Effectiveness of Code Contribution: From Patch-Based to Pull-Request-Based Tools

Jiaxin Zhu, Minghui Zhou*
Institute of Software, EECS, Peking University
Key Laboratory of High Confidence Software Technologies, Ministry of Education
Beijing 100871, China
{zhujiaxin, zhmh}@pku.edu.cn

Audris Mockus
University of Tennesee
1520 Middle Drive Knoxville, TN 37996-2250, USA
audris@utk.edu

ABSTRACT

Code contributions in Free/Libre and Open Source Software projects are controlled to maintain high-quality of software. Alternatives to patch-based code contribution tools such as mailing lists and issue trackers have been developed with the pull request systems being the most visible and widely available on GitHub. Is the code contribution process more effective with pull request systems? To answer that, we quantify the effectiveness via the rates contributions are accepted and ignored, via the time until the first response and final resolution and via the numbers of contributions. To control for the latent variables, our study includes a project that migrated from an issue tracker to the GitHub pull request system and a comparison between projects using mailing lists and pull request systems. Our results show pull request systems to be associated with reduced review times and larger numbers of contributions. However, not all the comparisons indicate substantially better accept or ignore rates in pull request systems. These variations may be most simply explained by the differences in contribution practices the projects employ and may be less affected by the type of tool. Our results clarify the importance of understanding the role of tools in effective management of the broad network of potential contributors and may lead to strategies and practices making the code contribution more satisfying and efficient from both contributors' and maintainers' perspectives.

CCS Concepts

•**Software and its engineering** → **Software maintenance tools; Collaboration in software development;**

Keywords

Code contribution, effectiveness, pull request, issue tracker, mailing list, FLOSS

*Corresponding author.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...$15.00
http://dx.doi.org/10.1145/2950290.2950364

1. INTRODUCTION

In Free/Libre and Open Source Software (FLOSS) projects, non-core contributors have to go through the steps of code contribution process [29] from code creation to review and acceptance. These steps are essential to ensure the quality of contributions because of the diverse nature and skills of FLOSS participants. Mailing lists and issue trackers, where code contributions are submitted by patches, have been widely employed to support code contributions. The participation in and efficiency of the code contribution process from the perspective of code reviews have been extensively studied (see, e.g., [3, 27, 25, 5, 21]). These studies indicate that a bulk of submissions do not receive a response and, therefore, do not involve a code review and are not considered to be within the scope of these studies. The ignored submissions, however, are a critical part of the contribution practice as they require participants to create, submit, and document their code. Such "wasted" effort may detract submitters from further contributions and deter them from contributing altogether. The core members, at the same time, may miss important and valuable contributions. It is, therefore, imperative to understand the nature of code contribution efficiency at every stage in order to increase the overall effectiveness of code contribution practice.

In the past decade, the code contribution tools based on pull requests, e.g., pull request systems provided by GitHub and Bitbucket, have attracted wide attention [12]. In these tools, code contributions are submitted, reviewed and integrated through pull requests. Recent studies of pull requests have started to look at some aspects of the possible inefficiencies by investigating what makes it likely for a code submission to get accepted [12], what the challenges faced by and working practices of integrators and contributors are [14, 13]. However, the role of tools in the overall effectiveness of code contribution practice and the relative amount of wasted effort have not been scrutinized in the research literature.

In this study we attempt to understand a complete picture from the perspective of a participant submitting code contribution to getting it accepted. More specifically, we aim to answer the following research question: **RQ0: what are the differences of code contribution effectiveness between pull-request-based and patch-based tools**? We gather information from published literature and retrieve development data for four GitHub projects to quantify the overall effectiveness of code contribution using different tool-

s. Following the literature, we measure the contribution effectiveness via the code accept and ignore rates, the times until first response and final resolution, and the numbers of contributions adjusted for codebase size. We start with the Ruby on Rails (Rails), a high profile project on GitHub which migrated from an issue tracker to the GitHub pull requests. We investigate whether the migration of tools within Rails is associated with a change of contribution efficiency. We continue the study by comparing eight projects using mailing list in the literature with four GitHub projects using pull requests. Overall we find that the pull requests are more efficient in terms of making submissions processed faster, and are associated with larger numbers of contributions. However, the within- and cross-project comparison lead to contradictory findings for accept and ignore rates. As discussed in Section 5, the contradiction may be explained away if we assume that the accept and ignore rates are primarily determined by the project contribution practices and may be less affected by the type of tool used.

This understanding could help improve the code contribution process from both contributors' and maintainers' perspective. First, projects that do not have good review practices like SVN or Linux kernel (both are still using mailing lists) may consider a move to pull-request-like tools. Second, practitioners or tool designers may consider adding features associated with higher effectiveness to their code submission tools. In particular, providing information on which version a submission is based on and transferring the base to the newest could prevent conflicts when the reviewers are experimentally merging the commits to verify their correctness. The communication mechanism similar to GitHub review board that keeps the discussions related to a contribution in single track, along with the additional features such as view and notification, are likely to help participants focus their attention on the nature of the change. It's also worth to note that the utilization of social features presented by GitHub may help attract and sustain participation.

The rest of the paper is organized as following. We introduce background and review related work in Section 2 and the methodology in Section 3, and Section 4 reports the results. We discuss the limitations of this work in Section 6 and conclude in Section 7.

2. BACKGROUND AND RELATED WORK

In software development, usually a shared central version control repository is set to manage the code and an experienced group of developers control write access to this repository. To keep high quality of the codebase, the code contribution practices tend to adhere to a protocol as illustrated in Figure 1: a contributor makes and submits code; before any changes are applied to the shared repository (i.e., committed) the code must receive positive reviews and be approved by the project maintainers. This review-then-commit (RTC) approach is the most common practice of code contribution in FLOSS projects. In this study, we only consider contributions using RTC channel. We introduce the evolution of the supporting tools in Section 2.1, and review related studies in Section 2.2.

2.1 Evolution of Code Contribution Tools

The code contribution tools have evolved from traditional patch-based tools, such as mailing lists (e.g., Mailman) and issue trackers (e.g., Bugzilla) to modern pull-request-based tools (e.g., GitHub pull request system). A code contribution is a patch attached to an email or an issue report or a set of commits encapsulated in a pull request.

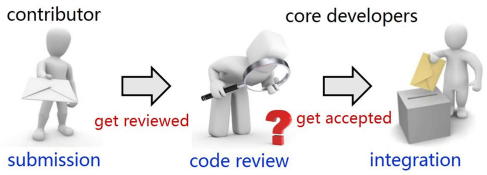


Figure 1: Code contribution process.

2.1.1 Patch-Based Tools

The mailing list and issue tracker, which were originally designed for communication and task management respectively, were the first introduced by FLOSS projects to manage code contributions. For example, Linux kernel uses its mailing lists [25] and Mozilla uses its issue tracker [24]. In mailing lists, the process begins with an author creating a patch (a change of the codebase) broadcasting to the potentially interested individuals via an email, while, in issue trackers, the patches are published with issues. After submission, the patch may receive no response (be ignored), or it may be reviewed with feedback sent to the contributor through emails in mailing lists or comments in issue trackers. The contributor and other developers revise and discuss the patch until it is ultimately accepted or rejected [27, 24].

2.1.2 Pull-Request-Based Tools

After 2008, the pull request mechanism for distributed version control system (DVCS) was introduced on collaboration platforms, e.g., GitHub[1], Bitbucket[2], Gitlab[3], etc. At the time of this study, many GitHub FLOSS projects are using pull request systems for code contribution, e.g., Rails, jQuery, etc. In a pull request system, users *fork*, which is know as *clone* in DVCS, the central repository of a project. That allows users to experiment on their own copy of the repository without affecting the original project (i.e., the central repository). When a set of changes are ready to be submitted to the central repository, they submit a pull request, which specifies a local branch with the changes to be merged with a branch in the central repository. Project's core team members are responsible for reviewing the changes and merging them to the project's branches.

2.2 Related Studies

There are many studies investigating the code contribution practice from various aspects, including the participation, quality assurance, review interval and contribution acceptance.

Participation. Asundi et al. [1] conducted a case study on five FLOSS projects that either use mailing lists or issue trackers for code contribution. The study compared the extent of code inspection participation of core developers among the projects and found it varying among projects. Nurolahzade et al. [24] studied patch review on Bugzilla of Mozilla Firefox and highlighted that peer developers play

[1]https://github.com/
[2]https://bitbucket.org/
[3]https://about.gitlab.com/

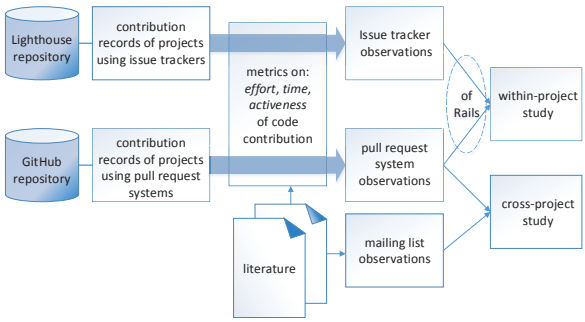


Figure 2: Study design.

supporting role by offering insights and ideas that help create more quality patches. Rigby et al. [27] investigated the mechanisms and behaviours that developers use to find code changes they are competent to review and explored how stakeholders interact with one another during the review process.

Quality assurance. Mockus et al. [23] investigated the number of defects found in formal code inspection, and found that the inspection team size, the number of sessions, and the sequence of the inspection steps do not affect defects detection. Rigby et al. [25] also found the number of reviewers increase the number of defects found in mailing-list-based review. McIntosh et al. [21] studied the impact of code review coverage and code review participation on software quality with three Gerrit-based projects and found that both of them share a significant link with software quality, in terms of number of post-release defects.

Review interval. Rigby et al. [25] studied the code inspections on mailing list of six FLOSS projects exploring how participation of code inspection, experience and expertise of the authors and reviewers, churn and change complexity of the patches impact the review interval.

Contribution acceptance. Weißgerber et al. [35] found that small patch has a higher chance to be accepted in a study on two FLOSS projects. Jiang et al. [15] inspected the patch acceptance of Linux kernel project, and found that experience of developers, number of affected subsystems and number of requested reviewers relevant. Bosu et al. [5] investigated the impact of developer reputation on code review acceptance in eight FLOSS Projects which use Gerrit and observed that the core developers receive quicker first feedback on their review request, complete the review process in shorter time, and are more likely to have their code changes accepted into the project codebase. Recently Gousios et al. [12] conducted a study of the GitHub pull requests usage within 291 GitHub projects and inspected effects of a number of features on the acceptance of a pull request, including characteristics of the pull request, project and developer.

Evolution of tools from mailing lists to pull request systems suggests that modern tools may improve the code contribution practice. For example, projects migrate to GitHub for team collaboration in commercial context [18]. However, in FLOSS, the question of whether or not the changes of tools brought the changes in code contribution effectiveness remains open. In this study we aim to evaluate the difference in effectiveness of patch-based and pull-request-based tools, and to shed more light on the potential ways to make the contribution process efficient.

3. METHODOLOGY

We describe how the study is designed in Section 3.1. We introduce the studied projects in Section 3.2 and the used metrics in Section 3.3. We present the research questions in Section 3.4.

3.1 Study Design

We synthesize the results of published papers and a new investigation of four GitHub projects to understand the differences of code contribution practice for different tools. Figure 2 shows the design framework of this study.

First, we borrow metrics for analyzing contribution practice from literature as detailed in Section 3.3. The value of replicating software engineering experiments has been highlighted by many researchers [17, 6, 8]. By reusing the published metrics (and the results on patch-based tools) in the literature, we are able to efficiently and effectively study the differences between code contribution practices using patch-based and pull-request-based tools.

Second, based on available resources, we conduct two substudies to address the internal and external validity concerns of empirical studies [30]: 1) **within-project study** where we investigate the Rails project which migrated from an issue tracker to the GitHub pull request system. Software development is complicated and associated with many factors. The within-project study ensures that the project context is approximately constant during the studied time periods, such as the project domain, culture, leaders, project size and review policy. We choose to study Rails for reasons explained in Section 3.2.1. 2) **cross-project study**, where we compare a number of classic projects that use patch-based tools and pull-request-based tools respectively. In particular, we gather results of eight mailing-list-based projects from the published studies, and retrieve the contribution history of four GitHub projects that use pull request systems and measure their effectiveness using published metrics.

3.2 Projects

Table 1 presents an overview of the projects used in this study. In the following two sections, we elaborate on the nature of projects and data extraction for within- and cross-project studies respectively.

3.2.1 Within-Project Study

The Rails project is a web application framework based on Ruby. It was migrated from an issue tracker (Lighthouse[4]) to the GitHub pull request system in September 2010. We choose Rails for two reasons. First, Rails is a popular project on GitHub with its code contribution practices that may be typical for the GitHub projects. Second, none of the mailing-list-based projects studied in literature, such as [25, 3, 15] have migrated to pull-request-based tools. The retrieval of Rails' pull request data is detailed in Section 3.2.2. For the issue tracker data of Rails (before September 2010), we download the HTML pages and attachments of the issues from the Lighthouse archive. For each patch, we extract the submitter and submission time, the operator who changes the status of the issue and operation time, and the commenter and comment time. Rails had been using Lighthouse since May 2008 (until September 2010). Considering the instability in the transition period,

[4]https://rails.lighthouseapp.com

Table 1: Studied Projects

Project	Domain	SLOC	Period	Tool	Ref
Apache	web sever	402972	1996-2005	mailing list	[25]
		-	-		[3]
SVN	version control	300527	2003-2008		[25]
Linux	OS	9950703	2005-2008		
FreeBSD		4746027	1995-2006		
KDE	OS GUI	6494034	2002-2008		
Gnome		4076395	2002-2007		
Python	program language	-	-		[3]
Postgres	database	-	-		
Rails	web app framework	172305	2011.4-2014.6	pull request system	-
		142108	2008.6-2010.6	issue tracker	
		161100	2011.6-2013.6	pull request system	
jQuery	javascript library	64692	2011.4-2014.6	pull request system	
PPSSPP	PSP emulator	201785	2012.12-2014.6		
Rust	programming language	216041	2011.4-2014.6		

* mean SLOC in the studied period.

we skip three months before September 2010 and include the patches published within two years, i.e., between June 2008 and June 2010. Similarly, we include pull requests submitted within another two years, starting from one year after the end of using the issue tracker, i.e., between June 2011 to June 2013.

3.2.2 Cross-Project Study

As shown in Table 1, the eight projects using mailing lists are retrieved from previous two papers [25, 3]. One [3] covers three and the other[25] covers six, the Apache project was studied in both papers. We select four projects using pull request systems from GitHub. They are Rails, jQuery, PPSSPP and Rust, representing different application domains, scale, and popularity. jQuery is a small JavaScript library mainly used for building dynamic web pages, PPSSPP is a virtual emulator of Sony Play Station Portable, and Rust is a programming language which targets at running fast, preventing crashes, and eliminating data races. Among the four projects, Rust is the biggest with ∼210K SLOC, followed by PPSSPP and Rails with ∼200K and ∼170K SLOC respectively, and jQuery with ∼64K SLOC is the smallest. Rails and jQuery are popular on GitHub ranking among the top 20 of all the GitHub repositories in terms of number of forks and stars (20K+ stars, 9K+ forks) when this study is conducted, while Rust has 10K stars, 2K forks, and PPSSP is somewhat less popular with 1.6K stars and 700 forks.

We locate the primary repositories of the four GitHub projects: *rails/rails*[5] for Rails, *jquery/jquery*[6] for jQuery, *hrydgard/ ppsspp*[7] for PPSSPP, and *rust-lang/rust*[8] for Rust, and retrieve their code contribution data through GitHub API[9]. For each repository we obtain issues (pull requests are combined with issues in the API), issue comments (pull request comments), review comments (these comments are directly attached to *diff* snippets, which are separated from the pull request comments in GitHub API), comments (these comments are attached in the commit view outside pull request view, which are also separated from the previous two types of comments in the API), issue events (pull request events), which include *open*, *subscribe*, *merge*, *close*, etc., pull request commits and all the commits on *master* branch of the central repository. For each pull request, we extract the submitter and submission time, the operator who merged or closed the pull request and operation time, and the commenter and comment time, as what we do for the patches on the Rails' issue tracker. Before the version 2.0 of GitHub pull request system was released, it did not support discussions[10]. Therefore, for Rails, jQuery and Rust, we set a study period from April 2011, about half a year after the version 2.0 was released, to June 2014, half a year prior to when the data were retrieved for our study. Because PPSSPP started in November 2012, we retrieve its data from December 2012 to June 2014. The youth of pull request systems restrains us from using the same studied periods as the prior studies.

3.3 Metrics

Borrowing metrics from existing literature, we quantify contribution effectiveness from three aspects: contribution effort, time interval and contribution activeness.

3.3.1 Effort Accepted/Ignored

To what extent the contribution effort is not wasted represents one important aspect of the contribution effectiveness. We borrow the metric of contribution acceptance rate from Bird et al.'s paper [3]. To highlight the wasted effort, we add a new metric, contribution ignore rate.

M1.1 *accept rate of contributions*: the ratio of accepted contributions out of all the contributions. Many participants in FLOSS communities are volunteers spending their limited spare time [37, 26] working for the projects. The more contributions are accepted, the less effort of making and reviewing the code changes is wasted for both contributors and reviewers.

Bird et al. [3] detected accepted patches by searching the central repository and finding the file version that applied the patch in the mailing lists. We identify accepted pull requests through checking whether there is a *merged* event or a *closed* event associated with commits integrated in the central repositories. This approach was used by Gousios et al. [12], and they also analysed the comments to detect merged pull requests that are merged informally, which leads to an average accept rate of 84.7%. Because we do not incorporate this strategy, the number of merged pull requests detected by us should be a lower bound. The detection of accepted patches in issue trackers is relatively simple. If the status of the issue attaching the patch is changed to *committed* or *resolved*, we label the patch as accepted.

M1.2 *ignore rate of contributions*: the ratio of ignored contributions (that don't receive any response) out of all the contributions. This measure may represent the attention for peripheral contributors from the community, sug-

[5]https://github.com/rails/rails
[6]https://github.com/jquery/jquery
[7]https://github.com/hrydgard/ppsspp
[8]https://github.com/rust-lang/rust
[9]https://developer.github.com/v3/
[10]https://github.com/blog/712-pull-requests-2-0

gesting an appreciation for contributors' effort of making the code changes and therefore motivating contributors to sustain in the project [37]. The attention could help the contributors find out problems with their contributions, learn project skills, and continue getting involved in the community and may eventually have their contributions accepted.

In Rigby et al.'s study [25], if a patch posted to the mailing list did not receive response from other developers, it was considered as ignored. Similarly, we consider a pull request or a patch in issue report which neither received at least one comment from others nor has the status changed as ignored.

3.3.2 Time Interval

Time is a common concern for contribution process because it may affect project development course and contributors' enthusiasm. Borrowing the metrics from Rigby et al.'s study [25], we use the time until the first response and final resolution to demonstrate the efficiency of the community to process the contribution.

M2.1 *time until first response*: the time span from the submission of the contribution to the first response to the contribution. Quick response may make contributors feel appreciated and help motivate contributors to take more active participation in the project [37]. Rails employed Rails-bot[11] in 2015, which implies the need for rapid response in practice.

We calculate the response time from the submission of a contribution to the first comment or first issue status operation or a pull request event (introduced in Section 3.2.2), whichever occurred earlier.

M2.2 *time until resolution*: the time span from the submission of the contribution to the final comment or resolution (accepted or rejected) of the contribution (ignored contributions are filtered). The resolve time determines how fast the acceptable contributions can be merged and delivered to the users. Fast resolution may also reduce waste of time for failed (rejected) contributions.

We calculate the resolve time as the time spent from submission to the final comment of a contribution. Because there are spam comments on Rails' issue tracker which falsely extend the lead time, in the within-project comparison, we calculate resolve time of a contribution as the time spent from submission to the final issue or pull request status operation (stopped at *committed, resolved, merged, closed*) of a contribution. It should be noted that the study of Rigby et al. did not consider multiple versions of a single patch [16]. The resolve time for a contribution extracted from their study [25] may be a part of the whole process time, i.e., lower than actual value.

3.3.3 Contribution Activeness

It is important to have a wide participation in FLOSS projects, which may be the powerful "engine" supporting the high efficiency of code development [22, 34]. Evidences show that effective process increases user interest in contributing to FLOSS projects [11]. A tool that improves process may appeal to developers and promote them to make contributions. We use the number of contributions, a metric to measure the contribution activeness, to indicate whether or not developers prefer to use the tool to contribute.

M3.1 *contribution frequency*: number of contributions per month. As found by Rigby et al. [25], project size is associated with the frequency of contributions. Therefore, in our measurement, we adjust it with source lines of code (SLOC) of the project. Because SLOC changes while project evolves, we retrieve the SLOC of each project in December of each year (it is each month for the within-project comparison of Rails) in the studied period from the OpenHub[12] and calculate the mean value (because there is no SLOC data for Rust on OpenHub, we download its repository and count SLOC each year with the tool of *cloc*[13]).

[11]https://github.com/rails/rails-bot

Table 2: Contributions Accepted in Rails

Tools	Submissions			Fisher's exact test	
	All	Accepted	Ratio	P-value	Odds ratio
issue tracker	2574	1652	64.2%	0.00768	1.14
pull request system	6014	4040	67.2%		

3.4 Research Questions

Targeting the research question RQ0, we measure the effectiveness of code contribution from three aspects: effort, time and activeness, and derive three specific research questions as follows:

- **RQ 1**: Is less contribution effort wasted when using pull request systems?
- **RQ 2**: Are contributions processed faster using pull request systems?
- **RQ 3**: Are contributions more frequent using pull request systems?

4. RESULTS

We answer the research questions through the analysis of within- and cross-project respectively.

4.1 Within-Project Comparison

From the issue tracker to the pull request system, Rails shows a minor variation of accept rate and ignore rate. However, both the response and resolve time are reduced over 90%, and the number of contributions are doubled.

Accept rate of contributions (M1.1). Table 2 shows that 67.2% of the contributions were accepted when Rails was using the pull request system, a bit higher than 64.2% when it was using the issue tracker. We employ Fisher's exact test to quantify the differences[14]. In the test, each observation is a contribution characterized by whether or not it is a pull request and whether or not it is accepted. The test produces a p-value < 0.01 with the odds ratio of 1.14. Although there is a statistically significant improvement of accept rate in the pull request system, the practical importance (odds ratio) is modest in magnitude.

Ignore rate of contributions (M1.2). The ignore rates of the two periods within Rails presented in Table 3, 0.9% and 0.6%, are both quite small. It appears that both the pull request system and the issue tracker perform well on

[12]https://www.openhub.net/

[13]http://cloc.sourceforge.net/

[14]Fisher's exact test is used to examine the significance of the association between two kinds of classifications [9]. We conduct the test using R: www.r-project.org.

Table 3: Contributions Ignored in Rails

Tools	Submissions			Fisher's exact test	
	All	Ignored	Ratio	P-value	Odds ratio
issue tracker	2574	22	0.9%	0.2598	0.74
pull request system	6014	38	0.6%		

Table 4: First Response Time in Rails

Tools	Hours (median)	P-value of Wilcoxon rank-sum test	Median decrease
issue tracker	30.5	$< 2.2 \times 10^{-16}$	98.4%
pull request system	0.5		

stimulating awareness of code contribution submissions. We employ Fisher's exact test to quantify the differences of the ignore rates, and obtain a p-value higher than the significance level of 0.05. That suggests an insignificant difference between the chance of a contribution getting ignored in pull request systems and the chance in issue trackers.

Time until first response (M2.1). Group FR.IT and FR.PR in Figure 3 show the boxplotof the first response time in the the issue tracker and the pull request system. The decrease from the issue tracker to the pull request system is dramatic. In Table 4, we can see that the median value of first response time in the pull request system decreases 98.4% compared to that in the issue tracker. We quantify their differences through Wilcoxon rank-sum test[15] using R, where each observation is the first response time of a contribution. The result verifies that Rails responds faster to the contributions using the pull request system than when it used the issue tracker.

Time until resolution (M2.2). Group RS in Figure 3 shows the boxplot of resolve time, where the median time for the pull request system is 98.7% lower than for the issue tracker. We quantify the difference through Wilcoxon rank-sum test and find statistically significant difference (see Table 5).

It appears the contribution resolve time is reduced by using the pull request system, we, therefore, fit linear regression models to verify the impact. We start with the model:

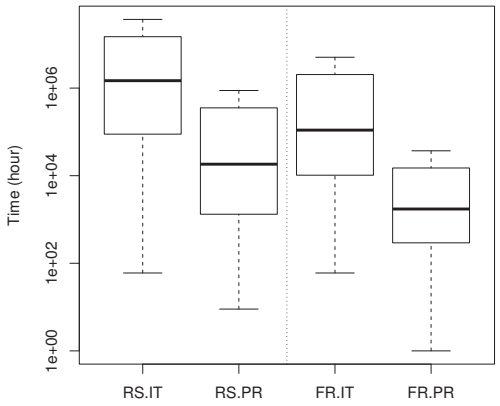
$$log(resolve\_time) \sim is\_PR \tag{1}$$

Each observation in it is a submitted contribution (patch or pull request). The response is the resolve time, and the predictors is whether the contribution is made via a pull request system.

[15]Wilcoxon rank-sum test is used to examine the significance that a particular population tends to have larger values than the other [20].

Table 5: Resolve Time in Rails

Tools	Hours (median)	P-value of Wilcoxon rank-sum test	Median decrease
issue tracker	390.2	$< 2.2 \times 10^{-16}$	98.7%
pull request system	5.0		



* Group RS and Group FR indicate resolve and first response time respectively. IT and PR are short for issue tracker and pull request system respectively. The Time axis is in log scale.

Figure 3: Time intervals in Rails.

The fitting results of Model 1 shows the use of the pull request system is significantly correlated with the resolve time. The factors that influence the resolve time of contributions have been previously investigated. We, therefore, add the predictors that have been widely considered (e.g., [25]) and fit a more complicated model:

$$\begin{aligned} log(resolve\_time) \sim\ & log(Churn+1) + log(\#Reviewers+1) \\ & + log(CExperience+1) + log(RExperience+1) \\ & + log(CExpertice+1) + log(RExpertice+1) \\ & + is\_PR \end{aligned} \tag{2}$$

The predictors are defined as following.

Churn of the contribution (Churn): the number of added and removed lines of code in the contribution. This is a common measure in the literature [25] and several studies have found that the contributions with small change size would have a higher chance to be accepted.

Experience of contributor (CExperience) and reviewer (RExperience): the time between a participant's first message in tool and the time of the submission.

Expertise of contributor (CExpertice) and reviewer (RExpertice): the adjusted amount of previous submissions or reviews done by a participant. The adjustment is detailed in Rigby et al.'s paper [25].

Number of reviewers (#Reviewers): the number of participants who comment or manage the contribution except for its contributor. The level of reviewer participation is found to have the largest impact on review time [25].

The fitting results of Model 2 are shown in Table 6. The coefficients of is_PR (whether the submission is a pull request) is significant at <0.005 level. Its negative sign means that use of the pull request system decreases the resolve time. The analysis of variance shows that 3% of the deviance is explained by is_PR in this model, which ranks in the middle among all the predictors. The variations from the previous studies are discussed in Section 5.1.

Contribution frequency (M3.1). The number of monthly contributions adjusted for codebase size increases as Rails moves from the issue tracker to the pull request system, as

Table 6: Model for Resolve Time in Rails

Model 1	Est	Std.Err.	P-value
$(Intcpt)$	13.481	0.077	$< 2 \times 10^{-16}$
is_PR	-3.379	0.900	$< 2 \times 10^{-16}$
adjusted R-square: 0.1488			
Model 2	**Est**	**Std.Err.**	**P-value**
$(Intcpt)$	12.035	0.229	$< 2 \times 10^{-16}$
is_PR	-1.533	0.099	$< 2 \times 10^{-16}$
$log(Churn + 1)$	0.050	0.025	0.048
$log(\#Reviewers + 1)$	1.927	0.066	$< 2 \times 10^{-16}$
$log(CExperience + 1)$	-0.049	0.018	0.006
$log(RExperience + 1)$	0.053	0.053	0.314
$log(CExpertise + 1)$	-0.430	0.035	$< 2 \times 10^{-16}$
$log(RExpertise + 1)$	-0.430	0.040	$< 2 \times 10^{-16}$
adjusted R-square: 0.3264			

Table 7: Number of Monthly Contributions in Rails

Tools	Average number (SLOC adjusted)	P-value of Wilcoxon rank-sum test	Averagely increase
issue tracker	107.25 (8.11×10^{-4})	3.151×10^{-11}	102.2%
pull request system	250.5 (1.64×10^{-3})		

shown in Figure 4. Table 7 presents the normalized number of monthly contributions in Rails. The number in pull request system period is double of that in issue tracker period. The Wilcoxon rank-sum test, where each observation is the number of contributions adjusted for SLOC per month, produces the p-value < 0.01 indicating that the difference is statistically significant.

4.2 Cross-Project Comparison

Compared to mailing lists, pull request systems double the accept rate and decreases ignore rate around 90%. Both the response and resolve time in pull request systems are less than half of that in mailing lists. The number of (adjusted) contributions is increased by a magnitude.

It's worth noting not all of the mailing list sources appear in each comparison because not all the compared measures were reported in the papers from which we obtain them.

Accept rate of contributions (M1.1). Table 8 shows the accept rates of projects that use mailing lists and pull request systems respectively. Less than half of the contributions are accepted in all the mailing-list-based projects with Postgres having the highest accept rate of 48.9%. Three out of four projects using pull request systems have higher accept rates than Postgres, except for jQuery with a 36.7% rate. We quantify the differences with Fisher's exact test where each observation is a contribution characterized by whether or not it is a pull request and whether or not it is accepted. The resulting p-value is statistically significant with the odds ratio of 3.86 favoring accept rates in pull request systems over accept rates in mailing lists.

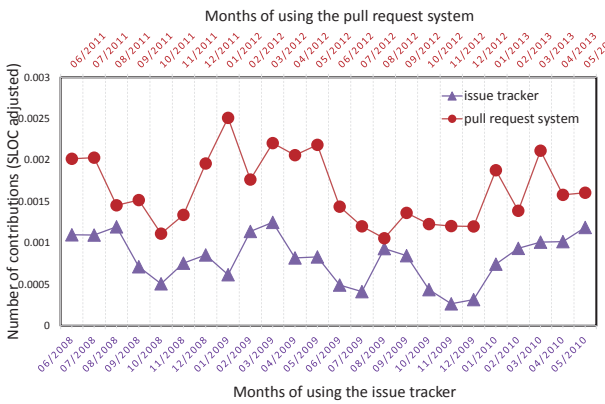


Figure 4: Monthly contributions of Rails.

Table 8: Contributions Accepted across Projects

Projects	Tools	Submissions			Ref
		All	Accepted	Ratio	
Apache	mailing list	4267	1087	25.5%	[3]
Python		644	173	26.9%	
Postgres		1209	591	48.9%	
Rails	pull request system	9933	6571	66.2%	-
jQuery		1304	479	36.7%	
PPSSPP		3666	3113	84.9%	
Rust		6723	3377	50.2%	
p-value, odds ratio of Fisher's exact test: $< 2.2 \times 10^{-16}$, 3.86					

Table 9: Contributions Ignored across Multiple Projects

Projects	Tools	Submissions			Ref
		All	Ignored	Ratio	
Apache	mailing list	4.6K	1.2K	26.1%	[25]
SVN		2.9K	0.1K	3.4%	
Linux		50K	22K	44%	
FreeBSD		73K	48K	65.8%	
KDE		22K	14K	63.6%	
Gnome		12K	4K	33.3%	
Rails	pull request system	9933	53	0.5%	-
jQuery		1304	24	1.8%	
PPSSPP		3666	14	0.4%	
Rust		6723	59	0.9%	
p-value, odds ratio of Fisher's exact test: $< 2.2 \times 10^{-16}$, 0.0059					

Ignore rate of contributions (M1.2). For ignored contributions, the performance difference between tools is bigger than that of the within-project study (see Table 9). Only a small portion (0.4% to 1.8%) of contributions are ignored in pull-request-based projects, while ignore rates of mailing-list-based projects are much higher except for SVN (3.4%). There is a large variation among projects with mailing lists. SVN has very low rates while more than 60% of the contributions are ignored in FreeBSD and KDE. Meanwhile, SVN has 3.4% even in mailing lists while jQuery has 1.8% in pull request systems. The exceptional performance of mailing lists in SVN may be at least partially attributed to a very strong discipline of the team to follow community policy of reviewing all published contributions [10]. Fisher's exact test yields p-value < 0.01 indicating statistical significance of the impact. The odds ratio of 0.0059 shows that magnitude of the impact is large.

Time until first response (M2.1). Third column in Table 10 shows the median first response time of contributions in the projects. The median first response time in projects using mailing lists ranges from 2.3 to 6.5 hours, and for pull request systems, it ranges from 0.5 to 5.3. The slowest and fastest of pull-request-based projects are faster than the two of mailing-list-based project respectively. We quantify the difference through Fisher's exact test. Because only the shortest and longest median first response times of mailing-list-based projects (those without $\sim$ prefix in Ta-

Table 10: First Response and Resolve Time across Multiple projects

Projects	Tool	First response time (hour)	Resolve time (hour)	Ref
Apache	mailing list	∼2.5	∼26	[25]
SVN		∼5	46	
Linux		2.3	∼33	
FreeBSD		∼6	23	
KDE		∼3	∼35	
Gnome		6.5	∼38	
Rails	pull request system	0.5	5.1	-
jQuery		2.3	44.2	
PPSSPP		0.5	1.6	
Rust		4.0	21.6	
Fisher's exact test		p-value: $< 2.2 \times 10^{-16}$ odds ratio: 1.55	p-value: $< 2.2 \times 10^{-16}$ odds ratio: 1.76	-

* all the values are median. Numbers with prefix ∼ are estimated according to the figure in [25].

Table 11: Number of Monthly Contributions across Multiple Projects

Tools	Projects	Number of contributions		Ref
		All (months, SLOC)	/month*	
mailing list	Apache	4600 (118, 402972)	9.67×10^{-5}	[25]
	SVN	2900 (67, 300527)	1.44×10^{-4}	
	Linux	50000 (42, 9950703)	1.20×10^{-4}	
	FreeBSD	73000 (144, 4746027)	1.07×10^{-4}	
	KDE	22000 (67, 6494034)	5.06×10^{-5}	
	Gnome	12000 (70, 4076395)	4.21×10^{-5}	
pull request system	Rails	9933 (39, 172305)	1.48×10^{-3}	-
	jQuery	1304 (39, 64692)	5.17×10^{-4}	
	PPSSPP	3666 (19, 201785)	9.56×10^{-4}	
	Rust	6723 (39, 216041)	7.98×10^{-4}	
p-value of Wilcoxon rank-sum test			0.0095	-

* all the values are mean and adjusted with (divided by) the codebase size, in terms of average SLOC in the studied period.

ble 10) are given in Rigby et al.'s paper [25], we select Linux kernel, which has the shortest time, as benchmark to test the pull-request-based projects. In the test, each observation is the first response time of a contribution characterized by whether it is made via a pull request system and whether it is lower than the median first response time of Linux kernel. The p-value is lower than 0.01, i.e., the difference is significant.

Time until resolution (M2.2). The forth column of Table 10 presents the median[16] resolve time of contributions in the projects. The median resolve time in the projects using mailing lists ranges from 23 to 46 hours. For pull-request-based projects, the fastest is PPSSPP where it takes only 1.6 hours in median, and the slowest, jQuery (44.2 hours in median), is still faster than the mailing-list-based SVN.

We quantify the difference through Fisher's exact test. Similar to the response time, only the shortest and longest median resolve times of mailing-list-based projects (those without ∼ prefix in Table 10) are given in [25], we use the fastest one, FreeBSD, as benchmark to test the pull-request-based projects. In the test, each observation is the resolve time of a contribution, and each contribution is characterized by whether it is a pull request and whether it has lower than the median resolve time of FreeBSD. The resulting p-value is lower than 0.01, suggesting the difference is significant.

Contribution frequency (M3.1). Table 11 shows the average number of monthly contributions adjusted for codebase size. All the numbers of mailing-list-based projects are higher than those of pull-request-based projects. We quantify the difference through Wilcoxon rank-sum test, where each observation is the (adjusted) average number of monthly contributions to a project. The p-value of the test is lower than the significance level of 0.01, therefore, we accept the alternative hypothesis that pull request systems are associated with more contributions than mailing lists.

So far, we obtain positive answers to RQ2 and RQ3 but no firm answer to RQ1. In summary, the answer to RQ0 is that contributions are processed faster, and participants contribute more frequently in pull-request-based tools. The GitHub pull request system showed a substantial improvement on accept and ignore rates in cross-project comparison, but not in the within-project comparison. These variations in the effort reduction may be most simply explained by the differences in contribution practices the projects employ and may be less affected by the type of tool used for code contribution. For example, older projects (with exception of SVN) may be used to the low accept and high ignore rates for historic reasons, while newer projects may be more tolerant. The accept rates may be more likely to be underestimated in mailing-list-based projects than in pull-request-based projects.

[16] we use median because the time until resolution is heavily skewed and mean is not a good summary statistic for such data.

5. DISCUSSION

We discuss the differences of our findings from previous studies in Section 5.1 and the insights of using pull request systems in Section 5.2.

5.1 Differences from Previous Findings

We model resolve time with predictors used in the existing literature and with an additional consideration on tool type. We discovered several interesting exceptions and variations.

Compared to earlier results [25], the significance and coefficient sign of some predictors flip over in our results. In particular, the significance of code churn and reviewers' experience disappears, while it emerges for reviewers' expertise. The sign of contributors' experience changes from positive to negative. We speculate that pull request systems may simplify the review of complex contributions and make code churn less impact the resolve time. These may be instances of the theory that social media has dramatically changed the landscape of software engineering, challenging some old assumptions about how developers learn and work with one another [31]. We develop conjectures for the variations in 5.2.

In the survey conducted by Gousios et al. [14], respondents told that reaching consensus of the decision on a contribution was challenging and the process could be delayed. However, we find that pull request systems reduce the time spent. The respondents did not mention how pull request systems perform compared to other tools, it would be interesting to make further investigation with developers who have experience in using different tools.

We reproduced the analysis used by Gousios et al. [12] to model the chance of a contribution getting accepted in Rails. We didn't find the most relevant factor (the number of total commits on files touched by the patch or pull request over three months before the submission time) to be significant and that may require further investigation.

5.2 Insights of Using Pull Request Systems

5.2.1 Advanced Features of Pull Request Systems

Integrating with DVCS. The pull request systems are built on the distributed version control system (DVCS) like Git. The branch model of DVCS is considered to be efficient and help participants save effort [2]. Contributors could *rebase* their branches [4] to relieve the core team members from solving the merge conflicts. A contribution through a pull request system is a sequence of commits on the contributor's branch, which can be easily merged into the target branch in the central repository. The update of a contribution is simple: what the contributor needs to do is making new commits on the branch in his own fork.

The pull request systems' close relationship with codebase may also lead to quick response. When maintainers are working on the central codebase, they will immediately find new contributions and give a quick response. If there is no conflict, the maintainer can merge the accepted submission through just clicking a button.

Track mechanism. The pull request systems automatically track the contributions. In the track mechanism [28], contributions are tracked independently, and participants can easily find the new submitted or updated contributions through key words, status, etc. While, mailing lists broadcast contributions, i.e., users are passively receiving emails, and core developers may receive as many as 300 emails per day [27]. Participants have to make strategies to filter the mails themselves to avoid being overwhelmed. If the filtering is not well handled, they may miss what they are interested in or spend a long time to discover them. Meanwhile, the states (e.g., open, resolve) of the contributions benefit awareness that is considered important in the distributed tasks [19]. For example, the *open* pull requests always make developers aware that there are still contributions not resolved. However, contributions via mailing lists are not traced with a state, and participants have to remember the unresolved ones. Naturally, some contributions may remain forgotten and ignored.

The insignificant difference between pull request systems and issue trackers on ignore rate indicates that pull request systems and issue trackers may be more adept at keeping track of submissions. As the primary function of an issue tracker is to keep track of things, it is reasonable that it does well as a pull request system in this regard.

Review board. The pull request systems' review board has two modules, the *code viewer* and the *review list*.

The *diff* and *comment* functionalities of the *code viewer* clearly present the code changes with colored lines, e.g., red for deletion and green for addition. That could promote the communication between submitters and reviewers to improve the code quality iteratively and lead to the acceptance [32]. Comments can be attached on code lines in *diff* view. It associates the review comment with specific code to make the discussion convenient and efficient. While, mailing lists and issue trackers do not have such functions.

The *review list* offers more convenient communications for a code submission [28]. First, most of the discussions of one contribution are recorded in single track, which simplifies the review of previous discussions. Second, any contributor can subscribe to a contribution to receive notifications. This makes followers aware of news of a contribution. Third, the states of contributions can remind the followers the in-processing contributions. However, the communications around a contribution via mailing list are scattered into different mails, threads, and even lists [27, 16]. It's hard for the followers to recall the ongoing contributions and review the discussion thread each time a change is made to the contribution. Therefore, the iteration of contributions via pull request systems are likely to be more efficient.

It's worth noting that the convenience of communication may make the inspection outcome more complex than simple acceptance or rejection [32]. Both core project members and third-party stakeholders sometimes implemented alternative solutions to address the issues over both the appropriateness of the problem that the submitter attempted to solve and the correctness of the implemented solution.

5.2.2 Influence of Social Platform

We observed a statistically significant increase of contributions for projects using pull request systems. The pull request system we studied is embedded in the popular platform, GitHub. In addition to integrating the advanced features of the pull request system, being a social collaboration platform itself, GitHub may substantially contribute to the broader participation.

Integration. GitHub integrates code submission, issue tacking and review discussion together, helping simplify the pull request contribution process and therefore may stimulate the participation. The publication of a contribution can be easily achieved within the system through clicking the *New pull request* button, and for maintainer if there is no conflict, she merges the accepted submission through clicking the *Merge* button. While in issue trackers and mailing lists, contributors have to follow guidances to make a patch, and publish the patch by sending an email or reporting/commenting an issue. Meanwhile, the authorship of contributions through pull requests on DVCS is kept by default and is easy to trace. That helps to build a contributor's reputation [5] and therefore, may motivate contributors to stay and continue their contributions. While in issue trackers and mailing lists, the authorship has to be manually maintained and is easy to get lost in the codebase.

Some automatic technologies that have been integrated into GitHub may also facilitate the contribution process. For example, the continuous integration system, Travis-CI, is found to increase software quality and team productivity [33].

Collaboration. GitHub is a fast growing software development platform[17]. It promotes "social coding" with a number of features which are similar to those of the social network sites such as Twitter. The users can follow others, star repositories, watch a repositories, etc. The transparency of such social activities can help people find what they are interested in and participate [7]. Moreover, the social network on GitHub is found to connect the projects with common participants [36]. As a result, GitHub provides the

[17]It has been used for other purpose, e.g., education, data sharing, writing books, etc.

hosted projects a large pool of potential contributors and is likely to facilitate the contribution participation.

6. LIMITATIONS

The way measures are collected, the confounding factors that may not be considered, and the generalization of the results are primary limitations of this study.

Measures. First, the measures of contributions using patch-based and pull-request-based tools in different epochs may introduce bias as the practice may evolve over time. This limitation is unavoidable as discussed at the end of Section 3.2.2. Second, in mailing lists, different versions of a patch may scatter into multiple mails and mail threads. In the study of Bird et al. [3], related patch versions are not grouped and each patch is split by files, therefore, the number of contributions is higher than actual. In this study we only use the accept rate, which is relative value and may not be sensitive to the absolute number. Third, both related papers and our analysis may suffer from the method of detecting accepted submissions. There could be accepted submission being detected as rejected (false negative) and rejected submission being identified as accepted (false positive). Fourth, the contributions through pull request systems may be made by mistake and closed by the authors themselves without external response. The metric defined in the literature would incorrectly regard such contributions as ignored. In this paper, we filter out the pull requests closed by their authors within one hour. Finally, the within-project study is conducted between the issue tracker and the pull request system rather than mailing lists and pull request systems because Rails switched from the issue tracker to the pull request system and we do not have other projects in our sample where a switch between mailing lists and pull request systems has occurred.

Confounding factors. Software development is a knowledge-intensive activity with a large number of potentially confounding factors [37], and this makes it difficult or impossible to discern the impact of code contribution tools. We address the challenges from a study design including a within-project comparison, where the project context is controlled, and a cross-project comparison, where the external validity is considered. We spend effort on reusing existing metrics and reproducing published analysis in order to make fair comparisons between different tools.

Generalization. In the comparison of tools, the mailing-list-based projects are borrowed from previous papers [25, 3] and the pull-request-based projects are selected from GitHub covering a variation of domain and scale. They may not be able to represent all the FLOSS projects, but they cover a relatively large scope of application domains and project size. Moreover, the comparison between issue trackers and pull request systems is only conducted with the data of one project, Rails, because of the availability of projects that changed their tools, and this may restrict the generalization of our findings.

7. CONCLUSIONS

"Build software better, together." is the slogan of GitHub, meaning that it aims for a better collaboration in software development through implementing the distributed fork&pull request model, which makes contributions easier to make, evaluate, improve and integrate. In this study, we investigate the effectiveness of FLOSS code contribution practice through the GitHub pull request system and compare it to patch-based tools. We measure tool effectiveness via the effort, time and activeness, and compare within a project that changed from an issue tracker to an pull request system and across projects that use mailing lists and pull request systems respectively. The results show that modern tools, such as pull request systems, have a lower processing time and attract more participation. We argue that these improvements are at least partially attributed to advanced features of pull request systems. In particular, the coupling with code repository, issue tracking and review discussion enables easier participation and better traceability. This, in turn, helps reduce time and effort. Furthermore, the social features enhanced by the collaboration platform of GitHub may help attracting new contributors and contributions.

In practice, this knowledge may help improve tools and practices for code contribution. Projects using traditional tools may want to add features of pull request systems that are associated with higher contribution effectiveness by plug-ins, additional modules, ect. Practitioners who have already used or intend to move to pull request systems may need to pay more attention to leveraging these features. Pull request systems' high efficiency, however, does not imply that patch-based tools should be abandoned. We observe that good discipline and skills of using traditional patch-based tools may work well too. For example, Linux kernel still uses its mailing list to do code inspection even though it has a mirror repository hosted on GitHub. It provides a guidance of making contribution title prefixed by tags enclosed in square brackets: "Subject: [PATCH tag] <summary phrase>"[18]. SVN has the community norm to have contribution reviewed that appears to be quite effective [10]. Such strategies should not be discounted when trying to improve the effectiveness of code contribution. Although pull request systems have a higher effectiveness, they may have weakness that can be improved. For example, some developers think there should be code analysis functionality for quality assurance on the GitHub pull request system, some are still unsatisfied with the way GitHub handles notifications [14]. Maintainers find reaching consensus of the decision of a contribution through the pull request comment mechanism and handling the workload imposed by the open submission process of pull request systems to be sometimes challenging [14]. Contributors think the communication within pull requests, although effective for discussing low-level issues, appears to be limited for other types of their communication needs [13].

In future, we intend to conduct a survey with developers who have experience of using different types of tools to improve our understanding of exactly what has been achieved and what challenge of contribution practices are still outstanding.

8. ACKNOWLEDGMENTS

This work is supported by the National Basic Research Program of China Grant 2015CB352203 and the National Natural Science Foundation of China Grants 61432001, 61421091 and 91318301.

[18]https://www.kernel.org/doc/Documentation/SubmittingPatches

9. REFERENCES

[1] J. Asundi and R. Jayant. Patch review processes in open source software development communities: A comparative case study. In *40th Hawaii International International Conference on Systems Science (HICSS-40 2007), CD-ROM / Abstracts Proceedings, 3-6 January 2007, Waikoloa, Big Island, HI, USA*, page 166, 2007.

[2] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and isolated development with branches. In *Proceedings of the 15th International Conference on Fundamental Approaches to Software Engineering*, FASE'12, pages 316–331, Berlin, Heidelberg, 2012. Springer-Verlag.

[3] C. Bird, A. Gourley, and P. T. Devanbu. Detecting patch submission and acceptance in OSS projects. In *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings*, page 26, 2007.

[4] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[5] A. Bosu and J. C. Carver. Impact of developer reputation on code review outcomes in OSS projects: an empirical investigation. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 33, 2014.

[6] F. Q. Da Silva, M. Suassuna, A. C. C. França, A. M. Grubb, T. B. Gouveia, C. V. Monteiro, and I. E. dos Santos. Replication of empirical studies in software engineering research: a systematic mapping study. *Empirical Software Engineering*, 19(3):501–557, 2014.

[7] L. Dabbish, H. C. Stuart, J. Tsay, and J. D. Herbsleb. Leveraging transparency. *IEEE Software*, 30(1):37–43, 2013.

[8] C. V. de Magalhães, F. Q. da Silva, and R. E. Santos. Investigations about replication of empirical studies in software engineering: preliminary findings from a mapping study. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 37. ACM, 2014.

[9] R. A. Fisher. On the interpretation of $\chi 2$ from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, pages 87–94, 1922.

[10] K. Fogel. *Producing open source software - how to run a successful free software project*. O'Reilly, 2005.

[11] A. H. Ghapanchi, A. Aurum, and F. Daneshgar. The impact of process effectiveness on user interest in contributing to the open source software projects. *Journal of software*, 7(1):212–219, 2012.

[12] G. Gousios, M. Pinzger, and A. van Deursen. An exploratory study of the pull-based software development model. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 345–355, 2014.

[13] G. Gousios, M.-A. Storey, and A. Bacchelli. Work practices and challenges in pull-based development: The contributor's perspective. In *Proceedings of the 38th International Conference on Software Engineering*, pages 285–296. ACM, 2016.

[14] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 358–368. IEEE Press, 2015.

[15] Y. Jiang, B. Adams, and D. M. Germán. 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, San Francisco, CA, USA, May 18-19, 2013*, pages 101–110, 2013.

[16] Y. Jiang, B. Adams, F. Khomh, and D. M. Germán. Tracing back the history of commits in low-tech reviewing environments: a case study of the linux kernel. In *2014 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '14, Torino, Italy, September 18-19, 2014*, page 51, 2014.

[17] N. Juristo and S. Vegas. Using differences among replications of software engineering experiments to gain knowledge. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 356–366. IEEE Computer Society, 2009.

[18] E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, and D. German. Open source-style collaborative development practices in commercial projects using github. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, 2015.

[19] S. Levin and A. Yehudai. Improving software team collaboration with synchronized software development. *arXiv preprint arXiv:1504.06742*, 2015.

[20] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[21] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan. The impact of code review coverage and code review participation on software quality: a case study of the qt, vtk, and ITK projects. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, pages 192–201, 2014.

[22] A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. In J. Feller and et al, editors, *Perspectives on Free and Open Source Software*, pages 163—210. MIT Press, 2005.

[23] A. Mockus, A. Porter, H. Siy, and L. G. Votta. Understanding the sources of variation in software inspections. *ACM Transactions on Software Engineering and Methodology*, 7(1), January 1998.

[24] M. Nurolahzade, S. M. Nasehi, S. H. Khandkar, and S. Rawal. The role of patch review in software evolution: an analysis of the mozilla firefox. In *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution*

(IWPSE) and software evolution (Evol) workshops, pages 9–18. ACM, 2009.

[25] P. C. Rigby, D. M. Germán, L. Cowen, and M. D. Storey. Peer review on open-source software projects: Parameters, statistical models, and theory. *ACM Trans. Softw. Eng. Methodol.*, 23(4):35, 2014.

[26] P. C. Rigby, D. M. Germán, and M. D. Storey. Open source software peer review practices: a case study of the apache server. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 541–550, 2008.

[27] P. C. Rigby and M. D. Storey. Understanding broadcast based peer review on open source software projects. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 541–550, 2011.

[28] N. Serrano and I. Ciordia. Bugzilla, itracker, and other bug trackers. *Software, IEEE*, 22(2):11–13, 2005.

[29] B. D. Sethanandha, B. Massey, and W. Jones. Managing open source contributions for software project sustainability. In *Technology Management for Global Economic Growth (PICMET), 2010 Proceedings of PICMET'10:*, pages 1–9. IEEE, 2010.

[30] J. Siegmund, N. Siegmund, and S. Apel. Views on internal and external validity in empirical software engineering. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, pages 9–19, 2015.

[31] M.-A. Storey, L. Singer, B. Cleary, F. Figueira Filho, and A. Zagalsky. The (r) evolution of social media in software engineering. In *Proceedings of the on Future of Software Engineering*, pages 100–116. ACM, 2014.

[32] J. Tsay, L. Dabbish, and J. D. Herbsleb. Let's talk about it: evaluating contributions through discussion in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 144–154, 2014.

[33] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.

[34] G. von Krogh, S. Spaeth, and K. R. Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241, July 2003.

[35] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR 2008 (Co-located with ICSE), Leipzig, Germany, May 10-11, 2008, Proceedings*, pages 67–76, 2008.

[36] L. Zhang, Y. Zou, B. Xie, and Z. Zhu. Recommending relevant projects via user behaviour: An exploratory study on github. In *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies*, CrowdSoft 2014, pages 25–30, New York, NY, USA, 2014. ACM.

[37] M. Zhou and A. Mockus. Who will stay in the floss community? modeling participant's initial behavior. *IEEE Transactions on Software Engineering*, 41(1):82–99, Jan 2015.