(No) Influence of Continuous Integration on the Commit Activity in GitHub Projects

Sebastian Baltes
University of Trier
Trier, Germany
research@sbaltes.com

Jascha Knack, Daniel
Anastasiou, Ralf Tymann
University of Trier
Trier, Germany

Stephan Diehl
University of Trier
Trier, Germany
diehl@uni-trier.de

ABSTRACT

A core goal of Continuous Integration (CI) is to make small incremental changes to software projects, which are integrated frequently into a mainline repository or branch. This paper presents an empirical study that investigates if developers adjust their commit activity towards the above-mentioned goal after projects start using CI. We analyzed the commit and merge activity in 93 GitHub projects that introduced the hosted CI system *Travis CI*, but have previously been developed for at least one year before introducing CI. In our analysis, we only found one non-negligible effect, an increased merge ratio, meaning that there were more merging commits in relation to all commits after the projects started using *Travis CI*. This effect has also been reported in related work. However, we observed the same effect in a random sample of 60 GitHub projects not using CI. Thus, it is unlikely that the effect is caused by the introduction of CI alone. We conclude that: (1) in our sample of projects, the introduction of CI did not lead to major changes in developers' commit activity, and (2) it is important to compare the commit activity to a baseline before attributing an effect to a treatment that may not be the cause for the observed effect.

CCS CONCEPTS

• **Software and its engineering** → **Software development techniques**;

KEYWORDS

continuous integration, open source software, commit activity, mining software repositories

1 INTRODUCTION

Continuous Integration (CI) is a software engineering practice where developers frequently integrate their work into a common "mainline" repository or branch [10, 25]. After changes have been committed to this repository or branch, a CI system automatically builds and tests the software. The approach has originally been proposed by Grady Booch [8] and became popular after being promoted as one of the Extreme Programming (XP) practices [5]. Many software projects on GitHub use hosted CI services such as *Travis CI* or *CloudBees* [15]. The specific CI process may differ between projects (e.g., when is a build triggered, what is considered to be a broken build, etc.) [32], but a core goal of CI is to work with small increments and to integrate them frequently into the mainline branch [25]. There has been research on different aspects of using CI in GitHub projects (see Section 5) and also one study investigating if the introduction of CI actually leads to a different commit activity in terms of smaller, but more frequent commits [37]. That study found an increasing number of merge commits after the introduction of CI, but a large variation regarding the "commit small" guideline. We assessed the CI guidelines with a different methodological lens and also found an increased merge ratio, that is more merging commits in relation to all commits after the projects started using CI. However, we observed the same effect in a random sample of projects not using CI and conclude that it is unlikely that the effect is caused by the introduction of CI alone.

2 RESEARCH DESIGN

The overall goal of our research was to analyze the impact of the introduction of continuous integration on the commit and merge activity in open source software projects. We selected projects hosted on GitHub that employ the CI system *Travis CI*. We identified such projects using the *TravisTorrent data set* (January 11, 2017) [6] and the *GHTorrent* Big Query data set (February 14, 2017) [11, 12]. We only considered projects that:
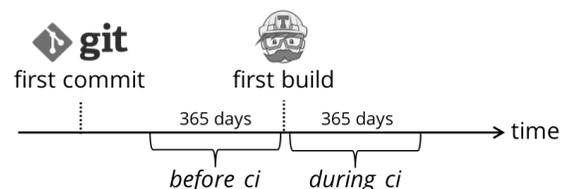
Figure 1: Partitioning of commits into two time frames for each analyzed project. We only analyzed the activity one year before and after the first build.

(1) were active for at least one year (365 days) before the first build with *Travis CI* (see *before_ci* in Figure 1),
(2) used *Travis CI* at least for one year (see *during_ci* in Figure 1),
(3) had commit or merge activity on the default branch in both of these phases, and
(4) used the default branch to trigger builds.

The motivation for restrictions (1) and (2) was to be able to compare a considerable amount of commit activity before and after the first build. To further exclude projects that use a different branch than the default branch as "mainline", we added restrictions (3) and (4).

We utilized the *GHTorrent* Big Query data set to identify the time frames *before_ci* and *during_ci* for the projects in the *TravisTorrent* data set. Of all projects in the data set, 544 satisfied restrictions (1) and (2). Of these projects, 366 were Ruby projects and 178 were Java projects. The mean time span *before_ci* was 2.9 years (*SD*=2.1, *Mdn*=2.2), the mean time span *during_ci* was 3.2 years (*SD*=1.1, *Mdn*=3.2). To compare two time frames of equal size, we restricted our analysis to the activity one year before and after the first build.

Our *units of observation* were the commits in the selected projects. We only considered changes to Java or Ruby source code files, because we were interested in the actual development activity, not in changes to binary files like images or PDF documents. We cloned all 544 project repos and extracted the version history for all branches with a tool we developed [1]. For each repo and branch, we created one log file with all regular commits and one log file with all merges. From those log files, we then extracted metadata about the commits and stored this data in CSV files using a second tool [2]. Combining the commit data and *TravisTorrent*, we applied restrictions (3), and (4). Moreover, we excluded projects where the first commit activity happened more than seven days before the project creation date according to *GHTorrent*. The motivation for this additional filtering step was that projects which started outside of GitHub and have later been ported could introduce a bias, because the commit activity may differ if features such as GitHub pull requests are not available [13, 14, 33]. This resulted in a sample of 113 projects (89 Ruby and 24 Java projects).

As *units of analysis*, we considered the commits themselves and the projects. The measures we compared for these units were the *commit rate*, the *commit size*, and the *merge ratio*, which we define below.

In the following, let $\mathcal{C}$ be the set of all commits in all projects we collected as described above. Further, let $\mathcal{P}_{\mathcal{C}}$ be the power set of $\mathcal{C}$. The *commit frequency* is the number of commits in a certain time span. We chose a time span of one week to adjust for the varying activity between working days and weekend. The partition $C^w = C^{w_1} \dots C^{w_n}$ divides a set of commits C into one subset for each week, beginning with the week of the first commit (w_1) until the week of the last commit (w_n) in C. For our analysis, we ignored w_1 or w_n if they did not contain data about a whole week.

Considering only non-merging commits, 52% of those weeks were inactive, meaning there was no commit activity. For the merging commits, there were 71% inactive weeks. A possible reason for this phenomenon could be developers working intensively on an open source project for a few days and then focusing on something else, for example other closed- or open-source projects. We consider an investigation of activity patterns in open source software projects to be an interesting direction for future work. In our analysis, we focused on active weeks weeks and excluded weeks without any commit activity.

In a last filtering step, we excluded projects without merges in either of the two phases or only data for incomplete weeks. This resulted in a final set of 93 projects. We provide the project list and the scripts and data used for the filtering process as supplementary material [3, 4]. With the terminology introduced above, we can now define the *median commit rate* based on a weekly partition of a set of commits C:

Definition 2.1 (Commit Rate). Let $C \in \mathcal{P}_{\mathcal{C}}$ be a set of commits and $C^w = C^{w_1} \dots C^{w_n}$ be a weekly partition of C. We define the *median commit rate* $\overline{c_w} \colon \mathcal{P}_{\mathcal{C}} \to \mathbb{R}_0^+$ as:

$$\overline{c_w}(C) = median(|C^{w_i}|), \ i \in \{1, \dots, n\}$$

Please not that, since we focus on active weeks in our analysis, a project with ten commits in one week and no commits in the nine following weeks has a higher commit rate than a project with one commit per week spread over 10 weeks. Beside the commit rate, we define two measures to describe the *change size of a commit*. In the following, $n_f(c)$ denotes the number of source code files changed by a commit $c \in C$, and $n_l(c)$ denotes the number of lines changed in the source code files modified by this commit. We define the code churn $n_l(c)$ as $l_c^+ + l_c^-$, where l_c^+ is the sum of all lines added to the modified files and l_c^- is sum of all lines deleted from those files. Both l_c^+ and l_c^- are based on the line-based diffs of the modified files. Please note that this measure overestimates the change size in case existing lines are modified, e.g., a change to one existing line is represented as one deleted and one added line.

Definition 2.2 (Change Size). For a commit $c \in \mathcal{C}$, we define the *change breadth* $b \colon \mathcal{C} \to \mathbb{R}_0^+$ and the *change depth* $d \colon \mathcal{C} \to \mathbb{R}_0^+$ as:

$$b(c) = n_f(c) \quad and \quad d(c) = \frac{n_l(c)}{n_f(c)}$$

For a set $C \in \mathcal{P}_{\mathcal{C}}$ of commits, we define the *median change breadth* $\overline{b}(C)$ as:

$$\overline{b}(C) = median(b(c)), \ c \in C$$

For a partition $C^\phi = \{C^1 \dots C^n\}$ of C, we define the *median change breadth* $\overline{b}(C^\phi)$ as:

$$\overline{b}(C^\phi) = median(\overline{b}(C^i)), \ i \in \{1, \dots, n\}$$

The *median change depths* $\overline{d}(C)$ and $\overline{d}(C^\phi)$ are defined analogously.

The last measure we are going to define is the amount of merging commits in relation to all commits. We use this measure to estimate the branching and merging activity in a certain time span. For all $C \in \mathcal{P}_{\mathcal{C}}$, let $n_m(C)$ denote the number of commits in C that merged other commits.

Definition 2.3 (Merge Ratio). For a set $C \in \mathcal{P}_{\mathcal{C}}$ of commits, we define the *merge ratio* $m \colon \mathcal{P}_{\mathcal{C}} \to [0, 1]$ as:

$$m(C) = \frac{n_m(C)}{|C|}$$

Now that we have precise definitions of *commit rate*, *change size*, and *merge ratio*, we can take different comparison perspectives to describe the commit activity in software projects. We have two parameters for aggregating single commits into sets of commits that we can then compare with our measures. First, we can aggregate commits according to their *origin*, e.g., all commits in the same project or from the same developer. Second, we can aggregate commits according to *time*, here mainly the two time frames before and after the first CI build. Regarding the *origin* of the commits, we define two partitions:

C^p : Partitions C into one set of commits for each project $p \in projects(C)$.

$C^{p,b}$: Partitions C into one set of commits for each branch in each project ($\forall p \in projects(C) \, \forall b \in branches(p)$).

For this paper, we do not consider other partitions such as $C^{p,d}$ for each developer in a project. Regarding the *time*, we define two partitions that correspond to the time frames described above:

C_{before} Contains all commits before the first CI build for each project, but not more than 365 days before the first build.

C_{during} Contains all commits after the first but before the last CI build for each project, but not more than 365 days after the first build.

It is possible to combine the above-mentioned partitions. $C^{p,b}_{before}$, for instance, contains one set of commits for each branch b in each project p, taking only commits into account that were committed before the first build in the corresponding project. For the following analyses, we first partition the sets using the branch partition $C^{p,b}$ and only consider the default ("mainline") branch for each project. This branch is often, but not always, called master. For better readability, we only write C^p instead of $C^{p,b}$, $b = default\_branch(p)$ in the following. We focused on the default branch, because this is usually the branch triggering the CI builds. In fact, according to *TravisTorrent*, 81.8% of all builds of the 93 projects in our final sample were triggered by commits to the default branch. For the following analyses, we take a *project perspective*, i.e., we partition the *before* and *during* sets into the commits for each project, then calculate and compare $\overline{c_w}$, $\overline{b}$, $\overline{d}$, and m for each project.

3 RESULTS

In the following, we test our a priori hypotheses about commit activity changes using the data we collected. We provide the raw data and all analysis scripts as supplementary material [3, 4].

As descriptive statistics, we report median (Mdn), interquartile range (IQR), and mean (M). To test for significant differences, we applied the non-parametric two-sided *Wilcoxon signed rank test* [35] and report the corresponding p-values (p_w, significance level $\alpha = 0.01$). To measure the effect size, we used *Cliff's delta* (δ) [9]. Its values range between +1, when all values of one group are higher than the values of the other group, and −1, when the reverse is true. We also provide the confidence interval of δ at a 95% confidence level (CI_δ). Our interpretation of δ is based on the thresholds described by Romano et al. [31]: *negligible effect* ($|\delta| < 0.147$), *small effect* ($0.147 \leq |\delta| < 0.33$), *medium effect* ($0.33 \leq |\delta| < 0.474$), otherwise *large effect*. Our general assumption was that, with the introduction of CI, the projects shift towards smaller increments

(in form of smaller commits) that are frequently integrated (directly committed or merged) into the default branch.

3.1 Commit Rate

If developers follow the advice to frequently integrate their work into the main branch, the introduction of CI may have an effect on the commit rate:

Hypothesis 1. After the introduction of CI, the commit rate increases.

To test this hypothesis, we compared the following sets:

$$\{\overline{c_w}(C^p_{before})\} \text{ vs. } \{\overline{c_w}(C^p_{during})\}, \, p \in projects(C)$$

For C_{before}, the median commit rate per project was 2.5 commits per week ($IQR = 2.5$, $M = 3.5$); for C_{during} it was 2.0 commits per week ($IQR = 2$, $M = 3.3$). The difference was not significant ($p_w = 0.66$) and negligible ($\delta = -0.02$, $CI_\delta = [-0.18, 0.15]$).

The median commit rate of merging commits was 1 merge per week ($IQR = 1$, $M = 1.5$) in the timespan *before* and also 1 merge per week ($IQR = 1$, $M = 1.5$) in the timespan *during*. The difference was was not significant ($p_w = 0.10$) and negligible ($\delta = 0.09$, $CI_\delta = [-0.06, 0.23]$).

Thus, we failed to reject the null hypothesis that the commit rate decreases or does not change.

3.2 Change Size

As described above, we employed two measures to capture the change size of a commit. We tested both its breadth, i.e., the number of changed source code files, as well as its depth, i.e., the number of changed lines divided by the number of files. Our hypothesis was that, with the introduction of CI, the median size of the changes decreases:

Hypothesis 2. After the introduction of CI, the commit changes become smaller, i.e., they have a lower change breadth and depth.

To test this hypothesis, we compared the following sets:

$$\{\overline{b}(C^p_{before})\} \text{ vs. } \{\overline{b}(C^p_{during})\}, \, p \in projects(C)$$
$$\{\overline{d}(C^p_{before})\} \text{ vs. } \{\overline{d}(C^p_{during})\}, \, p \in projects(C)$$

For C_{before}, the median commit breadth per project was 1 file per commit ($IQR = 1$, $M = 1.8$); for C_{during} it was also 1 file per commit ($IQR = 1$, $M = 1.6$). The difference was not significant ($p_w = 0.05$) and negligible ($\delta = -0.07$, $CI_\delta = [-0.21, 0.06]$). The median commit breadth of merged commits was 2 files per merged commit ($IQR = 1$, $M = 2.8$) in the timespan *before* and also 2 files per merged commit ($IQR = 0$, $M = 2.0$) in the timespan *during*. The difference was was not significant ($p_w = 0.04$) and negligible ($\delta = -0.10$, $CI_\delta = [-0.24, 0.06]$).

For C_{before}, the median commit depth per project was 7.8 lines per file ($IQR = 4.8$, $M = 9.9$); for C_{during} it was 7.0 lines per file ($IQR = 4.3$, $M = 8.6$). The difference was significant ($p_w = 0.0008$),

Merge ratio (CI sample)

(a) Merge ratio (number of merging commits in relation to all commits).

Pull request ratio (CI sample)

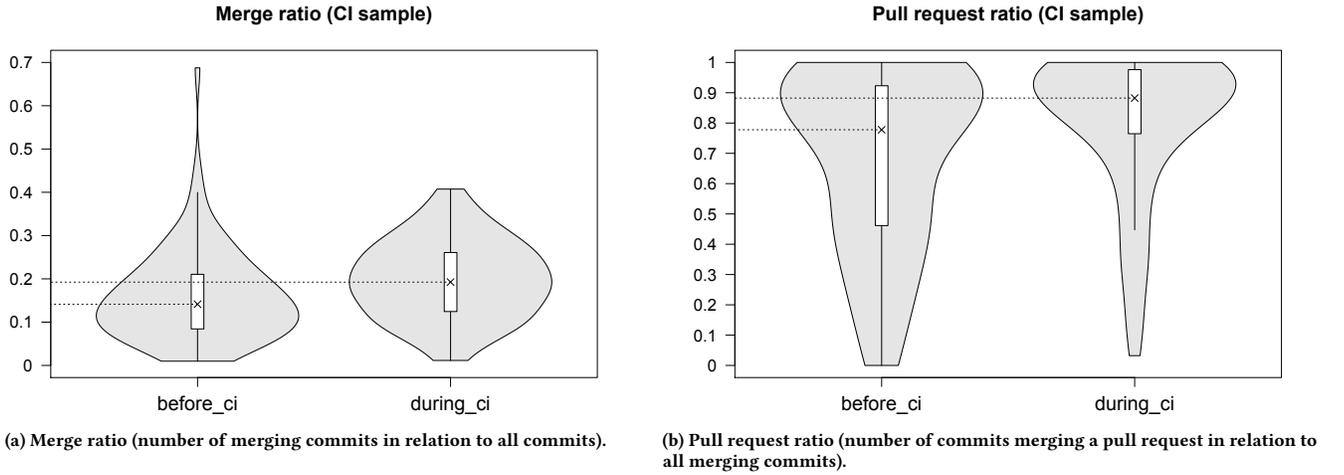(b) Pull request ratio (number of commits merging a pull request in relation to all merging commits).

Figure 2: Merge ratio and pull request ratio of projects ($n = 93$) before and after their first CI build.

but negligible ($\delta = -0.10$, $CI_\delta = [-0.26, 0.06]$). The median commit depth of merged commits was 10.0 lines per file ($IQR = 9.4$, $M = 16.8$) in the timespan *before* and 9.5 lines per file ($IQR = 7.5$, $M = 12.4$) in the timespan *during*. The difference was was not significant ($p_w = 0.10$) and negligible ($\delta = -0.08$, $CI_\delta = [-0.24, 0.09]$).

Thus, we failed to reject the null hypothesis that the change size decreases or does not change.

3.3 Merge Ratio

As a last step, we analyzed if the introduction of CI increases the merge ratio, and in particular the pull request ratio (the number of commits merging a pull request in relation to all merging commits). Our hypothesis was that after the introduction of CI, the amount of direct commits in the default branch decreases, because developers prefer to review the changes before triggering a build. On GitHub, the pull-based software development model has become more and more popular over the past years [13–15, 34]. This development model allows projects to review the changes before merging pull requests into the default branch.

Hypothesis 3. After the introduction of CI, the merge ratio increases.

To test this hypothesis, we compared the following sets:

$$\{m(C_{before}^p)\} \text{ vs. } \{m(C_{during}^p)\}, \ p \in projects(C)$$

For C_{before}, the median merge ratio over all projects was 0.14 ($IQR = 0.13$, $M = 0.16$); for C_{during} it was 0.19 ($IQR = 0.14$, $M = 0.20$). The difference was significant ($p_w < 5.4e{-}6$) and the effect was small ($\delta = 0.28$, $CI_\delta = [0.12, 0.43]$). Thus, we reject the null hypothesis that the change size decreases or does not change. Figure 2a shows violin plots visualizing the difference.

We also compared the pull request ratio: For C_{before}, the median pull request ratio was 0.78 ($IQR = 0.46$, $M = 0.68$); for C_{during} it was

0.88 ($IQR = 0.21$, $M = 0.80$). The difference was significant ($p_w < 7.1e{-}5$) and the effect was small ($\delta = 0.24$, $CI_\delta = [0.08, 0.40]$). Thus, an increased usage of pull requests is likely to be one major factor leading to the increased merge ratio.

3.4 Comparison Sample

To check whether the increased merge ratio can actually be attributed to the introduction of CI, we analyzed how the merge ratio changed in a random sample of GitHub projects that do not use CI. Since this sample should be comparable to the CI project sample, we applied the following constraints, selecting projects that:

(1) have Java or Ruby as their project language
(2) have commit activity for at least two years (730 days)
(3) are engineered software projects (at least 10 watchers)

We applied those constraints to the projects in the *GHTorrent BigQuery data set* (February 06, 2018) [11, 12]. Moreover, we used the same filter as for the CI projects to remove projects with commits more than one week before the GitHub project creation date. Since all projects in the CI sample were "engineered software projects" [26], we applied filter (3) to exclude small "toy" GitHub projects [21]. We applied this popularity filter, using the number of watchers or stargazers, because it has been used in several well-received studies and proved to have a very high precision in selecting engineered projects [26] (almost 0% false positives for a threshold of 10 watchers/stargazers). All of the 93 projects in the CI sample satisfied this constraint.

Of the 8,405 projects that satisfied the constraints, 359 were also in the *TravisTorrent* data set. We excluded those projects and drew a random sample of 800 projects from the remaining 8,046 projects. We retrieved the commit data in the same way as for the CI projects. Then, we determined the date that splits the development activity in those projects into two equally-sized time frames. For the analysis, we considered one year (365 days) of commit activity in the default branch before and after that date. We only considered projects with commit and merge activity in the default branch in

Merge ratio (comparison sample)
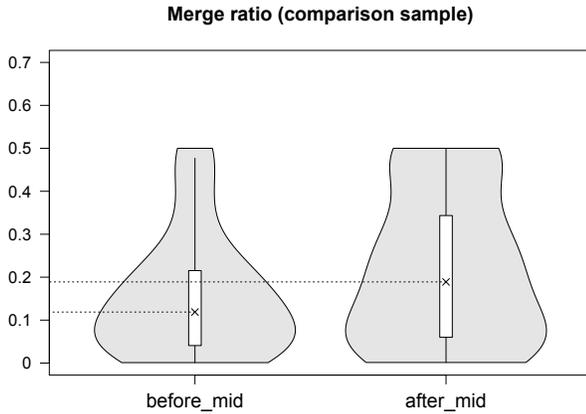
Figure 3: Merge ratio in randomly selected projects not using CI ($n = 60$) before and after median development date.

both time frames (130 projects), which we manually checked for CI configuration files. We removed 70 projects with such configuration files, resulting in 60 projects in which we did not find any indication that they use CI services.

For those projects, we compared the merge ratio in the two time frames to investigate if we can observe a similar increase like in the CI projects. For the commits before the split date, the median merge ratio over all projects was 0.12 ($IQR = 0.17$, $M = 0.16$); for the commits after that date it was 0.19 ($IQR = 0.28$, $M = 0.22$). The difference between the merge ratios of all projects in the two time frames was significant ($p_w = 0.006$) and the effect was small ($\delta = 0.20$, $CI_\delta = [-0.008, 0.40]$). As this effect was similar to the CI sample, we cannot attribute the increased merge ratio exclusively to the introduction of CI.

4 LIMITATIONS AND VERIFIABILITY

The main limitations of our research are the focus on Ruby and Java projects and the fact that we do not know if the introduction of CI would actually be the trigger for the effects we hypothesized. In fact, we are confident that the increased merge ratio cannot be attributed to the introduction of CI alone. Another limitation is the fact that projects may have used other CI tools than *Travis CI* before, meaning that our time span *before_ci* would actually be *during_ci*, but with another CI tool. This could be one reason why we did not observe the expected effects.

To check whether the higher merge ratio after the introduction of CI depends on the number of project contributors, we compared projects with many contributors to projects with fewer contributors. First, we identified all project contributors using the committer and author email addresses in the commit metadata we collected, yielding a median number of 14 contributors per project ($IQR = 13$, $M = 21.8$). We split the CI project sample into two groups, one with 47 small (≤ 14 contributors) and one with 46 large (> 14 contributors) projects. Then, we compared the merge ratios before and after the introduction of CI for those two groups separately. In both groups, the merge ratio was significantly higher after the introduction of CI ($p_w < 0.005$) and the effect was small (small

projects: $\delta = 0.29$, $CI_\delta = [0.06, 0.48]$; large projects: $\delta = 0.28$, $CI_\delta = [0.06, 0.49]$). Thus, we conclude that the increased merge ratio is independent from the number of project contributors.

In our random comparison sample of projects not using CI, we compared the commit activity before and after the median date. Obviously, the effects may be different when choosing another comparison date. Moreover, we used a weekly perspective. Other time frames such as months (see Zhao et al.'s work in Section 5) could lead to different results. Because of the common distinction into five workdays and two days weekend, we think a weekly perspective is reasonable.

We excluded projects with a commit history outside of GitHub (see Section 2). However, projects may have been imported with one large initial commit, without importing their complete Git history. Even if such a huge commit would be part of the timespan *before*, it is unlikely to bias our results, because our comparison is based on median and interquartile range and we applied robust statistical methods [22].

Our focus on open source GitHub projects limits the transferability of our results to closed source commercial projects. In those projects, the number of weeks without commit activity is likely to be much lower than in open source projects, which are often not developed full-time. We consider a comparison of the effects of introducing CI in open source versus closed source projects to be an important direction for future work.

To enable other researcher to verify our results, we provide the raw data and our analysis scripts as supplementary material [3, 4].

5 RELATED WORK

Vasilescu et al. [34] analyzed GitHub projects which introduced CI and found that CI improves the productivity of project teams in terms of merged pull requests. Hilton et al. [18], who analyzed GitHub projects, *Travis CI* builds, and surveyed 442 developers, found that (1) popular projects are more likely to use CI, (2) projects that use CI release more than twice as often as those that do not use CI, and (3) CI builds on the master branch pass more often than on the other branches. Furthermore, Hilton et al. [17] conducted semi-structured interviews with developers and conclude that developers encounter increased complexity, increased time costs, and new security concerns when working with CI.

Zhao et al. [37] investigated the impact of CI on GitHub projects. Their approach was similar to ours as they also considered one year of activity before and after the first CI build. However, they aggregated information for a whole month, opposed to one week in our analysis, and utilized a statistical modeling framework named *regression discontinuity design* [19]. Like us, they observed an increased number of merge commits over time, but as we described above, this trend is not limited to projects adapting CI. Thus, we argue against attributing this effect to the introduction of CI.

Zhao et al. also conclude that the adaption of CI is much more complex than suggested by other studies. For example, they found that more pull requests are being closed after adopting CI, but their analysis suggests that the expected increasing trend over time manifests itself only before adopting CI, afterwards the number of closed pull requests remains relatively stable. This indicates that

more work is needed to investigate if and how projects adapt after interventions such as the introduction of CI.

Memon et al. [24] analyzed data from Google's CI system *TAP* and found that code recently modified by more than three developers is more likely to break the build. Other studies investigated aspects such as the usage of static code analysis tools in CI pipelines [36], the personnel overhead of CI [23], the interplay between non-functional requirements and CI builds [27], the impact of CI on developer attraction and retention [16] or code reviews [28], and factors influencing build failures [7, 20, 29, 30, 38].

6 CONCLUSION AND FUTURE WORK

We presented an empirical study investigating if developers of open-source GitHub projects adjust their commit activity towards smaller but more frequent commits after the introduction of continuous integration (CI). We expected this change, because a core goal of CI is to work with small incremental changes. We analyzed the commit and merge activity in 93 GitHub projects that introduced *Travis CI* and have been developed on GitHub for at least one year before the introduction of CI. The only non-negligible effect we observed was an increased merge ratio, i.e., the number of merging commits in relation to all commits. However, we observed the same effect in a random sample of GitHub projects that do not use CI. Thus, we argue against attributing this effect to the introduction of CI alone. It is more likely that projects use merges more frequently when they grow and mature. Another reason could be the general trend of adopting the pull-based software development model [13–15, 34].

Beside those findings, this paper contributes a precise formalization of different commit activity measures, which we used to test for expectable changes in GitHub projects after the introduction of CI. Our results show that it is important to compare observed changes in commit activity to a baseline (in our case the comparison sample) to prevent attributing those changes to a treatment that may not be the actual cause.

Directions for future work include analyzing the commit activity from additional comparison perspectives, for example by partitioning the commits per developer as mentioned in Section 2. Moreover, one could broaden the research by conducting a more holistic quantitative analysis to identify dominant factors causing the increased merge ratio, for example using multiple regression. Another way to broaden the research would be to continue with a qualitative analysis, asking GitHub developers how they perceive the impact of CI on their projects.

ACKNOWLEDGMENTS

REFERENCES

[1] Sebastian Baltes. 2017. sbaltes/git-log-extractor on GitHub. https://doi.org/10.5281/zenodo.821011

[2] Sebastian Baltes. 2017. sbaltes/git-log-parser on GitHub. https://doi.org/10.5281/zenodo.821020

[3] Sebastian Baltes. 2018. (No) Influence of Continuous Integration on the Commit Activity in GitHub Projects — Supplementary Material. https://doi.org/10.5281/zenodo.1182934

[4] Sebastian Baltes and Jascha Knack. 2018. (No) Influence of Continuous Integration on the Development Activity in GitHub Projects — Dataset. https://doi.org/10.5281/zenodo.1140260

[5] Kent Beck. 2000. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Upper Saddle River, NJ, USA.

[6] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 356–367.

[7] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 447–450.

[8] Grady Booch. 1991. *Object oriented design: With applications.* Benjamin/Cummings Publishing, San Francisco, CA, USA.

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

[10] Martin Fowler. 2006. Continuous Integration. https://www.martinfowler.com/articles/continuousIntegration.html

[11] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *10th International Working Conference on Mining Software Repositories (MSR 2013)*, Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim (Eds.). IEEE, San Francisco, CA, USA, 233–236.

[12] Georgios Gousios. 2018. GHTorrent on the Google cloud. http://ghtorrent.org/gcloud.html

[13] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *36th International Conference on Software Engineering (ICSE 2014)*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, Hyderabad, India, 345–355.

[14] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. 2016. Work Practices and Challenges in Pull-Based Development: The Contributor's Perspective. In *38th International Conference on Software Engineering (ICSE 2016)*, Laura Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, Austin, TX, USA, 285–296.

[15] 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 *37th International Conference on Software Engineering (ICSE 2015)*, Antonia Bertolino, Gerardo Canfora, and Sebastian Elbaum (Eds.). IEEE, Florence, Italy, 358–368.

[16] Yash Gupta, Yusaira Khan, Keheliya Gallaba, and Shane McIntosh. 2017. The impact of the adoption of continuous integration on developer attraction and retention. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 491–494.

[17] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in continuous integration: Assurance, security, and flexibility. In *11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2017)*, Eric Bodden, Wilhelm Schäfer, Arie van Deursen, and Andrea Zisman (Eds.). ACM, Paderborn, Germany, 197–207.

[18] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, Singapore, 426–437.

[19] Guido W. Imbens and Thomas Lemieux. 2008. Regression discontinuity designs: A guide to practice. *Journal of Econometrics* 142, 2 (2008), 615–635.

[20] Md Rakibul Islam and Minhaz F. Zibran. 2017. Insights into continuous integration build failures. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 467–470.

[21] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela Damian. 2014. The promises and perils of mining GitHub. In *11th Working Conference on Mining Software Repositories (MSR 2014)*, Premkumar T. Devanbu, Sung Kim, and Martin Pinzger (Eds.). ACM, Hyderabad, India, 92–101.

[22] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart M. Charters, Shirley Gibbs, and Amnart Pohthong. 2017. Robust Statistical Methods for Empirical Software Engineering. *Empirical Software Engineering* 22, 2 (2017), 579–630.

[23] Marco Manglaviti, Eduardo Coronado-Montoya, Keheliya Gallaba, and Shane McIntosh. 2017. An empirical study of the personnel overhead of continuous integration. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 471–474.

[24] Atif M. Memon, Zebao Gao, Bao N. Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. 2017. Taming Google-Scale Continuous Testing. In

39th International Conference on Software Engineering, Sebastián Uchitel, David Shepherd, and Natalia Juristo (Eds.). IEEE, Buenos Aires, Argentina, 233–242.

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

[26] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* 22, 6 (2017), 3219–3253.

[27] Klerisson V. R. Paixao, Cricia, Fernanda Madeiral Delfim, and Marcelo de Almeida Maia. 2017. On the interplay between non-functional requirements and builds on continuous integration. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 479–482.

[28] Mohammad Masudur Rahman and Chanchal K. Roy. 2017. Impact of continuous integration on code reviews. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 499–502.

[29] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 345–355.

[30] Marcel Reboucas, Renato O. Santos, Gustavo Pinto, and Fernando Castor. 2017. How does contributors' involvement influence the build status of an open-source software project?. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 475–478.

[31] Jeanine Romano, Jeffrey D. Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys. In *2016 Annual Conference of the Florida Association for Institutional Research (FAIR 2016)*, Sherri Sahs (Ed.). Florida Association for Institutional Research, Cocoa Beach,

FL, USA, 1–33.

[32] Daniel Stahl and Jan Bosch. 2014. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software* 87 (2014), 48–59.

[33] Jason Tsay, Laura Dabbish, and James D. Herbsleb. 2014. Influence of social and technical factors for evaluating contribution in GitHub. In *36th International Conference on Software Engineering (ICSE 2014)*, Pankaj Jalote, Lionel C. Briand, and André van der Hoek (Eds.). ACM, Hyderabad, India, 356–366.

[34] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2015)*, Elisabetta Di Nitto, Mark Harman, and Patrick Heymans (Eds.). ACM, Bergamo, Italy, 805–816.

[35] Frank Wilcoxon. 1945. Individual comparisons by ranking methods. *Biometrics* 1, 6 (1945), 80–83.

[36] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How open source projects use static code analysis tools in continuous integration pipelines. In *14th International Conference on Mining Software Repositories (MSR 2017)*, Jesus M. Gonzalez-Barahona, Abram Hindle, and Lin Tan (Eds.). IEEE Computer Society, Buenos Aires, Argentina, 334–344.

[37] 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 *32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, Urbana, IL, USA, 60–71.

[38] Celal Ziftci and Jim Reardon. 2017. Who Broke the Build? Automatically Identifying Changes That Induce Test Failures in Continuous Integration at Google Scale. In *39th International Conference on Software Engineering*, Sebastián Uchitel, David Shepherd, and Natalia Juristo (Eds.). IEEE, Buenos Aires, Argentina, 113–122.