Studying the Impact of Adopting Continuous Integration on the Delivery Time of Pull Requests

João Helis Bernardo
Federal Institute of Rio Grande do Norte
Pau dos Ferros, Brasil
joao.helis@ifrn.edu.br

Daniel Alencar da Costa
Queen's University
Kingston, Canada
daniel.alencar@queensu.ca

Uirá Kulesza
Federal University of Rio Grande do Norte
Natal, Brazil
uira@dimap.ufrn.br

ABSTRACT

Continuous Integration (CI) is a software development practice that leads developers to integrate their work more frequently. Software projects have broadly adopted CI to ship new releases more frequently and to improve code integration. The adoption of CI is motivated by the allure of delivering new functionalities more quickly. However, there is little empirical evidence to support such a claim. Through the analysis of 162,653 pull requests (PRs) of 87 GitHub projects that are implemented in 5 different programming languages, we empirically investigate the impact of adopting CI on the time to deliver merged PRs. Surprisingly, only 51.3% of the projects deliver merged PRs more quickly after adopting CI. We also observe that the large increase of PR submissions *after* CI is a key reason as to why projects deliver PRs more slowly *after* adopting CI. To investigate the factors that are related to the time-to-delivery of merged PRs, we train regression models that obtain sound median R-squares of 0.64-0.67. Finally, a deeper analysis of our models indicates that, *before* the adoption of CI, the integration-load of the development team, i.e., the number of submitted PRs competing for being merged, is the most impactful metric on the time to deliver merged PRs *before* CI. Our models also reveal that PRs that are merged more recently in a release cycle experience a slower delivery time.

KEYWORDS

continuous integration; pull-based development; pull request; delivery time; delivery delay

ACM Reference Format:
João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. 2018. Studying the Impact of Adopting Continuous Integration on the Delivery Time of Pull Requests. In *Proceedings of 15th Working Conference on Mining Software Repositories (MSR 2018).* ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The increasingly user demands for new functionalities and performance improvements rapidly changes customer requirements and turn software development into a competitive market [37]. In this scenario, software development teams need to deliver new functionalities more quickly to customers to improve the time-to-market [10, 23]. This faster delivery may lead customers to become engaged in the project and to give valuable feedback. The failure of providing new functionalities and bug-fixes, on the other hand, may reduce the number of users and the project's success.

The agile methodologies, such as Scrum [30] and Extreme Programming (XP) [1], brought a series of practices with the allure of providing a more flexible software development and a faster delivery of new software releases. The frequency of releases is one of the factors that may lead a software project to success [3, 38]. The release frequency may also indicate the vitality level of a software project [7].

In order to improve the process of shipping new releases, i.e., in terms of software integration and packaging, Continuous Integration (CI) appears as an important practice that may quicken the delivery of new functionalities [23]. In addition, CI may reduce problems of code integration in a collaborative environment [33].

The CI practice has been widely adopted by the software community [11] in open source and industrial settings. It is especially important for open source projects given their lack of requirement documents and geographically distributed teams [33]. To the best of our knowledge, no prior research empirically verified the impact of CI on the time that is needed to deliver new software functionalities to end-users.

Existing research has analyzed the usage of CI in open source projects that are hosted in GitHub [19, 33, 34]. For instance, Vasilescu *et al.* [34] investigated the productivity and quality outcomes of projects that use CI in GitHub. They found that projects that use CI merge pull requests (PRs) more quickly when they are submitted by core developers. Also, core developers discover a significantly larger amount of bugs when they use CI. According to Ståhl and Bosch [32], CI may also improve the release frequency, which hints that software functionalities may be delivered more quickly for users. Additionally, Zhao et al. [40] studied the impact of CI on development practices, such as code writing and submission, issue and pull request closing, and testing practices. The authors observe that practices such as "commit often" and "commit small" are indeed employed after the adoption of CI. However, the growing trend of closed issues slow down after the adoption of CI.

Nevertheless, little is known about whether CI quickens the delivery of new merged PRs to end users. This is an important investigation, since delays in releasing software functionalities can be frustrating to end-users because they are most interested in experiencing such new functionalities.

In this matter, our work empirically analyzes whether CI improves the time-to-delivery of new *Pull-Requests* (PRs) that are submitted to GitHub projects. GitHub provides an opportunity to

MSR 2018, Gothenburg, Sweden

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

investigate the impact of CI on the time to deliver new PRs. GitHub is considered the most popular version hosting worldwide [15], with more than 14 million of registered users, a wide variety of projects of different programming languages, sizes and characteristics. Our study investigates the impact of adopting CI in 87 GitHub projects that are implemented in 5 different programming languages.[1] We analyze a total of 162,653 PRs with 39,110 PRs *before* and 123,543 PRs *after* the adoption of CI. In particular, we address the following research questions:

- ***RQ1: Are merged pull requests released more quickly using continuous integration?*** Interestingly, we find that the time to deliver PRs is shorter *after* the adoption of CI in only 51.3% of the projects. In addition, we find that in 62 (62/87) of the studied projects, the merge time of PRs is increased *after* adopting CI.
- ***RQ2: Does the increased development activity after adopting CI increases the delivery time of pull requests?*** We find that there exists a considerable increase in the number of PR submissions and in the churn per releases *after* adopting CI. The increased PR submissions and churn are key reasons as to why projects deliver PRs more slowly *after* adopting CI. 71.3% of the projects increase the rate of PR submissions *after* adopting CI.
- ***RQ3: What factors impact the delivery time after adopting continuous integration?*** Our models indicates that, *before* the adoption of CI, the integration-load of the development team, i.e., the number of submitted PRs competing for being merged, is the most impactful metric on the delivery time of PRs *before* CI. On the other hand, our models reveal that *after* the adoption of CI, PRs that are recently merged in a release cycle are likely to have a slower delivery time.

Paper organization. The rest of this paper is organized as follows. In Section 2, we present the necessary background definitions to the reader. In Section 3, we explain the design of our empirical study. In Section 4, we present the results of our empirical study, while we discuss its threats to the validity in Section 6. In Section 7, we discuss the related work. Finally, we draw conclusions in Section 8.

2 BACKGROUND & DEFINITIONS

The goal of our work is to investigate the impact of adopting CI on the time that is needed to deliver merged PRs. In the following, we outline the necessary background definitions to the reader.

2.1 The pull-based development model

There are two general ways that potential contributors can submit their contributions to a software project in a distributed code-hosting environment (e.g., GitHub):

(i) shared repository. The core team shares the read and write accesses to the central repository, enabling external contributors to clone the repository, work locally and push their code contributions back to the central repository.

[1] https://prdeliverydelay.GitHub.io/#studied-projects

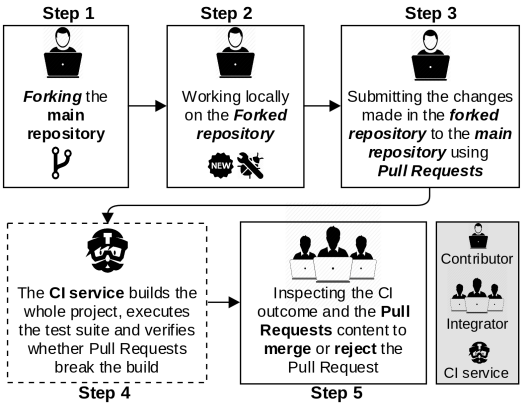


Figure 1: An overview of the pull-based development model that is integrated with Continuous Integration. The Step 4 is only performed when CI is used.

(ii) pull-based development. This paradigm is broadly used by contributors of open source projects to develop software in a distributed and collaborative way. The most popular code hosting providers, e.g. GitHub and Bitbucket allow any user to fork and clone any public repository and send PRs [14]. A PR is a mechanism enabled by Git that allows contributors to work locally on the forked repository and ask to have their contributions merged into the main repository. The writing access is not mandatory to submit PRs. Figure 1 shows an overview of the process to send contributions using PRs. We explain each step of the process below:

Step 1. Fork a repository. The main repository of a project is not shared to external contributors. Instead, contributors can clone the main repository by forking it, so they can modify the code without interfering in other repositories and with no need of being a team member.

Step 2. Work locally the forked repository. The contributors develop new functionalities, fix bugs or provide features and enhancements to the forked repository.

Step 3. Submit the local changes to the main repository. When changes are ready to be submitted, contributors request a pull of such changes to the main repository by sending a PR [39]. Such PR specifies the local branch that has to be merged into a given branch of the main repository.

Step 4. Verify whether the PR breaks the build. The CI service automatically merge the PR into a test branch. Next, the CI service builds the whole project and runs the test suite to verify whether the PR breaks the codebase. Typically, if tests fail during the process of CI, the PR is rejected and additional changes are required to the external contributor to improve his/her PR [39]. In case that all tests pass during the CI process, the integrators thoroughly review the PR before deciding to accept the contributions. This decision is based on the quality, technical design, and the priorities of the submitted PRs [16].

Step 5. Accept or reject a PR. After the PR submission, an integrator of the main repository must inspect the changes to decide whether they are satisfactory. In case that the changes fulfill the requirements of the project, the integrator pulls them to the specified

branch of the main repository. Otherwise, the core team may request additional changes to the external contributor to make his/her PR acceptable. In the pull-based development, the integrator plays a crucial role by managing contributions [16].

2.2 Continuous Integration in a pull-based development model

CI is a set of practices that lead developers to integrate their work more frequently, i.e., at least daily [12, 25]. All code must be maintained in a single repository. When a contributor commits to the repository, an automated system verifies whether the change breaks the codebase (*Step 4* of Figure 1) [25]. The entire process must be automated. Ideally, a build should compile the code and include a test suite to verify whether the codebase is broken after adding new changes. In CI, the work of developers is continually compiled, built, and tested [39].

CI is widely used on GitHub. According to Gousios *et al.* [16], 75% of GitHub projects that makes a heavy use of PRs also tend to use CI. Several CI services, such as Jenkins, TeamCity, Bamboo, CloudBees and Travis-CI [25] are available for development teams. Jenkins and Travis-CI are the most used by GitHub projects [34]. Travis-CI is a CI platform for open source and private GitHub projects. Currently, over 300k projects are using this tool.[2]

3 EMPIRICAL STUDY

In this section we explain how we select the studied projects and construct the database that we use in our analyses.

3.1 Studied Projects

Our goal is to identify projects that have a long historical data and that adopted CI at some point of their life. We use such projects to better understand the impact of adopting CI on the delivery time of merged PRs. We use a similar approach as used by Vasilescu *et al.* [34] to select our projects. The selection process of our projects is shown in Figure 3. We describe each step of this process in the following.

First, we use the GitHub API to identify the 3,000 most popular projects that are written in the five most popular programming languages (Java, Python, Ruby, PHP and JavaScript) of GitHub.[3, 4] The popularity of a project is measured by the number of stars that are assigned to that project.[5] We performed our search on GitHub in *November 11th, 2016*.

Next, we check whether a project adopts a CI service. In our study, we only consider projects that use Travi-CI. Similar to Vasilescu *et al.* [34], we avoid projects that use Jenkins, since the entire CI history of such projects is not available. We identify that a given project use Travis-CI when there are builds that are associated with the Travis-CI API. We use the date of the first Travis-CI build as the moment at which a project started to adopt CI. Out of 3,000 projects, 1,784 (59,5%) have used Travis-CI.

In step 3, we use the GitHub API to gather the merged PRs of each project. We group the PRs into the *before*- and *after*-CI buckets.

[2]https://travis-ci.org/
[3]https://developer.GitHub.com/v3
[4]https://GitHub.com/blog/2047-language-trends-on-GitHub
[5]https://help.GitHub.com/articles/about-stars/

Table 1: Summary of the number of projects and released pull requests grouped by programming language.

Language	Projects	PRs total	PRs *after* CI	PRs *before* CI
JavaScript	33	57,104	39,548	17,556
Python	23	55,003	45,896	9,107
Java	11	7,700	4,267	3,433
Ruby	10	22,864	19,667	3,197
PHP	10	19,982	14,165	5,817
Total	**87**	**162,653**	**123,543**	**39,110**

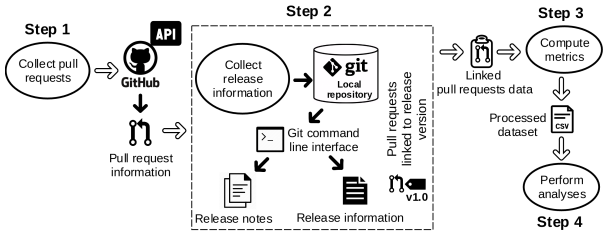


Figure 2: An overview of our data collection process.

We exclude projects that have less than 100 merged PRs in the *before* or the *after* buckets to maintain a considerable amount of data to perform our analyses. 156 projects remains after step 3.

In step 3, we also use the GitHub API to fetch all PRs and their metadata for the remaining projects. We then link the PRs to their specific releases. Such links help us to calculate the total time between when a PR was merged and when that PR was released. We refer to this time interval as to "delivery time". Finally, we filter out projects that have less than 100 linked PRs in the *before* or *after* buckets. A total of 90 projects remains.

Finally, we removed "toy projects" and projects with no releases *before* or *after* the adoption of CI. For example, students in software engineering courses may use GitHub for versioning their assignments. We refer to these cases as "toy projects" because their content are trivial and not suitable to our research. Hence, we verify the project name and it's README.md file to avoid toy projects in our analyses. Out of 90 projects, 87 remains after Step 4 (33 JavaScript, 23 Python, 11 Java, 10 Ruby and 10 PHP). 123,543 PRs were delivered *after* the adoption of CI, while 39,110 were delivered *before* the adoption of CI (a total of 162,653 PRs, see Table 1). This unbalanced number of PRs between project phase are related to how long projects have adopted CI. In average, our studied projects are 5.17 years old. The adoption of Travis-CI comprises 60% of the age of our projects. Table 1 shows the number of PRs per programming language *before* and *after* CI.

3.2 Data collection

After we select our studied projects, we fetch PR and release metadata for each project. The data collection process is shown in Figure 2. Each step of the process is detailed below.

Step 1. Collect pull request information. We use the GitHub API to collect PRs and their respective meta-data. For each PR, we select the following attributes: *author* (GitHub user), *pull-number*, *title*, *description*, *number of added and deleted lines (churn)*, *number*

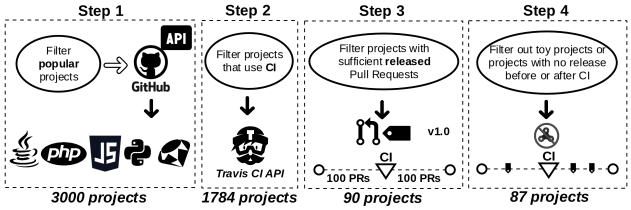


Figure 3: An overview of our project selection process.

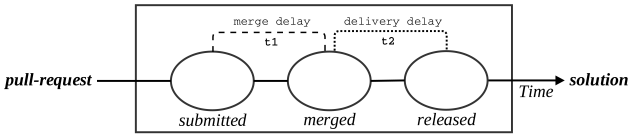


Figure 4: The basic life-cycle of a released pull request.

of changed files, *number of activities*, *number of comments*, *date of comments*, *state* (Open, Closed, Merged), *creation date*, *close date*, and *closedBy* (GitHub user).

Step 2. Link pull requests to releases. After we collect the PR information, we collect the release information of the studied PRs. We collect the *publish date*, *start date*, the *number of commits* and the *number of PRs* for each release of the studied projects. We also manually verify whether the releases are user-intended, so that we do not consider pre, beta, alpha, rc (release candidate) releases in our analyses. Instead, in case that a PR was released in a non-user-intended release, we link such a PR with the next user-intended release. For example, if a project has the following release tags: *v1.0*, *v1.1.pre*, *v1.1* and *v2.0*, chronologically ordered and a PR is released in *v1.1.pre* release, we move the publish date of this PR to the next user-intended release (i.e., the date of release *v.1.1*). We clone all repositories of our studied projects and fetch all of their releases tags. We then compute a diff between these tags to verify which commit logs were added in a given tag. Next, we parse our obtained commit logs. For instance, if we find the pattern *"Merge pull request #<**X**>"* (which is automatically generated by Git when a PR is merged) between release tags *v1.1* and *v2.0*, we consider that such a commit log was released in *v2.0*. By using the pattern *"Merge pull request #<**X**>"* we could link 84.1% (162,653/193,328) of the merged PRs to its commits. The remaining 15% may still be waiting for a release to be shipped or the integrator might have cherry picked the commits of these PRs. In the latter case, the pattern *"Merge pull request #<**X**>"* is not automatically recorded in the respective commit logs. Finally, we link merged PRs to their releases based on the tags that are associated to the commits.

Step 3 and 4. Compute metrics and perform analyses. We use data from Steps 1 and 2 to compute the metrics that we use in our analyses. The detailed information about all computed metrics for each PR is described in Section 4. We calculate these metrics because we suspect that they share a relationship with the *delivery time* of merged PRs.

4 RESULTS

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

RQ1 - Are merged pull requests released more quickly using continuous integration?

Motivation. In recent years, many software companies have adopted the CI practice in their development life cycle. This wide adoption is related to the perceived benefits that are brought by CI. For instance, the risk reduction, a higher confidence of the development team regarding their software product [11], higher productivity, higher release frequency and predictability [32], and the allure of delivering new features more quickly [23]. However, there is a lack of studies that empirically check the impact of CI on the time-to-delivery of merged PRs. In RQ1, we study the delivery time of merged PRs *before* and *after* the adoption of CI.

Approach. Figure 4 shows the basic life cycle of a released PR: ($t1$) merge phase; and ($t2$) delivery phase. We refer to the $t1 + t2$ time as to the lifetime of a PR. In RQ1, we analyze the *merge* and *delivery* phases. The *merge phase* ($t1$) is the required time for PRs to be merged into the codebase, whereas the *delivery phase* ($t2$) refers to the required time for PRs to be released after they have been merged, i.e., ready to be delivered to end-users.

We use Mann-Whitney-Wilcoxon (MWW) tests [35] followed by Cliff's delta effect-size measures [6]. The MWW test is a non-parametric test whose null hypothesis is that two distributions come from the same population ($\alpha = 0.05$). Cliff's delta is a non-parametric effect-size metric to verify the magnitude of the difference between the values of two distributions. The higher the Cliff's delta value, the greater the difference between distributions. A positive Cliff's delta shows how larger are the values of the first distribution, while a negative Cliff's delta shows the opposite. We use the thresholds provided by Romano *et al.* [28], i.e. $delta < 0.147$ (*negligible*), $delta < 0.33$ (*small*), $delta < 0.474$ (*medium*), and $delta >= 0.474$ (*large*). We use such statistical tools to analyze the entire life-cycle of a PR *before* and *after* CI. First, we analyze the PR *delivery time* ($t2$). Then, we analyze the *($t1$) merge time*. Finally, we investigate PR *lifetime* ($t1 + t2$).

Results. Only 51.3% of the projects deliver merged PRs more quickly after the adoption of CI. Out of our 87 projects, we observe that 82.7% (72/87) obtained significant $p-values$ (i.e., $p < 0.05$) when comparing the delivery time of merged PRs *before* and *after* adopting CI. Surprisingly, we observe that only 51.3% (37/72) of these projects deliver merged PRs more quickly *after* adopting CI. Our analyses indicate that 82.7% (72/87) of the projects have a statistical difference on the delivery time of merged PRs, but a small median Cliff's delta of 0.304.

In 73% (46/63) of the projects, PRs are merged faster before adopting CI. A total of 72.4% (63/87) of the projects have a statistical difference on the time to merge PRs with a median Cliff's delta of 0.206 (*small*). With respect to such projects, we observe that 73% (46/63) merge PRs more quickly *before* CI.

Surprisingly, in 54% of the projects, PRs have a longer lifetime after adopting CI. We observe that in 54% (47/87) of our projects, PRs have a larger lifetime after adopting CI. 71.3% (62/87) of these projects have a statistically significant difference (*p-value < 0.05*) and a $non-negligible$ median $delta$ between the distributions

of lifetime of PRs ($delta >= 0.147$). 37.1% (23/62) of such projects obtained a large delta (median 0.604), while 22.6% (14/62) and 40.3% (25/62) of the projects obtained medium and small deltas, respectively (medians of 0.362 and 0.223). Regarding the projects that obtained a $p - value < 0.05$, we observe that 51.6% (32/62) have a shorter PR lifetime *before* adopting CI, while 48.4% (30/62) had a shorter PR lifetime *after* adopting CI.

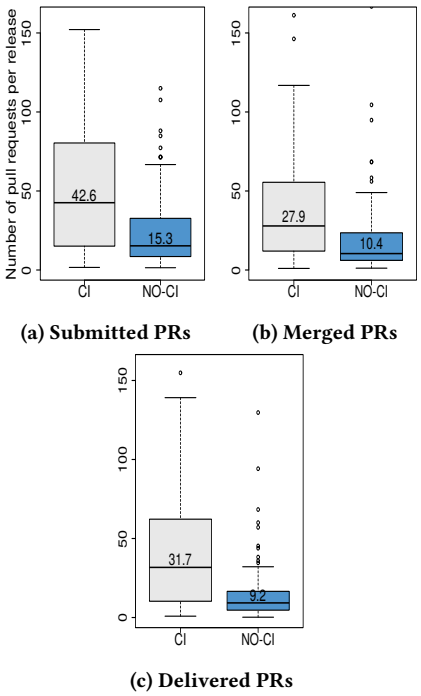
Surprisingly, only 51.3% of the projects deliver merged PRs more quickly after adopting CI. In 54% (47/87) of the projects, PRs experience a longer lifetime after the adoption of CI. Finally, PRs are merged faster before adopting CI in 71.3% (63/87) of the studied projects.

RQ2 - Does the increased development activity *after* adopting CI increase the delivery time of pull requests?

Motivation. In *RQ*1, we find that only 51.3% of the projects deliver merged PRs more quickly after adopting CI. Also, in 54% (47/87) of the projects, PRs experience a longer lifetime *after* adopting CI. These results contradict our assumption that merged PRs would be delivered more quickly *after* the adoption of CI in the great majority of our projects. Although the adoption of CI is motivated by the increase of the release frequency and predictability [32], our results suggest a different trend. Hence, we are inclined to ask the following question: Why do 54% of our studied projects have PRs that experience a longer lifetime *after* the adoption CI? This investigation is important to better understand the impact of adopting CI in software development.

Approach. Similar to RQ1, we use Mann-Whitney-Wilcoxon tests [35] and Cliff's deltas [6] to analyze the data. We also use box plots [36] to visually summarize and perform comparisons. In RQ2, we investigate whether the increase on the lifetime of PRs *after* adopting CI is related to a significant increase in the PR submission, merge and delivery rates *after* adopting CI. We group our dataset into two buckets: *before* and *after* the adoption of CI. For each bucket, we count the number of PRs that are submitted, merged and delivered per release. We perform three comparisons in this RQ. First, we compare whether the PR submission, merge and delivery rates per release significantly increase *after* adopting CI. Next, we verify whether there is a statically difference in the release frequency of the projects *after* adopting CI. A high increase or decrease in the release frequency also may lead to an increase or decrease in the PR delivery rate per release, once the release size changes. In addition, we use the Pearson correlation test [2], which tests whether the correlation between two variables is significant. The *Correlation Coefficient* (CC) between two variables is comprised between −1 and 1. A CC of −1 indicates a strong negative correlation, i.e., every time x increases, y decreases. A CC of 0 indicates no correlation between the two variables, while 1 indicates a strong positive correlation, i.e., when x increases, y also increases.

Results. 71.3% (62/87) of the projects increase PR submissions after adopting CI. Figure 5 shows the distributions of the number of submitted, merged and delivered PRs per release for the studied projects. We observe that projects tend to submit a median of 42.6 PRs per release *after* adopting CI, while a median of 15.3 PRs *before* adopting CI. A Wilcoxon signed rank test reveals that the



(a) Submitted PRs **(b) Merged PRs**

(c) Delivered PRs

Figure 5: PR submission, merge, and delivery rates per release.

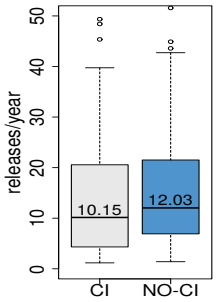


Figure 6: Releases per year before and after CI.

increase in the number of PR submissions is statistically significant ($p - value = 0.0001547$), with a Cliff's delta of 0.332 (*medium* effect-size). We also observe a significant increase in the number of merged PRs per release *after* the adoption of CI ($p - value = 7.897e - 05$, with a *medium* Cliff's delta of 0.347). The number of merged PRs per release increases from 10.4 (median) *before* CI to 27.9 *after* CI. Interestingly, we also observe an increase in the sum of PR code churn per release *after* adopting CI. We obtain a $p - value = 0.002273$ and a Cliff's delta value of 0.27 (small). This significant increase in the PR code churn per release might also explain the increased lifetime of PRs *after* adopting CI. Since more code modifications are performed in the PRs of the releases *after* CI, it may require a longer time to review, merge and deliver such PRs.

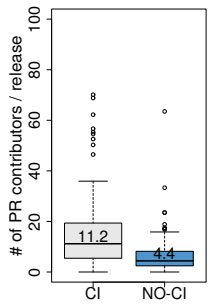


Figure 7: # of PR contributors

Figure 8: PR contributors per release.

After adopting CI projects deliver 3.43 times more PRs per release than before CI. When we turn to PR throughput per release, we find that the number of delivered PRs per release also highly increases *after* the projects adopting CI, it increases from 9.2 before CI to 31.7 *after* CI (see Figure 5c). Furthermore, the increase on the number of delivered PRs per release *after* CI is statistically significant ($p-value = 1.366e-05$, with a Cliff'sŹdelta of 0.3819527, which is considered *medium*).

We do not observe a significant difference of release frequency after adopting CI. A great increase in the PRs submission may be related to an increase in the release frequency after adopting CI. Figure 6 shows the distributions of the number of releases per year *before* and *after* the adoption of CI for each of the studied projects. In the median, projects tend to ship 12.03 releases per year *before* CI, while it drops to 10.15 *after* CI. We obtain a $p - value = 0.146$, which indicates that the release frequency per year *before* and *after* CI are statistically insignificant. Our results suggest that the high increase in the number of delivered PRs per release *after* adopting CI is unlikely to be linked with an increase in the number of releases frequency. We investigate whether the increased number of delivered PRs is due to an increase in the number of contributors *after* the adoption of CI.

We find that 75.9% (66/87) of the studied projects tend to increase the number of PR contributors per release after adopting CI. Figure 8 shows the distributions of the number of contributors per release both *before* and *after* the adoption of CI. The number of PR contributors per release increases from 4.4 (median) *before* CI to 11.2 *after* CI. We observe that the number of PR contributors is statistically significant ($p - value = 2.525e - 06$), with a Cliff's delta of 0.413, which is considered *medium*.

Despite the increase in both the number of delivered PRs per release and in the number of PR contributors per release *after* the projects adopting CI, we did not observe a statistically correlation between these variables. Our results show that the number of delivered PRs per release and the number of PR contributors per release have small positive coefficient correlation of 0.1906346. Furthermore, a Pearson correlation test reveals that this correlation is not statistically significant ($p - value = 0.07695$). Our observations suggest that the higher increase of delivered PRs *after* the adoption of CI is not tightly related to the increase in the number of releases or contributors. Such increase in the number of delivered PRs might be due to the quicker feedback and automated tests that are provided by CI. A qualitative study with developers may shed more light upon this matter. We further discuss this issue in Section 6.

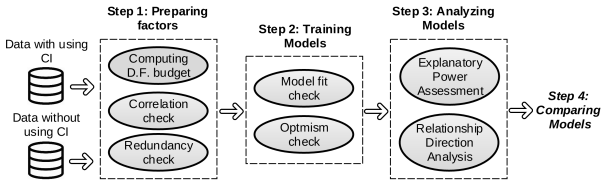


Figure 9: Overview of the process that we use to build our explanatory models.

After adopting CI, projects deliver 3.43 times more PRs per release than before CI. The large increase on the PR submission, merge and delivery rate after CI is a possible reason as to why projects may deliver PRs more quickly before adopting CI.

RQ3 - What factors impact the delivery time *after* adopting continuous integration?

Motivation. In *RQ*1 and *RQ*2, we study the impact of adopting CI on the delivery time of merged PRs. We observe that in only 51.3%, of the projects, PRs are delivered more quickly *after* the adoption of CI. One possible reason for that, is the increase of PR submissions *after* the adoption of CI. Nevertheless, it is also important to understand what are the characteristics of the delivery time of merged PRs *before* and *after* the adoption of CI. Such information may help project managers to track and avoid a high delivery time.

Approach. We use multiple regression modeling (*Ordinary Least Squares*) to describe the relationship between X (i.e., the set of explanatory variables, e.g., *churn*, *description length*), and the response variable Y, i.e., the *delivery time* of merged PRs in terms of days. In addition, we control covariates that might influence the results. For each project, we build two explanatory models, one using the PRs data *before* CI, and another using PRs data *after* CI. Table 2 shows the definition and rationale for each variable of our explanatory models. We model our response variable Y as the time between when a PR was merged and when that PR was released (i.e., delivery time).

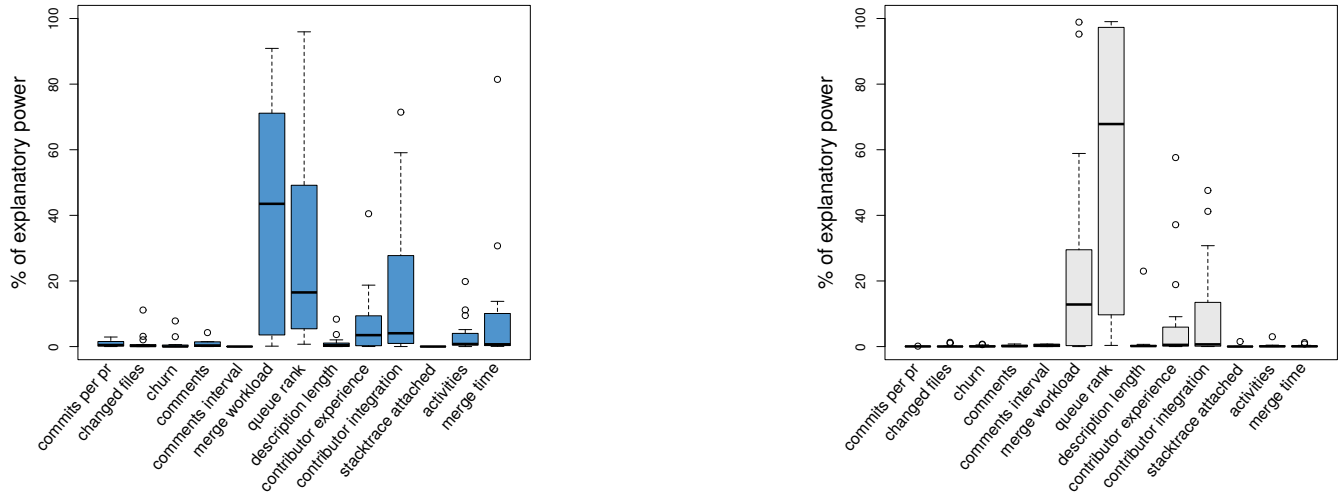
We follow the guidelines of Harrell Jr. [18] for fitting linear models. Figure 9 shows an overview of the process that we use to build our statistical models. In the first step, we estimate the budget (degrees of freedom, or simply D.F.) that we can spend in our models. Then, we account for similarity and correlation of our explanatory variables by using the Spearman rank correlation test. This test checks the metrics that are highly correlated with another one in order to remove them before building our explanatory models. In the second step, we assess the fit of our linear regression models using the R^2. The R^2 corresponds the proportion of the variability in Y that can be explained by using X. In general, it is a challenge to determine what is a good R^2 value, since it depends on the nature of the problem that is being investigated [20]. In this study, we consider for our analyses, only the models that achieve R^2 values higher than 0.5. In other words, we ensure that at least 50% of the variability of our data is explained by our models. We analyze 34

Table 2: Metrics that are used in our explanatory models.

Dimension	Atributes	Type	Definition (d) | Rationale (r)
Resolver	Contributor Experience	Numeric	**d:** The number of previously released PRs that were submitted by the contributor of a particular PR. We consider the author of the PR to be its contributor.
			r: The greater the experience and participation of a user within a specific open source project, the greater his/her chance of having his/her PR reviewed and integrated into codebase of such project by its core integrators [31].
	Contributor Integration	Numeric	**d:** The average in days of the previously released PRs that were submitted by a particular contributor.
			r: If a particular contributor usually submit PRs that are merged and released quickly, his/her future PR might be merged and released quickly as well [9].
Pull Request	Stack Trace Attached	Boolean	**d:** We verify if the PR report has an stack trace attached in its description.
			r: If the PR provide a bug fix, a stack trace attached may provide useful information regarding the causes of the bug and the importance of the submitted code, which may quicken the merge of the PR and its delivery in a release of the project [29].
	Description Size	Numeric	**d:** The number of characters in the body (description) of a PR.
			r: PRs that are well described might be more easier to merge and release than PRs that are more difficult to understand [9].
Project	Queue Rank	Numeric	**d:** The number that represents the moment when a PR is merged compared to other merged PRs in the release cycle. For example, in a queue that contains 100 PRs, the first merged PR has position 1, while the last merged PR has position 100.
			r: A PR with a high *queue rank* is a recently merged PR. A merged PR might be released faster/slower depending of its queue position [9].
	Merge Workload	Numeric	**d:** The amount of PRs that were created and still waiting to be merged by a core integrator at the moment at which a specific PR is submitted.
			r: A PR might be released faster/slower depending of the amount of submitted PRs waiting to be merged. The higher the amount of created PRs waiting to be analyzed and merged, the greater the workload of the contributors to analyze these PRs, which may impact the delivery time of them.
Process	Number of Impacted Files	Numeric	**d:** The number of files linked to a PR submission.
			r: The delivery time might be related to the high number of files of a PR, because more effort must be spent to integrate it [21].
	Churn	Numeric	**d:** The number of added lines plus the number of deleted lines to a PR.
			r: A higher churn suggests that a great amount of work might be required to verify and integrate the code contribution sent by means of PR [21, 26].
	Merge Time	Numeric	**d:** Number of days between the submission and merge of a PR.
			r: If a PR is merged quickly, it is more likely to be released faster.
	Number of Activities	Numeric	**d:** An activity is an entry in the PR' history.
			r: A high number of activities might indicate that much work was required to turn the PR acceptable, which may impact the integration of such PR into a release [21].
	Number of Comments	Numeric	**d:** The number of comments of a PR.
			r: A high number of comments might indicate the importance of a PR or the difficulty to understand it [13], which may impact its delivery time [21].
	Interval of Comments	Numeric	**d:** The sum of the time intervals (days) between comments divided by the total number of comments of a PR.
			r: A short *interval of comments* indicates the discussion was held with priority, which suggest that the PR is important, thus, the PR might be delivered faster [9].
	Commits per PR	Numeric	**d:** Number of commits per PR.
			r: The higher the number of commits in a PR, the greater the amount of contribution to be analyzed by the project integrators, which might impact the delivery time of the PR.

models in total — 18 using PRs data *before* CI, and 16 using data *after* CI. While R^2 gives an indication of how much variability may be explained by our models, this metric may also be very dependent of the specific data to which our models were fitted, i.e., overfitted [24]. Therefore, in the next step, we assess how stable are our models by computing the *optimism-reduced* R^2. The optimism of the R^2 is computed as follows: (i) we count the DF that are spent to fit the original model, then we select a bootstrap sample to fit another model with the same DF of the original model; (ii) we compute the R^2 of both models that were fitted for the bootstrap and the original samples. The optimism is the difference of the R^2 statistics of the original and bootstrap samples. In our analyses, we fit models for 1,000 bootstrap samples and the average optimism is computed. The R^2 *optimism-reduced* is calculated by subtracting the average optimism from the initial R^2 estimate. Finally, we use the Wald X^2 maximum likelihood test to evaluate the impact that each explanatory variable has on the models that we fit. The larger the X^2 value for a variable, the larger the impact of such a variable on the models to explain the variance of the response variable [9]. Then, we analyze the relationship that the most impactful variables share with the response variable (delivery time). To do this, we use the *Predict* function of the *rms* package of the R language to plot the change in the delivery time against the change in each impactful variable while holding the other variables constant at their median values.

Results. Our models achieve a median R^2 of 0.64 using pull request data before CI, while achieving 0.67 after CI. Moreover, the *median* bootstrap-calculated optimism is less than 0.069 for both



(a) Explanatory power of variables *before* adopting CI.

(b) Explanatory power of variables *after* adopting CI.

Figure 10: Distributions of the *explanatory power* of each variable of our models.

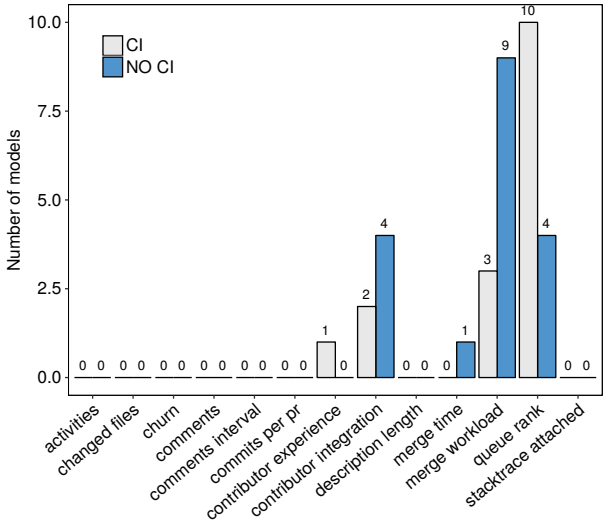


Figure 11: The number of models per most influential variables.

set of R^2 of our set of models.[6] These results suggest that our models are stable enough to perform the statistical inferences that follow.

The "merge workload" is the most influential attribute in the models that we fit for data before the adoption of CI. *Merge workload* represents the number of PRs competing to be merged (see Table 2). Figure 10 shows the distributions of the explanatory power of each variable of our models. The higher the median of the explanatory power for a variable, the higher the influence that such a variable has on the delivery time of PRs. We observe that *merge workload* has the largest influence on our models to explain delivery time *before* the adoption of CI. Our models reveal that the higher the merge workload of the project, the higher the delivery time of their PRs. Figure 11 shows each explanatory variable and the number of models for which these variables are the most influential. Indeed, *merge workload* is the most influential variable in (9/18) of models that we fit using data *before* CI. Figure 12 shows the relationship that the most influential variables of our models share with delivery time. The relationship between *merge workload* and delivery time is shown in Figure 12a. We choose 3 out of the 34 models with the higher R^2 to plot the relationships. Nevertheless, the rest of our models produce the same trend.[7]

The "queue rank" variable is the most influential variable in our models using data after the adoption of CI. *Queue rank* is the moment when a PR is merged with respect to other merged PRs in the release cycle. Figure 12b shows the relationship that *queue rank* shares with delivery time. Our models reveal that merged PRs have a lower delivery time when they are merged more recently in the release cycle. In addition, *contributor integration* is the third most influential variable in our models for both data *before* and *after* the adoption of CI. Contributor integration represents the average in days of the previously delivered PRs that were submitted by a particular contributor. Our models also reveal that if a contributor has his/her prior submitted PRs delivered quickly, his/her next submitted PRs tend to be delivered more quickly (Figure 12c).

Our models suggest that "merge workload" is the most influential variable to model the delivery time of merged PRs, before the adoption of CI. Additionally, our models show that, after CI, merged PRs have a lower delivery time when they are merged more recently in the release cycle.

[6] https://prdeliverydelay.GitHub.io/#rq3-r-squared-and-optimism

[7] https://prdeliverydelay.GitHub.io/#rq3-variables-explanatory-power

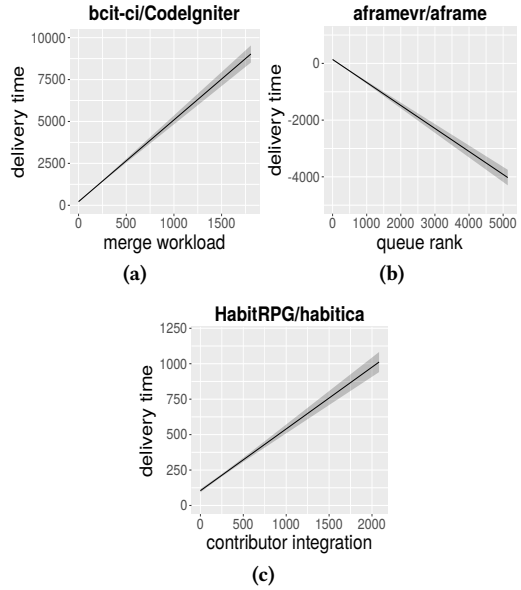


Figure 12: The relationship between the most influential variables and delivery time.

5 DISCUSSION

In this section, we outline the implications of our quantitative observations for both research and practice in the software engineering field.

Continuous integration is not a silver bullet. Through our quantitative analyses, we observe that continuous integration does not always reduce the time for delivering merged PRs (i.e., new software functionalities) to end users. In fact, by analyzing 87 projects, we observe that only 51% of the projects deliver merged PRs more quickly after adopting CI (Section 4 - RQ1). If the decision to adopt CI is mostly driven by the allure of quickening the delivery of merged PRS [23], such a decision must be more carefully considered by development teams. Finally, previous research suggest that the adoption of CI increases the release frequency of a software project [19]. However, we did not observe such an increase in our quantitative analyses (Section 4 - RQ2).

Does CI stimulate more contributions? We observe that the adoption of CI is associated with a higher number of contributors, PR submissions, and a higher sum of PR churn per release (Section 4 - RQ2). Future research should be invested to better understand whether the adoption of CI leads projects to receiving more contributions when compared projects that did not adopt CI in their life-cycles.

There exists hope for later merged PRs. We observed that, after the adoption of CI, our studied projects tend to deliver the last merged PRs more quickly (Section 4 - RQ3). Hence, contributors should not be discouraged to wok on advancing their PRs integration (e.g., engaging code reviewers) despite the late stage of a release cycle. Finally, we observe that contributors whose previous submitted PRs were merged and delivered quickly, are also likely to have their future PRs delivered quickly. Hence, we recommend that the first PR submissions of a new contributor should be carefully crafted in order to maintain a successful track record in their projects (so that their future PRs are delivered more quickly).

6 THREATS TO THE VALIDITY

In this section, we discuss the threats to the validity of our study.

Construct Validity. The construct threats to validity are concerned with errors caused by the methods that we use to collect our data. We use the GitHub API to develop tools to collect our data. We also develop tools to link PRs to their respective releases. Bugs in these tools may influence our results. However, we use subsamples of the studied projects to carefully assess our tools' outcomes, which produced consistent results.

Internal Validity. Internal threats are concerned with the ability to draw conclusions from the relationship between the dependent variable (the delivery time of merged PRs) and independent variables (e.g., release commits and queue rank).

The method that we use to link PRs to releases may not match the actual number of delivered PRs per release. For instance, if a version control system of a project have the following release tags *v1.0, v2.0, no-ver and v3.0*, we remove the *no-ver* tag. If there are PRs associated with the *no-ver release*, such PRs will be associated to the release *v3.0*. However, only 5.36% (403/7519) of our studied releases are included in this case.

With respect to our explanatory models, the predictors that we use in our models are not exhaustive. Although our models achieve sound R^2 values, other variables may be used to improve performance (e.g., a *boolean* indicating whether a PR is associated with an issue report and another *boolean* that verifies whether a PR was submitted by a core developer or an external contributor). Nevertheless, our set of predictors should be approached as a preliminary set that can be easily computed rather than a final solution.

External Validity. External threats are concerned with the extent to which we can generalize our results [27]. In this work we analyzed 162,653 PRs of 87 popular open source projects from GitHub. All projects adopt the most popular CI server on GitHub, i.e., Travis-CI. We recognize that we cannot generalize our results to any other projects with similar or different settings (e.g., private software projects). Nevertheless, in order to achieve more generalizable results, replication of our study in different settings is required. For replication purposes, we publicize our datasets and results to the interested researcher.[8]

7 RELATED WORK

In this section, we situate our study with respect to prior work that analyze the impact of adopting CI in open source projects.

Despite the wide adoption of *Agile Release Engineering* (ARE) practices (i.e., continuous integration, rapid releases, continuous delivery and continuous deployment), there is still a lack of empirical studies that investigate the impact that these practices have on the software development activities, i.e., in terms of productivity and quality. Through a systematic literature review, Karvonen *et al.* [22] analyzed 619 papers and selected 71 primary studies that are related to ARE practices. They found that only 8 out of the 71 primary studies empirically investigate CI. Karnoven *et al.* highlights

[8] https://prdeliverydelay.GitHub.io/#datasets

that empirical research in this field is highly necessary to better understand the impact of adopting CI on software development.

Hilton *et al.* [19] analyzed 34,544 open source projects from GitHub and surveyed 442 developers. The authors found that 70% of the most popular GitHub projects use CI. The authors found that CI helps projects to release more oftenly and that the CI build status may lead to a faster integration of PRs. Differently for Hilton *et al.*, we quantitatively observe that CI do not lead to an increase in the release frequency. Instead, more PRs are integrated into releases after the adoption of CI.

Vasilescu *et al.* [33] studied the usage of Travis-CI in a sample of 223 GitHub projects written in Ruby, Python and Java. They found that the majority of projects (92.3%) are configured to use Travis-CI, but less than half actually use it. In a follow up research, Vasilescu *et al.* [34] investigated the productivity and quality of 246 GitHub projects that use CI. They found that projects that use CI merge PRs more quickly when they are submitted by core developers. Also, core developers find significantly more bugs when using CI. We use a similar approach as used by Vasilescu *et al.* [34] to identify projects that use Travis-CI. We also analyze the merge time of PRs and find that the majority of the studied projects merge PRs more quickly *before* CI. In addition, we also observe that the number of merged PRs per release is higher *after* adopting CI for most of the projects.

Regarding the acceptance and latency of PRs in CI, Yu *et al.* [39] used regression models in a sample of 40 GitHub projects that use Travis-CI. The authors found that the likelihood of rejection of a PR increase by 89,6% when the PR breaks the build. The results also show that the more succinct a PR is, the greater the probability that such a PR is reviewed and merged earlier. We complement the prior work by analyzing the most influential factors that impact the delivery time of merged PRs *before* and *after* the adoption of CI.

Zhao et al [40] conducted an empirical study to investigate the transition to Travis-CI in a large sample of GitHub open-source projects. They quantitatively compared the CI transition in these projects using metrics such as commit frequency, code churn, pull request closing, and issue closing. In addition, they conducted a survey with a sample of developers of those projects. They used a set of three questions related to the adoption of Travis and CI. They also asked how their development process was adapted to accommodate the transition to CI. The main results of their study are: (i) a small increase in the number of merged commits after CI adoption; (ii) a statistically significant decreasing in the number of merge commit churn; (iii) a moderate increase in the number of issues closed after CI adoption; and (iv) a stationary behavior in the number of closed pull requests as well as a longer time to close PRs after the CI Adoption. As opposed to Zhao et al [40], our study focuses on the analysis of delivered pull requests. We find that for 54% of projects, the submitted PRs experience a longer lifetime after the adoption of Travis-CI. Moreover, we observe that PRs are delivered 3.43 times more per release *after* the adoption of CI.

Other work has studied the delivery time of new features, enhancements, and bug fixes [4, 5, 8, 9]. Costa *et al.* [9] investigated the impact of switching from traditional releases to rapid releases on the delivery time of fixed issues of the Firefox project. They used predictive models to discover which factors significantly impact the delivery time of issues in each release strategy. Differently from prior work, our study focuses on the impact of adopting CI on the time-to-delivery of merged PRs.

8 CONCLUSION

We perform an empirical study that investigates the impact of adopting CI on the time-to-delivery of merged PRs. We use 162,653 PRs of 87 GitHub projects to explore the factors that affect (and improve) the time to deliver PRs. In this study, we observe the following:

- In 54% (47/87) of the projects, submitted pull requests experience a longer lifetime *after* the adoption of continuous integration. Furthermore, PRs are merged faster *before* adopting CI in 71.3% (63/87) of the studied projects.
- One possible reason for the faster delivery of PRs *before* CI, is the considerable increase on the number of PR submissions and the release churn *after* the adoption of CI. 71.3% (62/87) of the projects that adopt CI increase the rate of PR submissions.
- The main factor that affect the time-to-delivery of merged PRs *before* the adoption of CI is the merge workload, which represents the number of submitted PRs competing for being merged. The models also show that *queue rank* (i.e., the time at which a PR is merged during a release cycle) also have the strongest impact on the time to deliver merged PRs *after* the adoption of CI.

Open source projects that plan adopt CI should be aware that the adoption of CI will not necessarily deliver merged PRs more quickly. On the other hand, as the pull-based development can attract the interest of external contributors, and hence, increase the projects workload, CI may help in other aspects, e.g., delivering more functionalities to end users (see RQ2).

Our work is the first to explore the impact of CI on the time-to-delivery of PRs in software development. However, more work is necessary to better understand and improve the activities of integrating and delivering PRs. Further research on the field can investigate additional projects, programming languages, and build tools, in order to help developers and project managers to be aware of the estimated time-to-delivery of new PRs based on their characteristics. Finally, replications of this study in different settings (e.g. private initiative) is necessary.

ACKNOWLEDGMENTS

We would like to thank Dr. Ahmed E. Hassan, from Queen's University, Canada and Dr. Shane McIntosh, from McGill University, Canada for their valuable feedback in the final stages of this work. Additionally, this work is partially supported by the National Institute of Science and Technology for Software Engineering (INES), CNPq grant 465614/2014-0, and National Council for Scientific and Technological Development, CNPq grants 459717/2014-6 and 312044/2015-1.

REFERENCES

[1] Kent Beck. 2000. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional.

[2] DJ Best and DE Roberts. 1975. Algorithm AS 89: the upper tail probabilities of Spearman's rho. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 24, 3 (1975), 377–379.

[3] Jiyao Chen, Richard R Reilly, and Gary S Lynn. 2005. The impacts of speed-to-market on new product success: the moderating effects of uncertainty. *IEEE Trans. Eng. Manage.* 52, 2 (2005), 199–212.

[4] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2015. Predicting Delays in Software Projects Using Networked Classification (T). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 353–364.

[5] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. 2017. Predicting the delay of issues with due dates in software projects. *Empirical Software Engineering Journal* (2017), 1–41.

[6] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114, 3 (1993), 494.

[7] Kevin Crowston, Hala Annabi, and James Howison. 2003. Defining open source software project success. *ICIS 2003 Proceedings* (2003), 28.

[8] Daniel Alencar da Costa, Surafel Lemma Abebe, Shane McIntosh, Uirá Kulesza, and Ahmed E Hassan. 2014. An empirical study of delays in the integration of addressed issues. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 281–290.

[9] Daniel Alencar da Costa, Shane McIntosh, Uirá Kulesza, and Ahmed E. Hassan. 2016. The Impact of Switching to a Rapid Release Cycle on the Integration Delay of Addressed Issues: An Empirical Study of the Mozilla Firefox Project. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 374–385.

[10] Adam Debbiche, Mikael Dienér, and Richard Berntsson Svensson. 2014. Challenges when adopting continuous integration: A case study. In *International Conference on Product-Focused Software Process Improvement*. Springer, 17–32.

[11] Paul Duvall, Stephen M Matyas, and Andrew Glover. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk (The Addison-Wesley Signature Series)*. Addison-Wesley Professional.

[12] Martin Fowler and Matthew Foemmel. 2006. Continuous integration. *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* (2006), 122.

[13] Emanuel Giger, Martin Pinzger, and Harald Gall. 2010. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*. ACM, 52–56.

[14] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*. 345–355.

[15] Georgios Gousios and Diomidis Spinellis. 2012. GHTorrent: GitHub's data from a firehose. In *Mining software repositories (msr), 2012 9th ieee working conference on*. 12–21.

[16] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. 358–368.

[17] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie Van Deursen. 2015. Work practices and challenges in pull-based development: the integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 358–368.

[18] Frank Harrell. 2015. *Regression modeling strategies: with applications to linear models, logistic and ordinal regression, and survival analysis*. Springer.

[19] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*.

[20] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2014. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated.

[21] Yujuan Jiang, Bram Adams, and Daniel M German. 2013. Will my patch make it? and how fast? case study on the linux kernel. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*. IEEE, 101–110.

[22] Teemu Karvonen, Woubshet Behutiye, Markku Oivo, and Pasi Kuvaja. 2017. Systematic literature review on the impacts of agile release engineering practices. *Information and Software Technology* 86 (2017), 87 – 100.

[23] Eero Laukkanen, Maria Paasivaara, and Teemu Arvonen. 2015. Stakeholder Perceptions of the Adoption of Continuous Integration – A Case Study. In *Proceedings of the 2015 Agile Conference (AGILE '15)*. IEEE Computer Society, 11–20.

[24] Shane Mcintosh, Yasutaka Kamei, Bram Adams, and Ahmed E. Hassan. 2016. An Empirical Study of the Impact of Modern Code Review Practices on Software Quality. *Empirical Softw. Engg.* 21, 5 (2016), 2146–2189.

[25] Mathias Meyer. 2014. Continuous integration and its tools. *IEEE Softw.* 31, 3 (2014), 14–16.

[26] Nachiappan Nagappan and Thomas Ball. 2005. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*. IEEE, 284–292.

[27] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. 2000. Empirical Studies of Software Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering (ICSE '00)*. ACM, 345–355.

[28] J. Romano, J.D. Kromrey, J. Coraggio, and J. Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen'sd for evaluating group differences on the NSSE and other surveys?. In *annual meeting of the Florida Association of Institutional Research*. 1–3.

[29] Adrian Schroter, Adrian Schröter, Nicolas Bettenburg, and Rahul Premraj. 2010. Do stack traces help developers fix bugs?. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 118–121.

[30] Ken Schwaber. 1997. SCRUM Development Process. In *Business Object Design and Implementation*, Dr Jeff Sutherland, Cory Casanave, Joaquin Miller, Dr Philip Patel, and Glenn Hollowell (Eds.). Springer London, 117–134.

[31] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M Ibrahim, Masao Ohira, Bram Adams, Ahmed E Hassan, and Ken-ichi Matsumoto. 2010. Predicting reopened bugs: A case study on the eclipse project. In *Reverse Engineering (WCRE), 2010 17th Working Conference on*. IEEE, 249–258.

[32] Daniel Ståhl and Jan Bosch. 2014. Modeling Continuous Integration Practice Differences in Industry Software Development. *J. Syst. Softw.* 87 (2014), 48–59.

[33] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ van den Brand. 2014. Continuous integration in a social-coding world: Empirical evidence from GitHub. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 401–405.

[34] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*.

[35] Daniel S Wilks. 2011. *Statistical methods in the atmospheric sciences*. Vol. 100. Academic press.

[36] David F. Williamson, Robert A. Parker, and Juliette S. Kendrick. 1989. The box plot: A simple visual method to interpret data. *Annals of Internal Medicine* 110 (1989), 916–921.

[37] Krzysztof Wnuk, Tony Gorschek, and Showayb Zahda. 2013. Obsolete software requirements. *Information and Software Technology* 55, 6 (2013), 921–940.

[38] Claes Wohlin, Min Xie, and Magnus Ahlgren. 1995. Reducing time to market through optimization with respect to soft factors. In *The Engineering Management Conference*. 116–121.

[39] Yue Yu, Gang Yin, Tao Wang, Cheng Yang, and Huaimin Wang. 2016. Determinants of pull-based development in the context of continuous integration. *Sci. China Inf. Sci.* 59, 8 (2016).

[40] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 60–71.