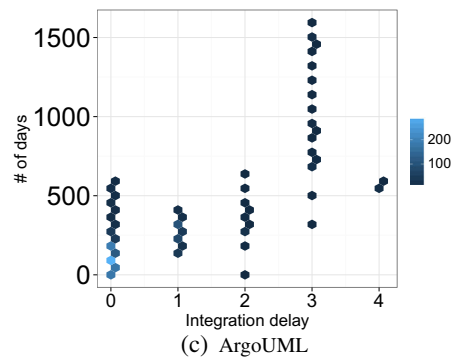
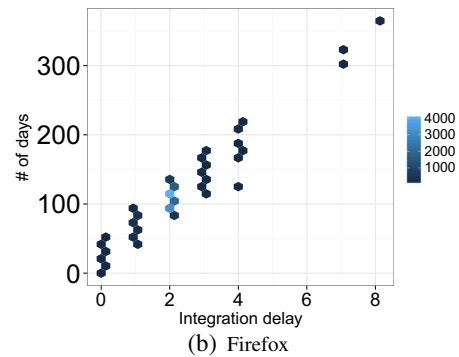
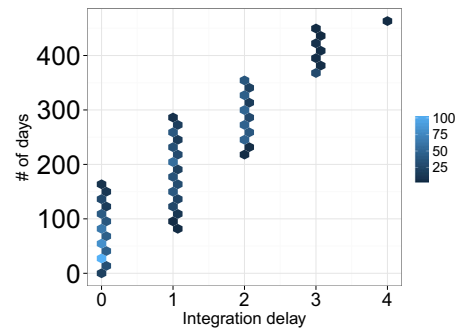
**RQ5: Approach** We calculate prolonged integration time (Definition 3) as described in Section 3.2.3. Indeed, in Fig. 5, we observe that the distribution of integration time of the Eclipse and ArgoUML projects have more variation than the distribution of the Firefox project.

The hexbin plots of Fig. 7 show the relationship between the integration time in terms of releases and days. Hexbin plots are scatterplots that represent several data points with hexagon-shaped bins. The lighter the shade of the hexagon, the more data points that fall within the bin. Indeed, Fig. 7 suggests that the longer the integration time in terms of days, the longer is the integration time in terms of releases. This tendency is more clear in the Eclipse and Firefox projects. In the ArgoUML project, we observe fixed issues with a longer integration time in terms of releases but with a shorter integration time in terms of days. For example, we observe fixed issues with a integration time of four releases that have a shorter integration time in terms of days than fixed issues with a integration time of three releases. Such behaviour in the ArgoUML project may be explained by the skew in the distance between the releases of this project (*cf.* Fig. 9).

Table 4 shows the medians and MADs for each project to identify fixed issues that have a long integration time. For example, a fixed issue have a long integration time in the Firefox project when that issue takes more than 123 days to be integrated. Figure 8 shows the proportion of issues that have a long integration time per project. We observe that 13%, 12%, and 22% of the fixed issues in the Eclipse, Firefox, and ArgoUML projects have a long integration time, respectively.

To train our exploratory models, we produce a dichotomous response variable  $Y$ , where  $Y = 1$  means that a fixed issue has a long integration time, while  $Y = 0$  means that the integration time of that issue is normal. Finally, we train random forest models to study

**Fig. 7** Relationship between integration time in terms of releases and days. We observe that a longer integration time in terms of releases is associated with a longer integration time in terms of days



whether a given fixed issue is likely to have a long integration time. Similar to RQ2, we evaluate our models using *precision*, *recall*, *F-measure*, and *AUC*.

### 3.3.6 RQ6: What are the Most Influential Attributes for Identifying the Issues that Will Suffer from a Long Integration Time?

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

**Table 4** Long integration time thresholds

	Eclipse	Firefox	ArgoUML
Median integration time	166	107	146
Median absolute deviation	142	16	131
Long integration time	>308	>123	>278

We present the median integration time in terms of days, the MAD, and the long integration time threshold for each project

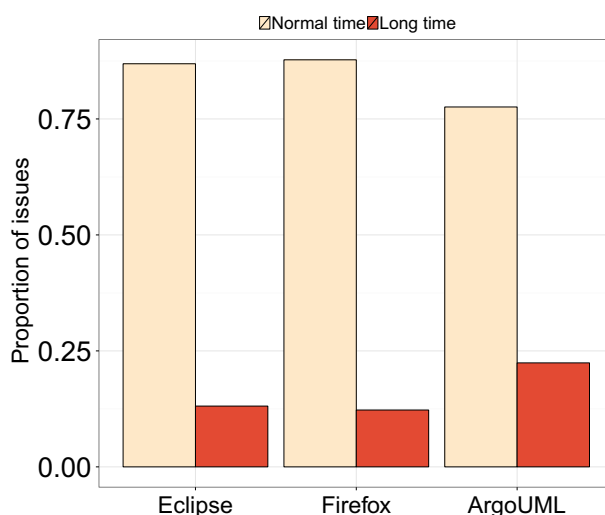
**RQ6: Approach** Similar to RQ4, in this research question, we analyze our explanatory models by computing the variable importance score of the attributes.

## 4 Results for the Concrete Integration Time Dimension

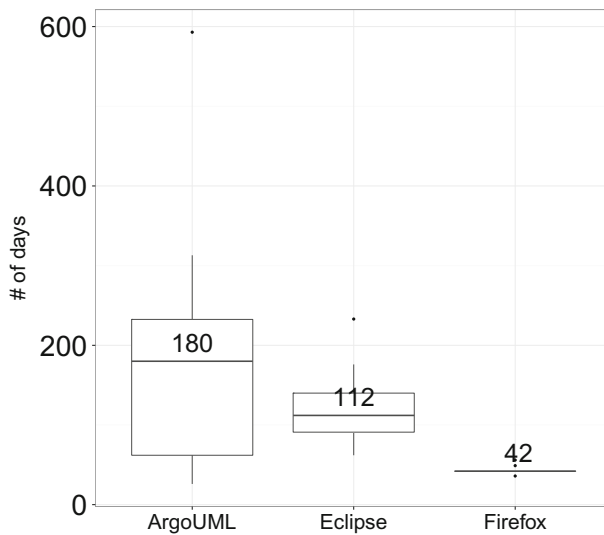
In this section, we present the results with respect to the *concrete integration time* dimension. This dimension involves the investigation of integration time in terms of number of releases and number of days (Definitions 1 and 2). This dimension is comprised of RQ1–RQ4. Below, we present the obtained results for each RQ.

### 4.1 RQ1: How Often are Fixed Issues Prevented from Being Released?

**Fixed Issues Usually Miss the Next Release in the Firefox Project** Figure 9 shows the difference between the studied projects in terms of the time interval between their releases. The median time in days for the Firefox project (42 days) is approximately  $\frac{1}{4}$  that of the ArgoUML project (180 days), and  $\frac{1}{3}$  that of the Eclipse project (112 days). Unlike the



**Fig. 8** Fixed issues that have a long integration time. We present the proportion of fixed issues that have a long integration time per project. 13%, 12%, and 22% of the fixed issues of the Eclipse, Firefox, and ArgoUML projects have a long integration time, respectively



**Fig. 9** Number of days between the studied releases of the ArgoUML, Eclipse, and Firefox projects. The number shown over each boxplot is the median interval

Eclipse and Firefox projects, the distribution for the ArgoUML project is skewed. In addition, Fig. 4 shows that the vast majority of fixed issues for the Firefox project is integrated *after-2* releases, whereas for the Eclipse and ArgoUML projects, the majority is integrated in the *next* release.

The reason for the difference may be due to the release policies that are followed in each project. For example, Fig. 9 shows that the Firefox project releases consistently every 42 days (six weeks), whereas the time intervals between the releases of the ArgoUML project vary from 50 to 220 days. Indeed, the release guidelines for the ArgoUML project state that the ArgoUML team should release at least one stable release every 8 months (see Section 3.1.3). The delivery consistency of the Firefox releases might lead to fixed issues being prevented from a greater number of releases, since the Firefox project rigidly adhere to a six-week release schedule despite accumulating issues that could not be integrated (see Section 3.1.2).

Although a fixed issue usually misses the next release in the Firefox project, issues are usually shipped faster when compared to the other projects. Indeed, Fig. 5 shows that fixed issues in the Firefox project take a median of 107 days to be released, while it takes 166 and 146 days in the Eclipse and ArgoUML projects, respectively.

**34% to 60% of Fixed Issues had Their Integration Prevented from at Least One Release in the Traditionally Released Projects** Figure 4 shows that 98% of the fixed issues in the Firefox project are prevented from integration in at least one release. However, for the projects that adopt a more traditional release cycle, i.e., the ArgoUML and Eclipse projects, 34% to 60% of the fixed issues are prevented from integration in at least one release. This result indicates that even though an issue is fixed, integration may be prevented by one or more releases, which can frustrate end users.

**Many Issues that were Prevented from Integration are Fixed Well Before the Upcoming Release Date** Fixed issues could be prevented from integration because they

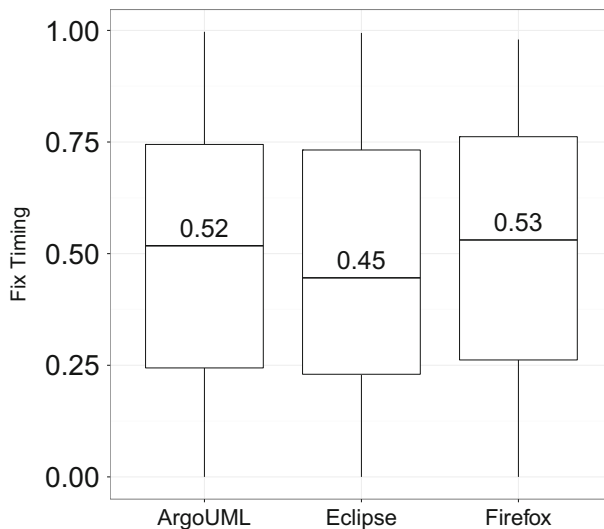
were fixed late in the release cycle, e.g., one day or one week before the upcoming release date. To check whether fixed issues are being prevented from integration mostly because they are being fixed late in the release cycle, we compute the *fix timing* metric.

Figure 10 shows the distribution of the *fix timing* metric for each project. The smallest *fix timing* median is observed for the Eclipse project, which is 0.45. For the ArgoUML and Firefox projects, the median is 0.52 and 0.53, respectively. The *fix timing* medians are roughly in the middle of the release. Moreover, the boxes extend to cover between 0.25 and 0.75. The result suggests that, in the studied projects, issues that are prevented from integration are usually fixed  $\frac{1}{4}$  to  $\frac{3}{4}$  of the way through a release. Hence, it is unlikely that most fixed issues are prevented from integration solely because they were fixed too close to an upcoming release date.

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

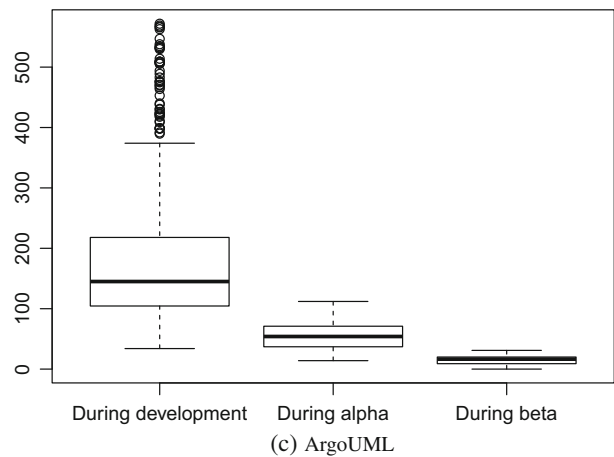
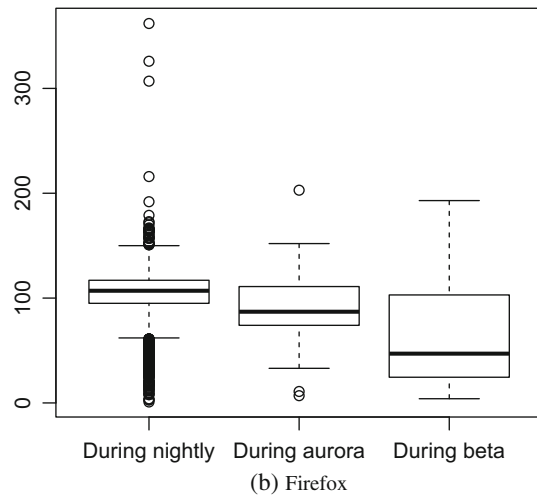
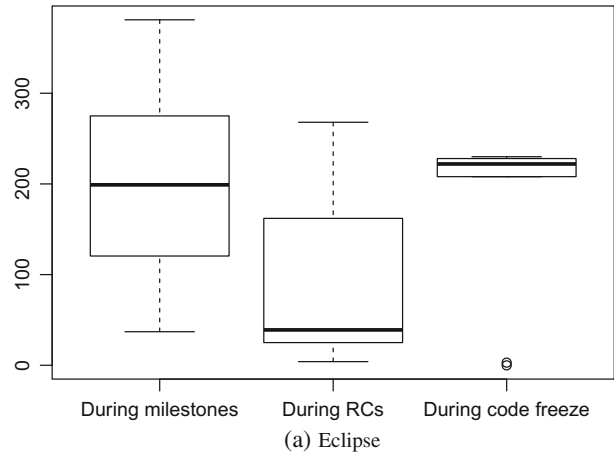
## 4.2 RQ2: Does the Stage of the Release Cycle Impact Integration Time?

**Issues that are Fixed During More Stable Stages of a Release Cycle Have a Shorter Integration Time** Figure 11 shows the distributions of integration time (in terms of days) per each release cycle stage of the studied projects. For the Eclipse project, the stages are divided into *milestones*, *RCs* (Release Candidates), and *code freeze* (see Section 3.1.1). Indeed, issues that are fixed during *RCs* have a shorter integration time when compared to issues that were fixed during *milestone* releases. For the difference between *milestones* and *RCs*, we observe a  $p = 1.47 \times 10^{-52}$  and a *large* effect-size of  $\delta = 0.63$ . All of the



**Fig. 10** Fix timing metric. We present the distribution of the *fix timing* metric for fixed issues that are prevented from integration in at least one release

**Fig. 11** Integration time during release cycle stages. Issues that are fixed during more stable stages of a release cycle are likely to have a shorter integration time



**Table 5** Statistical analysis

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

An overview of the  $p$ -values and  $\delta$ s that are observed during our statistical analyses

$p$ -values and  $\delta$ s of our statistical analysis are shown in Table 5. Even though integration time seems to be larger during the *code freeze* stage, we do not observe a significant  $p$ -value when comparing the *code freeze* stage to the other stages. In fact, only ten issues were fixed during the *code freeze* stage in our data, which impairs statistical observations of trends in such a stage.

For the Firefox project, we observe that integration time tends to be shorter as fixes are performed along more stable stages. For example, by comparing the integration time values between the *NIGHTLY* and *AURORA* stages, we observe a  $p = 5.1 \times 10^{-49}$  and a *medium* effect-size of  $\delta = 0.40$  (the other comparisons are shown in Table 5).

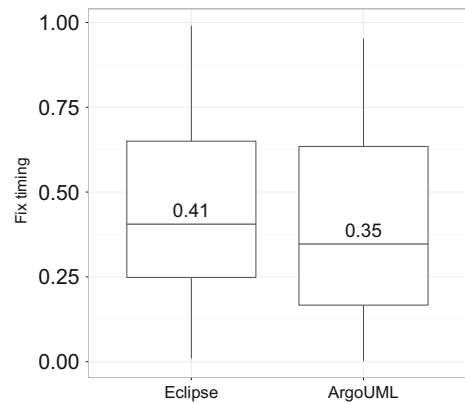
Finally, for the ArgoUML project, we also observe a trend of shorter integration time as the fixes are performed during more stable stages of release cycles. For instance, when we compare the integration time of fixed issues of the *alpha* and *beta* stages, we obtain a  $p$ -value of  $3.98 \times 10^{-09}$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

**Many Issues that are Prevented from Integration are Fixed Well Before the Code Freeze Stage of Their Respective Release Cycle** We compute the *fix timing* metric that we present in RQ1. However, instead of counting the number of days until an upcoming release, we count the number of days until an upcoming code freeze stage (2). Our goal is to check whether fixed issues are being prevented from integration mostly because they are being fixed too close to a code freeze stage (i.e., a period during which integration of new code changes would likely be minimal).

In Fig. 12, we show the *fix timing* values for the Eclipse and ArgoUML projects, since both projects adopt a *code freeze* stage. For the Eclipse project, the code freeze starts after the last release candidate, while for the ArgoUML project, the code freeze starts at the beginning of the *beta* stage (see Section 3.3.2). Naturally, we observe a drop in the *fix timing* values, since both code freeze stages start considerably before the official release dates. Nevertheless, we observe that even after correcting for the code freeze stages of the Eclipse and ArgoUML projects, it is unlikely that fixed issues are being prevented from integration solely because of an approaching code freeze stage. For instance, although the median *fix timing* for the ArgoUML project dropped from 0.52 to 0.35, the development team would still have 2 months to integrate a fixed issue—since the median duration of a release cycle in the ArgoUML project is 180 days.



**Fig. 12** Fix timing values for the code freeze period. The median fix timing values drop from 0.45 and 0.52 to 0.41 and 0.35 in the Eclipse and ArgoUML projects, respectively



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

### 4.3 RQ3: How Well Can We Model the Integration Time of Fixed Issues?

#### 4.3.1 RQ3: Results for Integration Time in Terms of Releases

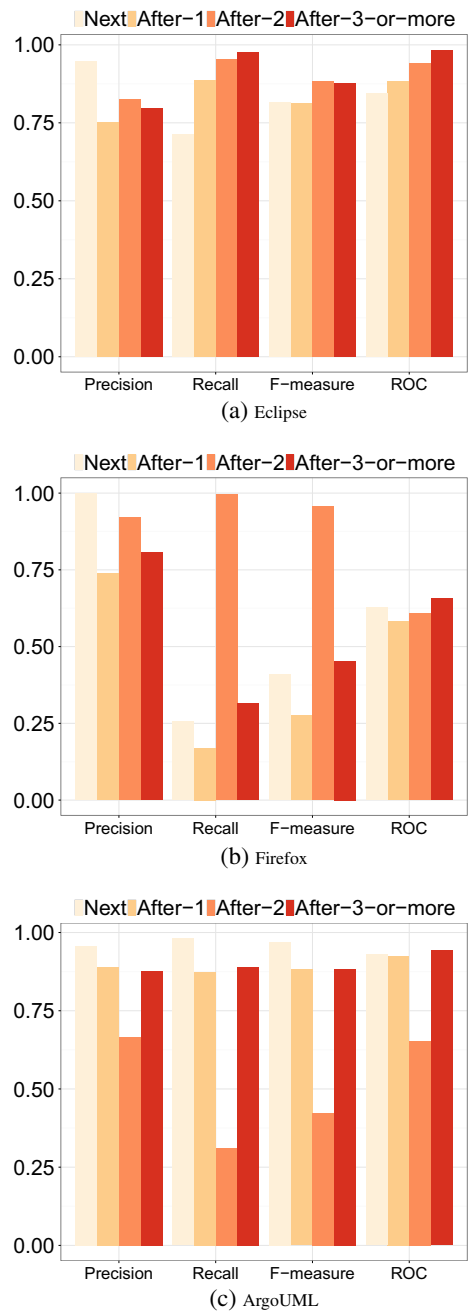
**Our Explanatory Models Obtain a Median Precision of 0.81 to 0.88 and a Median Recall of 0.29 to 0.92** Figure 13 shows the precision, recall, F-measure, and AUC of our explanatory models. The bar charts show the values that we observe for each bucket. The values of precision, recall, F-measure, and AUC are also shown in Table 6.

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

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

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

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



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

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

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

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

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

#### 4.3.2 RQ3: Results for Integration Time in Terms of Days

**Our Explanatory Models Obtain  $R^2$  Values of 0.39–0.65 and MAE Values Between 7.8 to 67 Days** Our models obtain fair  $R^2$  values to model the variability of integration time in days in the studied projects. Table 7 shows the  $R^2$  and MAE values that are obtained by each of our regression models. The  $R^2$  values for the Eclipse, Firefox, and ArgoUML projects are of 0.39, 0.48, and 0.65, respectively. Additionally, our regression models can

**Table 7** Regression results of model fit

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

Our explanatory models obtain  $R^2$  values between 0.39 to 0.65 and MAE values between 7.8 to 66 days

provide fair estimations of integration time in days, specially for the Firefox project. For instance, the median interval in days between releases of the Firefox project is 42 days (see Fig. 9), while the MAE value for the Firefox project is 7.8 days, which equates to an error ratio of 18% (see Table 7).

**Our Explanatory Models Obtain a Good Stability with Bootstrap Calculated Optimism Between 0.0035 to 0.0267 of the  $R^2$  Values** We also observe that our regression models are stable. Table 7 shows the *bootstrap-calculated* optimism of the  $R^2$  values of our models. The optimism for the Eclipse, Firefox and ArgoUML projects are 0.0267, 0.0162, and 0.0035, respectively. Such results indicate that our explanatory models are unlikely to be overfitted to our data and that our models are stable enough for us to perform the statistical inferences that follow.

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

#### 4.4 RQ4: What are the Most Influential Attributes for Modeling Integration Time?

##### 4.4.1 RQ4: Results for Integration Time in Terms of Releases

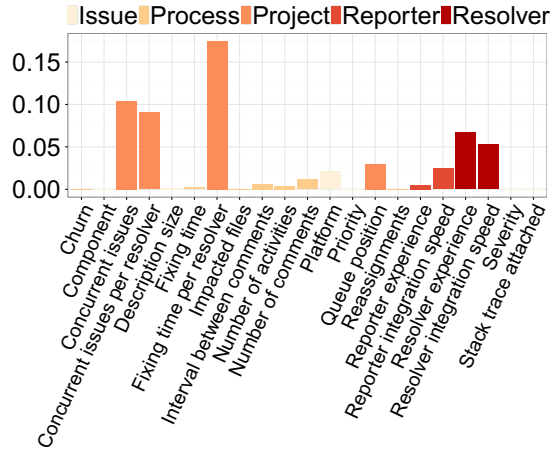
**The Fixing Time per Resolver and Integration Workload Attributes are the Most Influential Attributes in our Models** Figure 14 shows the variable importance values of the LOOCV of our models. The most influential attribute is the *fixing time per resolver*. The *fixing time per resolver* attribute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

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

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

Furthermore, we study the distribution of fixed issues across components in the Firefox project. Figure 15 shows the top seven components of the Firefox project, each having more than 400 fixed issues. We analyze the proportion of fixed issues where integration was prevented in the top seven components. Figure 15 shows that, for buckets *next* and *after-1*, the majority of issues are related to the *General component*, whereas for *after-2* and *after-3-or-more* the majority are related to the *Javascript engine* component. Fixed issues related

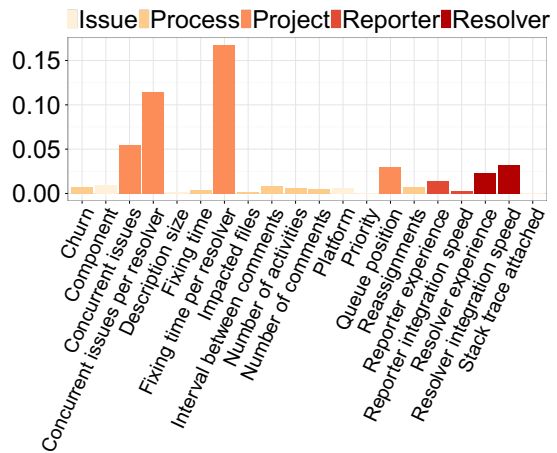
**Fig. 14** Variable importance scores. We show the importance scores that are computed for the LOOCV of our models



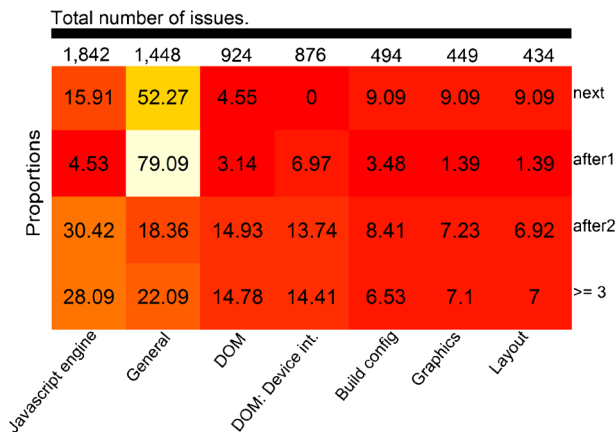
(a) Eclipse



(b) Firefox



(c) ArgoUML



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

to the *General* component may be easy to integrate, whereas issues related to the *Javascript Engine* may require more careful analysis before integration.

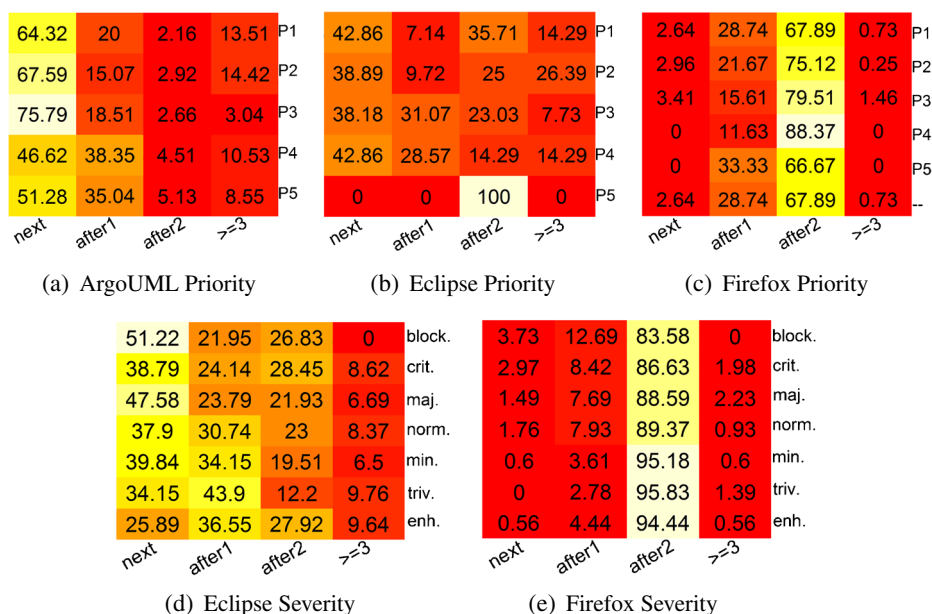
**Severity and Priority have Little Influence on Integration Time in Terms of Releases** Users and contributors of software projects can denote the importance of an issue using the *priority* and *severity* fields. Previous studies have shown that priority and severity have little influence on bug fixing time (Tian et al. 2015; Herraiz et al. 2008; Mockus et al. 2002). For example, while an issue might be severe or of high priority, it might be complex and would take a long time to fix.

However, in the integration context, we expect that priority and severity would be more influential, since the issues have already been fixed. Even though priority and severity are often left at their default values (see Section 3.1), one would expect that the integrators would fast-track the integration of issues for which they care about increasing the levels of severity or priority. For instance, according to the Eclipse project guidelines for filing issue reports, a priority level of P1 is used for serious issues and specifies that the existence of a P1 issue should prevent a release from shipping.<sup>24</sup> Hence, it is surprising that priority and severity play such a small role in determining the release in which a fixed issue will appear. Indeed, Fig. 14 shows that the priority and severity metrics obtain low importance scores.

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

In the ArgoUML project, while the majority of priority P1 issues (64%) were integrated in the *next* release, 36% of them had their integration prevented in at least one release. For the Firefox project, 97% of the P1 issues and 96% of the *blocker* issues were prevented from integration in at least one release. Finally, for the Eclipse project, 57% of P1 issues and 49% of *blocker* issues had their integration prevented in at least one release. Hence, our data shows that, in the context of issue integration, the *priority* and *severity* values that are

<sup>24</sup>[http://wiki.eclipse.org/Development\\_Resources/HOWTO/Bugzilla\\_Use](http://wiki.eclipse.org/Development_Resources/HOWTO/Bugzilla_Use).



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

recorded in the ITSs have little influence on integration time. Instead, fixed issues might be prioritized by the level of risk that are associated to them.<sup>25</sup> This might explain why the time that is invested on fixing issues during a release cycle reduces integration time—a risk of a fixed issue breaking the code would be smaller when more time is invested at fixing activities.

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

#### 4.4.2 RQ4: Results for Integration Time in Terms of Days

**Project Family Attributes, Such as the Backlog of Issues and Queue Position Provide Most of the Explanatory Power of our Models** Table 8 shows the explanatory power of each of the attributes of our models. The two most influential attributes for each model are shown in bold. *Queue position*, i.e., the time at which an issue is fixed is the most

<sup>25</sup>Two issues from our sample were promoted to stabler release channels due to low associated risk [https://bugzilla.mozilla.org/show\\_bug.cgi?id=724145](https://bugzilla.mozilla.org/show_bug.cgi?id=724145) and [https://bugzilla.mozilla.org/show\\_bug.cgi?id=732962](https://bugzilla.mozilla.org/show_bug.cgi?id=732962), while another issue was prevented from integration due to code break [https://bugzilla.mozilla.org/show\\_bug.cgi?id=723793](https://bugzilla.mozilla.org/show_bug.cgi?id=723793).



influential attribute in all of the models that are fitted to our studied projects. Interestingly, we observe that *resolver integration speed*—the median integration time of the previously resolved issues of a particular resolver—plays an influential role in our models that are fit

**Table 8** Explanatory power of attributes

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

**Table 8** (continued)

		Eclipse	Firefox	ArgoUML
Fixing time per resolver	D.F.	1	2	4
	$\chi^2$	7***	$\approx 0^{**}$	8***

We present the  $\chi^2$  proportion and the degrees of freedom that are spent for each attribute. The  $\chi^2$  of the two most influential attributes of each model are in bold

⊙ Discarded during correlation analysis

⊕ Discarded during redundancy analysis

⊖ The variable does not apply to the dataset

\*  $p < 0.05$

\*\*  $p < 0.01$

\*\*\*  $p < 0.001$

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

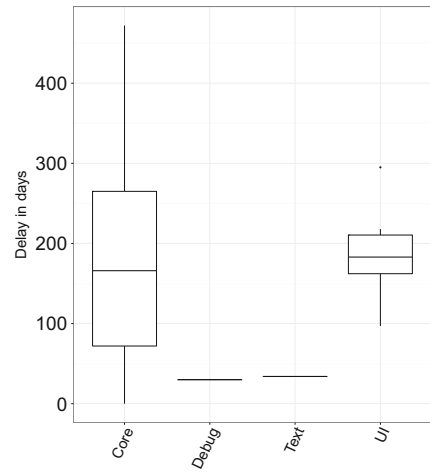
**The Component to Which an Issue is Fixed has Little Impact in the Integration Time in Terms of Days** To demonstrate this, we group each fixed issue according to the components that such an issue modifies. We use components that have at least 100 fixed issues as a threshold for our analysis. We then compare the distribution of integration time in terms of days in these components. Figure 17 shows the distributions of integration time in terms of days per component. We do not observe a considerable difference between distributions of integration time in the ArgoUML or Firefox projects. The distribution of the “Other” component in the ArgoUML project is more skewed, which is suggestive of its generic role—such a component may encompass a more broad spectrum of fixed issues. On the other hand, 99% of the fixed issues in the Eclipse (JDT) project belong to the “Core” component (thus its skewness). Finally, the “Debug” and “Text” Eclipse components contain only one fixed issue each.

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

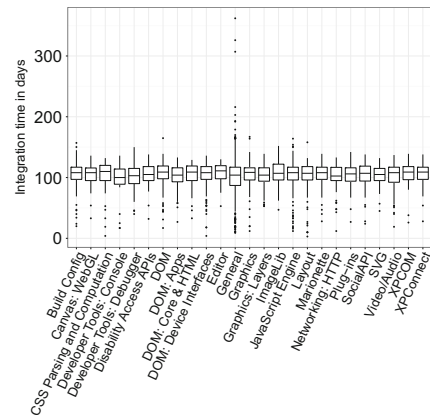
## 5 Results for the Prolonged Integration Time Dimension

In Section 4, we analyze integration time with respect to the number of releases and number of days that a fixed issue requires before integration. Additionally, we study projects with different release cycles. For instance, in the Firefox project, we observed that fixed issues have their integration prevented in two consecutive releases (89% of the fixed issues). However, a long integration time of one project may be shorter than a typical integration time of another project. Hence, we set out to complement our previous analyses by studying fixed issues that suffer a long integration time when compared to other fixed issues of that

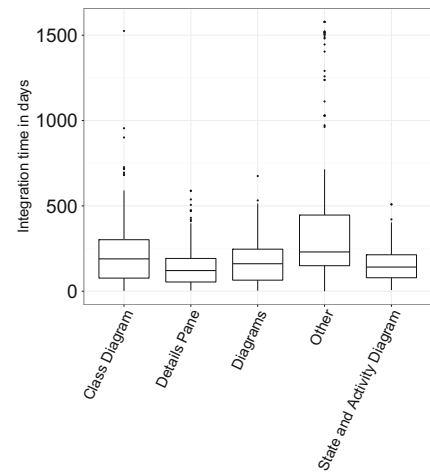
**Fig. 17** Integration time per component. The Figure shows the distributions of integration time in terms of days for each component of the studied projects



(a) Eclipse



(b) Firefox



(c) ArgoUML

particular project. The *prolonged integration time* dimension addresses RQ5 and RQ6. We present the results for each RQ below.

## 5.1 RQ5: How Well Can We Identify the Fixed Issues that Will Suffer from a Long Integration Time?

**Our Models Obtain F-measures from 0.79 to 0.96** Table 9 shows the performance of our exploratory models. Our models that we train for the Eclipse project obtain the highest F-measure (0.96). On the other hand, our models trained for the Firefox and ArgoUML projects obtain F-measures of 0.79 and 0.88, respectively. Moreover, our models obtain AUC values of 0.82 to 0.96. Such results suggest that our models vastly outperform naïve models, such as random guessing (AUC value of 0.50).

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

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

## 5.2 RQ6: What are the Most Influential Attributes for Identifying the Issues that Will Suffer from a Long Integration Time?

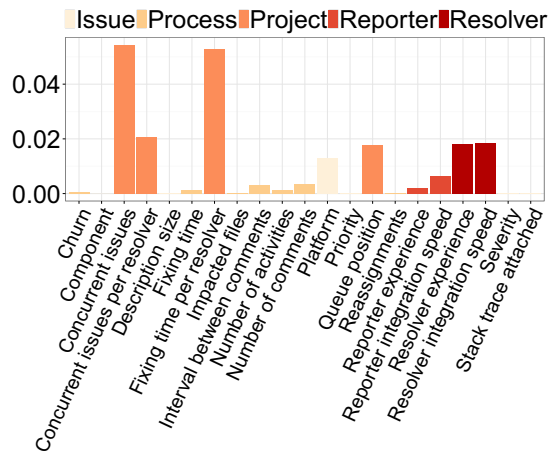
**Long Integration Time is Most Consistently Associated with Attributes of the Project Family** Figure 18 shows the importance scores that are computed for the LOOCV that we use to evaluate our random forest models. We observe that the attributes that are related to the *project* family are the most influential attributes in the projects. The *backlog of issues* is the most influential attribute in our Eclipse models, while *queue position* and *fixing time per resolver* are the most influential attributes in our Firefox and ArgoUML models, respectively. In addition, we observe that attributes that are related to workload, such as the *backlog of issues* and the *backlog of issues per resolver* are at least the third most influential attributes in all of our models. Such results suggest that a *long integration time* is associated with project-related attributes and that the amount of fixed issues that are to be integrated also plays a major role to identify a *long integration time*.

**Table 9** Performance of the random forest models

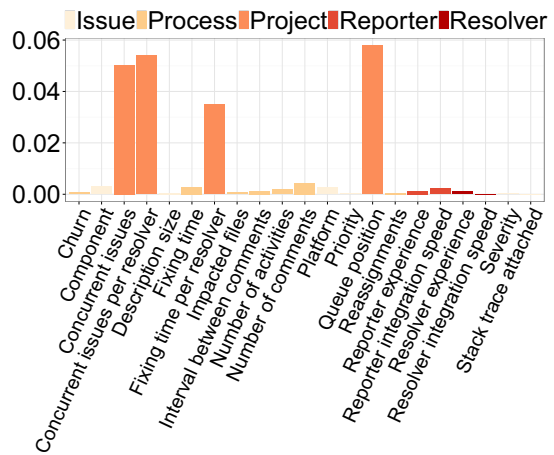
	Eclipse	Firefox	ArgoUML
Precision	0.97	0.99	0.98
Recall	0.96	0.66	0.80
F-measure	0.96	0.79	0.88
AUC	0.96	0.82	0.89

The table shows the values of Precision, Recall, F-measure, and AUC values that are computed for the LOOCV of our models

**Fig. 18** Variable importance scores. We show the importance scores that are computed for the LOOCV of our models



(a) Eclipse



(b) Firefox



(c) ArgoUML

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

## 6 Discussion

### The Most Important Attributes Vary as We Study Different Kinds of Integration Time

While we observe that *fixing time per resolver* is the most influential attribute to model integration time in terms of releases, the time at which an issue is fixed (*queue position*) is the most influential attribute to model integration time in days. This difference may be explained by the different focus of these kinds of integration time. The integration time in terms of releases focuses on the releases from which the integration of fixed issues is prevented. In this context, the *fixing time per resolver* attribute becomes influential, because it is a measure of the amount of time that was invested by the team to fix issues, which may lead to smoother integration of an issue in the upcoming releases. This smoother integration might be either because issues were fixed more carefully or because complex/risky issues had the necessary time to become stable enough to avoid breakage.

Integration time in terms of days focuses on the total time that is required to ship a fixed issue regardless the number of releases that are missed. In this case, the time at which an issue is fixed in the release cycle becomes more influential (i.e., *queue position*). For example, a fixed issue might be shipped faster because it was fixed during a *beta stage* (see RQ2), i.e., when the collaborators deal with a narrower *backlog of issues* so that fixes can be performed more carefully. The increased focus due to a narrower *backlog of issues* may lead the fixed issue to become easier to integrate in the next release cycle.

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

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

### The Backlog of Fixed Issues Awaiting Integration May Introduce an Overhead that Must be Managed by Software Teams

We observe that integration workload attributes (e.g., *backlog of issues* and *backlog of issues per resolver*) are influential in all studied kinds of integration time. This finding suggests that the overhead that is introduced by the backlog of fixed issues that are awaiting integration may increase the integration time as a whole.

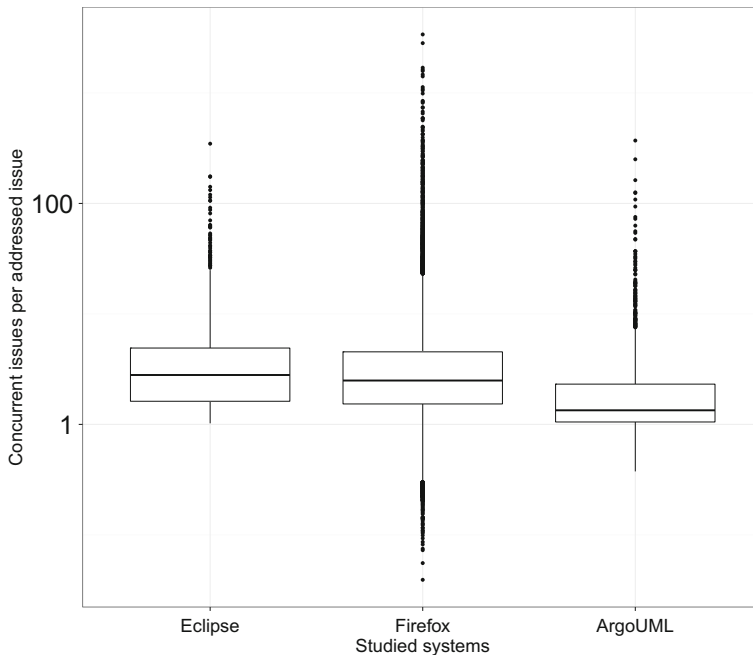
## 7 Exploratory Data Analysis

### 7.1 Backlog of Issues per Fixed Issue

We observe that the integration workload in terms of the number of backlog of issues is an influential attribute in all of the studied projects. Because of this observation, we also investigate the competition that is due to issues that are waiting for integration per fixed issue in the release cycle. Figure 19 shows the distributions of the number of competing issues for a fixed issue of a given release cycle. For each fixed issue, a median of three, two, and one other issues are competing for integration in the Eclipse, Firefox, and ArgoUML projects, respectively. The distribution of the Firefox project is equivalent to the Eclipse project one, even though the Firefox releases are more frequent. This observation might suggest an intense period of activity in the Firefox release cycles (high rates of integration and fixing activity).

### 7.2 Practical Suggestions

In our study, we observe that attributes such as: *fixing time per resolver*, *backlog of issues*, *resolver integration speed*, and *queue position* have a considerable impact on the studied kinds of integration time. As such, we suggest that our investigated attributes could be used as a starting point in project management tools to track the integration time of fixed issues. For example, a tool that could automatically track the *backlog of issues* by using the ITS, could raise warnings when the backlog for integration crosses a project-specific threshold.



**Fig. 19** Backlog of issues per fixed issue of the current release cycle. The median number of concurrent fixes per fixed issue for the Eclipse, Firefox, and ArgoUML projects are 3, 2, and 1, respectively

Such a warning could lead to early integration sessions before the official release deadline, and prevent log jams in the integration queue.

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

## 8 Threats to Validity

### 8.1 Construct Validity

A number of tools were developed to extract and analyze the integration data in the studied projects. Defects in these tools could have an influence on our results. However, we carefully tested our tools using manually-curated subsamples of the studied projects, which produced correct results.

### 8.2 Internal Validity

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

The main threat in this regard is the representativeness of the data. Although the Firefox and Eclipse projects report the list of fixed issues in their release notes, we do not know how complete this list truly is. In addition, issues may be incorrectly listed in a release note. For example, an issue that should have been listed in the release notes for version 2.0 but only appears in the release note for version 3.0. Such human errors may introduce noise in our datasets. To explore how correct the release notes are, we draw a random sample of 120 Firefox fixed issues, each one listed in the release notes of versions 17 to 27. We verify the corresponding *tag* that such issues were integrated into in the BETA channel, i.e., the most stable channel of the Firefox project that lead to the RELEASE channel.<sup>26</sup> Indeed, 94% ( $\frac{113}{120}$ ) fixed issues were integrated into the corresponding tag that lead to the release for which the release notes have listed such issues. This sample can be found on the supplemental material Web page.<sup>27</sup>

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

In addition, the way that we segment the response variable of our explanatory models is also subject to bias. For the integration time in terms of releases (Definition 1), we segment the response variable into *next*, *after-1*, *after-2*, and *after-3-or-more*. Although we found it to be a reasonable classification, a different classification may yield different results. Also, we use at least one MAD above the median as a threshold to split the response variable of the prolonged integration time (Definition 3) into two categories. A different threshold to split the response variable may yield different results.

<sup>26</sup><https://hg.mozilla.org/releases/mozilla-beta/tags>.

<sup>27</sup>[http://sailhome.cs.queensu.ca/replication/integration\\_delay/](http://sailhome.cs.queensu.ca/replication/integration_delay/).



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

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

### 8.3 External Validity

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

## 9 Related Work

Estimating the effort and time required to fix an issue has become an important project planning activity. To assist developers and project managers in this regard, several studies have proposed different approaches to estimate effort and time required to triage and to fix issues (Anvik et al. 2005; Hooimeijer and Weimer 2007; Anbalagan and Vouk 2009; Giger et al. 2010; Kim and Whitehead 2006; Marks et al. 2011; Weiß et al. 2007; Zhang et al. 2013). In each of the following subsections, we describe prior work about triaging, fixing, and integrating issues.

### 9.1 Our Prior Work

This paper is an extended version of our prior work (Costa et al. 2014). We extend our prior work to:

1. Expand the set of explanatory attributes (Tables 2 and 3) to include:
  - (a) The time to fix an issue, i.e., from OPENED to RESOLVED-FIXED.
  - (b) An indicator of whether or not a stack trace has been attached to the issue report.
  - (c) The time at which an issue is fixed during the current release cycle (*queue position*).
  - (d) Attributes that are related to the resolvers of issues (*resolver experience* and *resolver integration speed*).
  - (e) Heuristics that estimate the effort that was invested in fixing the issues of a given release cycle (*fixing time per resolver*).

- (f) The number of fixed issues that are waiting to be integrated, normalized by the size of the team (*backlog of issues per resolver*)
2. Study fixed issues that suffer from a long integration time (RQ5 and RQ6 in Section 5).
3. Study integration time in terms of days (RQ3 and RQ4 in Section 4).
4. Study the relationship between the components of the studied systems and integration time (RQ4 in Section 4).
5. Perform an exploratory analysis of how release cycle stages are related with integration time (RQ2).
6. Analyze the backlog of issues that are waiting to be integrated, normalized by the number of issues that were fixed in a given release cycle (Section 7).
7. Outline key insights for practitioners based on our findings (Section 7).

## 9.2 Triageing Issues

Triageing issues is the process of deciding which issues have to be fixed, and assigning the appropriate developer to them (Anvik et al. 2006). This decision depends of several factors, such as the impact of the issue on the software, or how much effort is required to fix the issue. Projects usually receive a high number of issue reports. Issue reports come from a diverse audience that is usually larger than the developer team. Hence, effective triageing of issue reports is an important means of keeping up with user demands.

Hooimeijer and Weimer (2007) built a model to classify whether or not an issue report will be “cheap” or “expensive” to triage by measuring the quality of the report. Based on their findings, the authors state that the effort required to maintain a software system could be reduced by filtering out reports that are “expensive” to triage. Saha et al. (2014) studied long lived issues, i.e., issues that were not fixed for more than one year. They found that the time to assign a developer and fix such issues is approximately two years. Our work complements these prior studies by investigating the time to integrate issues once they are fixed rather than the time to assign a developer to fix the issue.

## 9.3 Fixing Issues

Once an issue is properly triaged, the assigned developer starts to fix it. Kim and Whitehead (2006) computed the needed time to fix issues in the ArgoUML and PostgreSQL projects. They found that the issue-fixing time for their studied projects ranges from 100 to 200 days. To estimate the required time to fix issues, some approaches use the similarity of an issue to existing issues (Weiß et al. 2007; Zhang et al. 2013). For example, Weiß et al. (2007) used the title and description of a newly reported issue to find similar issues on the Jira ITS of the JBoss project. The authors used the Apache Lucene framework to find the similar issues. The effort that is needed to fix a newly reported issue is computed based on the average fix time of its similar issues. Other approaches train prediction models using different machine learning techniques (Panjer 2007; Anbalagan and Vouk 2009; Giger et al. 2010;

Guo et al. (2010; Marks et al. 2011) to estimate the needed time to fix an issue. For example, Panjer (2007) used Decision Trees, Logistic Regression models, and the Naïve Bayes algorithm to estimate the lifetime of issues, i.e., the total time between the date of the *NEW* status and the date of the *RESOLVED-FIXED* status. Moreover, Guo et al. (2010) used logistic regression models to predict the probability that a new issue will be fixed. The authors trained the model on Windows Vista issues and obtained a precision of 0.68 and recall of 0.64 when tested on Windows 7 issue reports. These approaches focus on estimating the time that is required to fix an issue.

Other recent empirical studies investigate the significance of attributes that are used to estimate the needed time to fix issues. Bhattacharya and Neamtiu (2011) performed univariate and multivariate regression analyses to capture the significance of four attributes in issue reports. Their results indicate that more attributes are required to train better prediction models. Herraiz et al. (2008) studied the impact of issues' severity and priority levels on the required time to fix issues. The authors used one way analysis of variance to study significant differences among the priority and severity levels that are assigned to the issues of the Eclipse project. Based on their results, the authors suggest to reduce the severity and priority levels to three options only. Zhang et al. (2012) investigated the delays that are incurred by developers in the process of fixing issues. To do so, they extracted the initial and final dates of issues fixing activity from interaction logs. The authors used the collected information to analyze delays in the fixing process. Three dimensions related to issues were investigated: characteristics of issue reports, source code, and code changes. They found that attributes such as severity, operating system, issue description, and comments are likely to impact the delays in the process of fixing issues. Similar to Zhang et al. (2012), we use attributes that are related to issue reports to train explanatory models. However, we aim to understand which attributes are influential in the integration time of fixed issues rather than the needed time to fix issues. In addition, we investigate why severity and priority levels are not relevant to distinguish issue reports that are fixed and integrated more quickly compared to others. Our work advances the state of the art by studying how much time a fixed issue may require before reaching end users.

## 9.4 Integrating Issues

Jiang et al. (2013) studied attributes related to patch integration decisions in the Linux kernel. A patch is a record of changes that is applied to a software system to fix an issue. To identify such attributes, the authors trained decision tree models and conducted top node analysis. Among the attributes that are studied, developer experience, patch maturity, and prior subsystem are found to play an influential role in patch acceptance and integration time. Similar to Jiang et al. (2013), we also investigate the integration of fixed issues. However, we focus on several kinds of integration time rather than integration decisions.

Choetkiertikul et al. (2017) and Morakot et al. (2015) studied the risk of delaying the resolution of issues. For example, if an issue is resolved after its assigned due date, such an issue is considered to be delayed. Choetkiertikul et al. (2017) achieved a predictive performance of 79% precision, 61% recall, 72% F-measure, and AUCs above 80% using their predictive models. In addition, Morakot et al. (2015) studied the risk of delaying the resolution of a particular issue by analyzing the relationship that such an issue has with other issues. For example, if issue *A* blocks issue *B* and issue *A* was delayed, what is the risk of issue *B* to be delayed as well? The authors achieved predictive performance of 46%–97%

precision, 46%–97% recall, and AUCs of 78%–95%. Our work complements the aforementioned work by studying the needed time to deliver fixed issues to end users rather than studying the risk of delaying the resolution of a particular issue.

## 10 Conclusions

Once an issue is fixed, what users and code contributors care most about is when the software is going to reflect such a fixed issue, i.e., when the integration occurs. However, we observed that the integration of several fixed issues was delayed for a considerable amount of time. It was not clear why certain fixed issues take longer to be integrated than others. We performed an empirical study of 20,995 issues from the ArgoUML, Eclipse and Firefox projects. In our study, we:

- find that despite being fixed well before an upcoming release, 34% to 60% of the fixed issues were prevented from integration in more than one release in the ArgoUML and Eclipse projects. Furthermore, 98% of the Firefox project issues had their integration prevented in at least one release.
- train random forest models to model the integration time of a fixed issue. Our models obtain a median AUC values between 0.62 to 0.96. Our models outperform baseline random and Zero-R models.
- compute variable importance to understand which attributes are the most important in our random forest models to study integration time. Heuristics that estimate the effort that teams invest in fixing issues are the most influential in our models to study integration time in terms of number of releases.
- find that *priority* and *severity* have little impact on our exploratory models for integration time. Indeed, 36% to 97% of priority P1 fixed issues were prevented from integration in at least one release.
- find that a shorter integration time is associated with fixes that are performed during more controlled stages of a given release cycle.
- observe that the time at which issues are fixed and the resolvers of the issues have great impact on estimating the integration time of a fixed issue. Our explanatory models obtain  $R^2$  values between 0.39 to 0.65.
- verify that our models that identify fixed issues that have a long integration time outperform random guessing and Zero-R models, obtaining AUC values of 0.82 to 0.96.
- find that the time at which an issue is fixed (queue position), the integration workload (in terms of the backlog of fixed issues), and the heuristics that estimate the effort that teams invest in fixing issues (fixing time per resolver), are the most influential attributes for issues that have a long integration time.

Our work provides insights as to why some fixed issues are integrated prior to others. Our results suggest that characteristics of the release cycle are the ones that have the largest impact on integration time. Therefore, our findings highlight the importance of research and tooling that can support integrators of software projects. It is important to improve the integration stage of a release cycle, because the availability of a fixed issue in a release is what users and contributors care most about.

As part of our future work, we intend to perform a qualitative study (e.g., surveys and interviews) using our subject projects to better understand why a longer integration time occurs during the development of a software project to explain our findings in terms of developers' behavior.

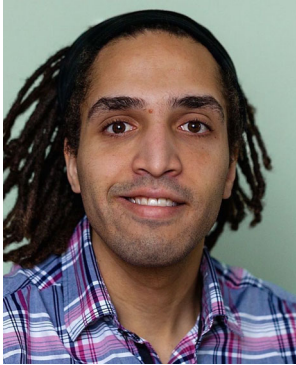
## References

- Anbalagan P, Vouk M (2009) On predicting the time taken to correct bug reports in open source projects. In: Proceedings of the 2009 IEEE international conference on software maintenance, ICSM '09, pp 523–526
- Anvik J, Hiew L, Murphy GC (2005) Coping with an open bug repository. In: Proceedings of the 2005 OOPSLA workshop on eclipse technology eXchange, eclipse '05, pp 35–39
- Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proceedings of the 28th international conference on software engineering, ICSE '06, pp 361–370
- Bersani FS, Lindqvist D, Mellon SH, Epel ES, Yehuda R, Flory J, Henn-Hasse C, Bierer LM, Makotkine I, Abu-Amara D et al (2016) Association of dimensional psychological health measures with telomere length in male war veterans. *J Affect Disord* 190:537–542
- Bhattacharya P, Neamtiu I (2011) Bug-fix time prediction models: can we do better? In: Proceedings of the 8th working conference on mining software repositories, MSR '11, pp 207–210
- Breiman L (2001) Random forests. In: *Machine Learning*, Springer Journal no. 10994, pp 5–32
- Brooks FP (1975) The mythical man-month, vol 1995. Addison-Wesley, Reading
- Choetkiertikul M, Dam HK, Tran T, Ghose A (2017) Predicting the delay of issues with due dates in software projects. *Empir Softw Eng J* 23:1–41
- Choi H, Varian H (2012) Predicting the present with google trends. *Econ Rec* 88(s1):2–9
- Cliff N (1993) Dominance statistics: ordinal analyses to answer ordinal questions. *Psychol Bull* 114:494–509
- Costa DAD, Abebe SL, McIntosh S, Kulesza U, Hassan AE (2014) An empirical study of delays in the integration of addressed issues. In: 2014 IEEE international conference on software maintenance and evolution (ICSME), pp 281–290
- Dunn OJ (1961) Multiple comparisons among means. *J Am Stat Assoc* 56(293):52–64
- Dunn OJ (1964) Multiple comparisons using rank sums. *Technometrics* 6(3):241–252
- Efron B (1986) How biased is the apparent error rate of a prediction rule? *J Am Stat Assoc* 81(394):461–470
- Freedman DA (2009) Statistical models: theory and practice. Cambridge University Press, Cambridge
- Giger E, Pinzger M, Gall H (2010) Predicting the fix time of bugs. In: Proceedings of the 2nd international workshop on recommendation systems for software engineering, RSSE '10, pp 52–56
- Guo PJ, Zimmermann T, Nagappan N, Murphy B (2010) Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In: Proceedings of the 32nd ACM/IEEE international conference on software engineering—volume 1, ICSE '10, pp 495–504
- Hanley JA, McNeil BJ (1982) The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology* 143(1):29–36
- Harrell FE (2001) Regression modeling strategies: with applications to linear models, logistic regression, and survival analysis, Springer, Berlin
- Herraiz I, German DM, Gonzalez-Barahona JM, Robles G (2008) Towards a simplification of the bug report form in eclipse. In: Proceedings of the 2008 international working conference on mining software repositories, MSR '08, pp 145–148
- Hooimeijer P, Weimer W (2007) Modeling bug report quality. In: Proceedings of the twenty-second IEEE/ACM international conference on automated software engineering, ASE '07, pp 34–43
- Howell DC (2005) Median absolute deviation. In: *Encyclopedia of statistics in behavioral science*
- Jeong G, Kim S, Zimmermann T (2009) Improving bug triage with bug tossing graphs. In: Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp 111–120
- Jiang Y, Adams B, German DM (2013) Will my patch make it? And how fast?: Case study on the linux kernel. In: Proceedings of the 10th working conference on mining software repositories, MSR '13, pp 101–110
- Kim S, Whitehead EJ Jr (2006) How long did it take to fix bugs? In: Proceedings of the 2006 international workshop on mining software repositories, MSR '06, pp 173–174
- Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. *J Am Stat Assoc* 47(260):583–621
- Leys C, Ley C, Klein O, Bernard P, Licata L (2013) Detecting outliers: do not use standard deviation around the mean, use absolute deviation around the median. *Exp Social Psychol J* 49:764–766
- Mantyla MV, Khomh F, Adams B, Engstrom E, Petersen K (2013) On rapid releases and software testing. In: 2013 29th IEEE international conference on software maintenance (ICSM), pp 20–29

- Marks L, Zou Y, Hassan AE (2011) Studying the fix-time for bugs in large open source projects. In: Proceedings of the 7th international conference on predictive models in software engineering, Promise '11, pp 11:1–11:8
- Mockus A, Fielding RT, Herbsleb JD (2002) Two case studies of open source software development: Apache and mozilla. *ACM Trans Softw Eng Methodol* 11(3):309–346
- Morakot C, Hoa Khanh D, Truyen T, Aditya G (2015) Predicting delays in software projects using networked classification. In: 30th international conference on automated software engineering (ASE)
- Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: 27th international conference on software engineering, 2005. ICSE 2005. Proceedings, pp 284–292
- Olkin GCSFI (2002) Springer texts in statistics
- Panjer LD (2007) Predicting eclipse bug lifetimes. In: Proceedings of the fourth international workshop on mining software repositories, MSR '07, p 29
- Rahman MT, Rigby PC (2015) Release stabilization on linux and chrome. *IEEE Softw* 32(2):81–88
- Romano J, Kromrey JD, Coraggio J, Skowronek J (2006) Should we really be using t-test and cohen's d for evaluating group differences on the nsse and other surveys? In: Annual meeting of the Florida association of institutional research
- Saha R, Khurshid S, Perry D (2014) An empirical study of long lived bugs. In: 2014 software evolution week—IEEE conference on software maintenance, reengineering and reverse engineering (CSMR-WCRE), pp 144–153
- Sarle W (1990) The varclus procedure. SAS/STAT User's Guide
- Schroter A, Bettenburg N, Premraj R (2010) Do stack traces help developers fix bugs? In: 2010 7th IEEE working conference on mining software repositories (MSR), pp 118–121
- Singer J (1999) Using the american psychological association (apa) style guidelines to report experimental results. In: Proceedings of workshop on empirical studies in software maintenance, pp 71–75
- Steel RG, James H (1960) Principles and procedures of statistics: with special reference to the biological sciences. Tech. rep. McGraw-Hill, New York
- Tian Y, Ali N, Lo D, Hassan AE (2015) On the unreliability of bug severity data. *Empir Softw Eng* 21:1–26
- Weiß C., Premraj R., Zimmermann T., Zeller A. (2007) How long will it take to fix this bug? In: Proceedings of the fourth international workshop on mining software repositories, MSR '07, p 1
- Zhang F, Khomh F, Zou Y, Hassan A (2012) An empirical study on factors impacting bug fixing time. In: 2012 19th working conference on reverse engineering (WCRE), pp 225–234
- Zhang H, Gong L, Versteeg S (2013) Predicting bug-fixing time: an empirical study of commercial software projects. In: Proceedings of the 2013 international conference on software engineering, ICSE '13, pp 1042–1051



**Daniel Alencar da Costa** received the bachelor's degree from the University Centre of The Pará State (CESUPA) and the MSc and PhD degrees from the Federal University of Rio Grande do Norte (UFRN), Brazil. In his research, he has studied the delay that happens during the delivery of new software functionalities to end users and the efficiency of debugging tools. His interests also include general mining software repositories and empirical software engineering research. More information at <http://danielcalencar.github.io>.



**Shane McIntosh** received the bachelor's degree from the University of Guelph and the MSc and PhD degrees from Queen's University. He is an assistant professor in the Department of Electrical and Computer Engineering, McGill University, where he leads the Software Repository Excavation and Build Engineering Labs (Software REBELs). During his PhD, he held an NSERC Vanier Scholarship and received the Governor General's Academic Gold Medal. In his research, he uses empirical software engineering techniques to study software build systems, release engineering, and software quality. He actively collaborates with academics in Canada, the Netherlands, Singapore, Brazil, and Japan, as well as industrial practitioners in Germany and the USA. More about him and his work is available online at <http://rebels.ece.mcgill.ca/>.



**Uirá Kulesza** received the bachelor's degree in computer science from Federal University of Campina Grande, the MSc degree in computer science from the University of São Paulo, and the PhD degree in computer science from Pontifical Catholic University of Rio de Janeiro (PUC-Rio). He is an associate professor in the Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte (UFRN), Brazil. His main research interests include software evolution, software architecture, and software analytics. He has published at several top-tier software engineering venues, such as the International Conference on Software Engineering (ICSE), the International Symposium on the Foundations of Software Engineering (FSE), and the European Conference on Object-Oriented Programming (ECOOP). More about him and his work is available online at <http://www.dimap.ufrn.br/~uira>.





**Ahmed E. Hassan** received the PhD degree in computer science from the University of Waterloo. He is the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair in the School of Computing, Queens University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. He also serves on the editorial boards of the IEEE Transactions on Software Engineering, the Springer Journal of Empirical Software Engineering, the Springer Journal of Computing, and Peer Journal Computer Science. Contact [ahmed@cs.queensu.ca](mailto:ahmed@cs.queensu.ca). More information at: <http://sail.cs.queensu.ca/>.



**Surafel Lemma Abebe** is currently an Assistant Professor in Addis Ababa Institute of Technology (AAiT), Addis Ababa University (AAU), Ethiopia. He was also a post-doctoral fellow at the Software Analysis and Intelligence Lab (SAIL) in the School of Computing at Queens University, Canada. He received his B.Sc. and M.Sc. degrees in Computer Science from AAU in 2003 and 2005, respectively, and a PhD degree from University of Trento in 2013. Dr. Abebe conducted his Ph.D. in the Software Engineering Unit of the Bruno Kessler Foundation (FBK), Italy. His current research interests are ICT for development, program comprehension, software evolution, and source code and source code artifact analysis.