The silent helper

The Silent Helper: The Impact of Continuous Integration on Code Reviews

Nathan Cassee
Eindhoven University of Technology
The Netherlands
n.w.cassee@tue.nl

Bogdan Vasilescu
Carnegie Mellon University
USA
vasilescu@cmu.edu

Alexander Serebrenik
Eindhoven University of Technology
The Netherlands
a.serebrenik@tue.nl

***Abstract*—The adoption of Continuous Integration (CI) has been shown multiple benefits for software engineering practices related to build, test and dependency management. However, the impact of CI on the social aspects of software development has been overlooked so far. Specifically, we focus on studying the impact of CI on a paradigmatic socio-technical activity within the software engineering domain, namely code reviews.**

Indeed, one might expect that the introduction of CI allows reviewers to focus on more challenging aspects of software quality that could not be assessed using CI. To assess validity of this expectation we conduct an exploratory study of code reviews in 685 GITHUB projects that have adopted Travis-CI, the most popular CI-service on GITHUB.

We observe that with the introduction of CI, pull requests are being discussed less. On average CI saves up to one review comment per pull request. This decrease in amount of discussion, however, cannot be explained by the decrease in the number of updates of the pull requests. This means that in presence of CI developers perform the same amount of work by communicating less, giving rise to the idea of CI as a silent helper.

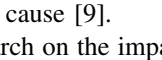
I. INTRODUCTION

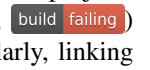
Continuous integration (CI), the software engineering practice typically implemented as the automation and frequent execution of the build and test steps with the goal of making software integration easier, has long been popular. A diversity of practitioner sources have been advocating for CI as a best practice since the early to mid 2000s, ranging from blog posts [1] and conference talks [2] to whole books [3].

More recently, researchers have began to study CI empirically, *e.g.*, to understand its associated costs and benefits [4]–[6], or assess its impact on developer productivity and software quality [7], [8] (see Section II for more discussion of related work). Often, these studies have found empirical evidence supporting many wide-held beliefs about CI, but they have also sometimes revealed differences between CI theory and practice, or revealed a more nuanced picture of the impact of CI than expected. For example, Hilton *et al.* [4] confirmed the belief that using CI correlates with higher release frequency [4]. In contrast, Vassallo *et al.* [9] found many examples of misuse of CI in practice, *e.g.*, builds staying broken for longer than usual on release branches, and failing tests being skipped in subsequent builds to avoid the build failure (the symptom) rather than fixing the cause [9].

However, while there has been prior research on the impact of CI on different practices and outcomes, the focus has typically been on technical software development tasks with high potential for automation, such as build, test, and dependency management. What is often overlooked by researchers are the social aspects of using CI as part of a complex socio-technical system with humans in the loop. An illustrative example is *code review*. Code review is one of the biggest determinants of software quality [10], [11] and a necessary gate-keeping mechanism in open-source communities, where pull request contributions often come from external project contributors [12]. Project maintainers often rely on automation tools such as CI to help with the pull request review process [7], [12], *e.g.*, to check whether the code builds, the tests pass, and the patch conforms to a linter-defined style guide. In theory, this automation should save project maintainers considerable effort and enable them to focus on other aspects of the pull request, inherently manual to assess, such as project fit, priority, adherence to non-functional requirements, and design [12]–[15]. Does that happen? *How much code review effort, if any, does CI save maintainers in practice?*

In this paper we report on an exploratory empirical study that begins to answer this question. On a large sample of GITHUB open-source projects using Travis-CI (the most popular cloud-based CI service [4]), we explore using quantitative methods how pull request review discussions changed after the introduction of CI. It is natural that process automation such as CI can help with relatively easily automatable tasks such as build and test. Here we go one step further, and quantify to what extent this automation also translates to reductions in the review burden of the project maintainers, as evidenced by the change, on average, in the number of comments posted on pull request threads.

There is high potential for effort savings. Indeed, social coding platforms like GITHUB achieve a high level of transparency by displaying a multitude of signals on individual and project pages [16], [17]. In particular, for CI-enabled pull requests on GITHUB, the CI build outcomes are displayed prominently alongside each commit as cross (✗) or check (✓) signs in the same user interface as the comments posted by the submitter and project maintainers; these signals also act as hyperlinks to the detailed build logs. Some projects also display repository badges (*e.g.*, 

, 

) indicating the CI outcome in real time and, similarly, linking to the detailed build logs. Therefore, one could expect that this

high level of transparency could reduce the need for human communication during pull request discussions. Our study investigates whether and how much this happens in practice.

Indeed, we find that on average the amount of discussion during pull request reviews, operationalized in multiple ways, decreases over time after the introduction of CI. At the same time, we find that this decrease in discussion effort cannot be explained by a decrease in expected pull request updates after review—the number of subsequent pull request commits after the pull request has been opened is, on average, unaffected by the adoption of CI. Based on our models, we estimate that on average CI saves up to one review comment per pull request, giving rise to the idea of *continuous integration as a silent helper*, where some of the tasks traditionally executed by human reviewers are now carried out by CI.

In summary, we contribute: a robust quantitative analysis of the impact of adopting Travis-CI on pull request review effort, on a large sample of GITHUB open-source projects; and a discussion of results and implications. Furthermore, the scripts and data used for the models are available online [1].

II. RELATED WORK

Continuous Integration. CI has been introduced by Fowler in a blog post in 2000, as means for the systematic integration and verification of code changes [1]. Following the introduction of CI as a practice and the onset of CI services such as Travis-CI, both closed and open-source software development teams have started to use CI, as has been found by Vasilescu *et al.* and Hilton *et al.* [4], [18]. The impact of CI on the software development process is a topic of active research. Stolberg discusses the process and results of integrating CI in an industry project [19]. Hilton finds that developers want easy to maintain, and flexible CI pipelines [20]. To better understand the motivations of practitioners using CI, Pham *et al.* interviewed open-source developers, finding several possible improvements that could be applied to improve the testing process [21]. Moreover, Rausch *et al.* analyzed the build failures of 14 open-source Java projects and found that most build failures are caused by test failures [22].

To understand how CI impacts software reviewing practices Vasilescu *et al.* found that CI also allows integrators to accept more outside contributions, without a decrease in code quality [7]. Further work by Yu *et al.* concluded that in addition to the build several technical and process variables explain review acceptance [23]. Meanwhile Zhao *et al.* used time-series analysis on a large set of open-source GITHUB projects confirming earlier results about the benefits of CI, such as a better adherence to best practices [8].

Code reviews. The notion of code inspections to improve software quality was introduced by Fagan in 1976 [24]. Fagan describes a process of line-by-line inspection of source code during team-wide meetings. While this method has proven to be effective [25], [26], it requires a significant amount of human effort, hindering its adoption [27]. As opposed to the very strict and formal method proposed by Fagan, Bacchelli and Bird described a more lightweight, asynchronous and flexible review process they call *Modern Code Review* [14]. Since then modern code review has been studied both in open-source projects and closed source-projects. In an open-source context Baysal *et al.* find that non-technical factors such as developer tenure and organization impact the outcome of code reviews [28]. Bavota and Russo have found that reviewed code is significantly less likely to contain bugs, and that code authored during a review is more readable than code authored before the review [10]. In a similar vein McIntosh *et al.* find that code review coverage shares a significant link with software quality [11]. To better understand how integrators review code Yu *et al.* investigated the latency of open-source code reviews, coming to the conclusion that many process related features impact the latency of code reviews [29]. Ebert *et al.* attempted to find expressions of confusion in code reviews, to further understand how confusion impacts code reviews [30]. Beller *et al.* investigate what types of changes are made during a code review, finding that most changes are made to improve evolvability [31]. By analyzing discussions in code reviews Tsay *et al.* found that not only do most changes relate to evolvability, evolvability is also explicitly discussed during a code review by both author and reviewer(s) [32]. Alami *et al.* [33] have interviewed 21 open source contributors to understand how code reviews are applied. They find that even though rejections are a part of code reviews, practitioners choose to use code reviews to ensure code quality is improved.

In a closed-source context Tao *et al.* interviewed Microsoft developers and found that developers tend to understand the risk of a change, and the completeness of the change [34]. Also at Microsoft Bosu *et al.* used qualitative methods to identify effective review comments and their features [35]. Furthermore, MacLeod *et al.* interviewed Microsoft developers who stated that in addition to defect finding, motivations for code reviews are knowledge sharing and improving code quality [36]. Meanwhile Sadowski *et al.* provide quantitative evidence of the modern code review process at Google, finding that most reviews tend to be lightweight and include a low number of reviewers [13]. In addition to the work by Sadowski *et al.* Kononenko *et al.* have found that practitioners consider reviews with more thorough feedback as higher quality reviews [37].

Continuous Integration and Code Reviews. Lions' share of the aforementioned studies of CI have investigated the impact of CI on technical aspects of the software development process, such as the frequency of commits or the number of bugs. Little attention has been given to the impact of CI on the social aspects of software development. Notable exceptions are recent works of Rahman and Roy [38] and and Zampetti *et al.* [39].

First, Zampetti *et al.* [39] have focused on discussions of pull requests leading to a build failure indicated by CI. The authors have created an extensive taxonomy of the topics being discussed with the test case failures and static analysis issues being responsible for most build failures. As opposed to this

[1] http://tiny.cc/6kexhz

work we consider all pull request discussions rather than those of pull requests leading to a build failure. Moreover, we study *how* the discussion takes place (*e.g.*, the number of comments) rather than *what* is being discussed (topics).

Second, Rahman and Roy [40] have observed that pull requests in projects that use CI (Travis-CI, Jenkins, and CircleCI) are more likely to be reviewed than projects that do not use CI, and that pull requests for which CI reports success, are associated with the largest amount of code reviews. The authors have compared projects using CI with those not using CI, and as such validity of their results might have been affected by project-specific differences. Indeed, larger projects or more mature projects might opt for CI and at the same time might be more successful in attracting reviews for pull requests. This is why in statistical modeling in Section IV we consider multiple control variables and explicitly take project-specific differences into account.

III. Research Questions

As discussed above, during a pull request review the communication between the author and the reviewers is very important. The comments give the author a chance to justify their proposed changes, and enable reviewers to express doubts, or to request changes, to the change set under review [41]. However, with the introduction of continuous integration the status of some aspects of the pull request under review is communicated by the CI service and the different signals associated with it on GitHub. Therefore, we expect that maintainers need to report and discuss these topics less frequently, which could indicate a reduction in the amount of communication needed during pull request reviews. A reduction in the number of comments required to finishing reviewing a pull request should help maintainers that are struggling with large numbers of changes that they have to review [12]. Therefore, we ask:

RQ$_1$: *How does the amount of communication during pull request reviews change with the introduction of CI?*

While the discussion between the pull request submitter and the project maintainers is important to clarify the submitter's intent and the quality and fit of their contribution, the review would not be complete without modifications made to the actual pull request contribution, whenever needed. When a pull request is submitted, the author signals that the proposed change is ready to be reviewed and integrated. Ideally, no changes would be needed after this point, however, it is not unusual for maintainers to request updates from the author, and for the author to implement these requested changes. For continuous integration to reduce maintainer effort, its results would need to be not just visible, but also acted upon by the pull request submitters. Therefore, we ask:

RQ$_2$: *How does the amount of updates to pull requests after they have been opened change with the introduction of CI?*

IV. Methods

The key idea behind our analysis is to collect longitudinal data for different outcome variables and treat the adoption

Figure 1: Anonymized fragment of a GitHub pull request review. Usernames and avatars have been masked in this screenshot to safeguard the privacy of the authors. Blue is the submitter, red and gray are the reviewers.

of the Travis-CI service by each project in our sample as an 'intervention'. This way, we can align all the time series of project-level outcome variables on the intervention date, and compare their evolution before and after adopting CI; this research design is known as a Regression Discontinuity Design (RDD) or an Interrupted Time Series Design. This section details the different steps involved.

A. Dataset

We assembled a dataset of code reviews (pull request discussions) from a sample of GitHub open-source projects that adopted Travis-CI as their sole CI service at some point in their history. GitHub is currently the largest platform for hosting open-source development, with over 100 million repositories at the time of writing. Travis-CI is the most popular cloud-based CI service used on GitHub.

To compose our study sample, we start from a list by Zhao *et al.* [8] of the 17,000 oldest GitHub repositories to have adopted Travis-CI at some point. We then filter this list to keep projects with a long history available for our analysis—we select repositories with at least 1,000 total pull request comments recorded over their lifetime. Finally, we use the commit status context feature of GitHub[2] to identify, and filter out, projects that also used other CI services besides Travis-CI, to remove potential confounding factors associated with such multiple-CI usage. Our final dataset contains 685 GitHub projects, collectively accounting for 1,214,826 pull requests and 5,150,112 commits.

[2]https://developer.github.com/v3/repos/statuses/

Figure 2: *Code review comments* on a pull request are attached to specific lines of code. As above blue is the author of the changes, while gray is the reviewer.

For each repository, we then collect data on all its pull requests and pull request commits, as well as two types of pull request comments: *general comments* (Figure 1), typically reflecting a higher level discussion between the author and reviewers, and *review comments* (Figure 2), which are associated with specific lines of code. Both types of comments represent code review communication between the pull request submitter and the project maintainers, and both represent types of effort that must typically be spent while evaluating pull requests. We compute all variables described below from these raw data.

B. Operationalization

Our statistical analysis (details below) uses multivariate regression modeling to test for the presence of association between the use of CI and changes in maintainers' code review effort (**RQ$_1$**) and the outcome of this effort (**RQ$_2$**). We compute multiple response variables to operationalize these concepts. We also compute several control variables to account for known co-variates.

a) Response variables: We operationalize code review effort (**RQ$_1$**) in terms of three variables that measure the volume of communication expended during pull request reviews. We operationalize the outcome of this effort (**RQ$_2$**) in terms of one variable that measures the amount of changes made to pull requests after opening, which typically happens in response to the reviewers' comments. We compute all variables at the level of individual pull requests, then later aggregate them at project level, per time window (Section IV-C). The raw data comes from the GITHUB API.

- **ReviewComments** (**RQ$_1$**): Total number of review comments per pull request; available from the raw data.
- **GeneralComments** (**RQ$_1$**): Total number of general comments per pull-request; available from the raw data.
- **EffectiveComments** (**RQ$_1$**): Total number of change-inducing review comments per pull-request. Here we follow Bosu *et al.* [35] to approximate whether review comments are effective. In their study of practitioners at Microsoft, the authors found that one of the strongest predictors of the effectiveness of a code review comment is whether the comment induces a subsequent change. In our case, we label a review comment as a change-inducing comment if the lines of code the comment is linked to were changed by a commit posted on the same pull request branch after the creation date of that review comment.
  The procedure used to identify *effective comments* is not described by Bosu *et al.* [35]. Hence, given the scraped data from GITHUB we have devised the algorithm in Figure 4 to detect *effective comments* in our dataset.
- **CommitsAfterCreate** (**RQ$_2$**): Total number of commits in the pull request that were authored after opening the pull request.

b) Control variables: We also measure known co-variates for each pull request:

- **Project name**: GITHUB slug of the project to which the pull request belongs. We use the slug to model project specific behavior as a random effect (details below).
- **Language**: Dominant programming language of the project as reported by GITHUB; similarly used as a random effect.
- **Additions**: Number of lines added in the pull request. As more lines are added we expect that more effort is required to review the modifications.
- **Deletions**: Number of lines deleted in the pull request. Similar rationale as *Additions*.
- **ChangedFiles**: Number of files modified in the pull request. As more files are modified we again expect more review activity and therefore more comments and changes during the pull-request.
- **Assignees**: Number of users assigned to the pull request as reviewers. As more people are assigned as reviewers, it is more likely that they contribute to the discussion, and therefore we expect this feature to impact the dependent variables.
- **Commits**: Total number of commits in the pull request. More commits indicate more code churn, therefore we expect more reviewing activity and in turn an effect on the dependent variables.

C. Time Series

A key decision point in our study design is how to assemble the different variables computed at the level of individual pull requests into a time series suitable for RDD modeling. The RDD analysis is typically set up such that each project in the sample contributes a fixed and constant number of observations in total, half of which are before and half after the intervention; and each observation typically corresponds to a fixed-length time window, *e.g.*, Zhao *et al.* [8] consider 12 months before and 12 months after the intervention.

Following Zhao *et al.* [8], we also aggregate individual pull request data in monthly time windows, *i.e.*, we average the values of the different variables across all pull requests of a

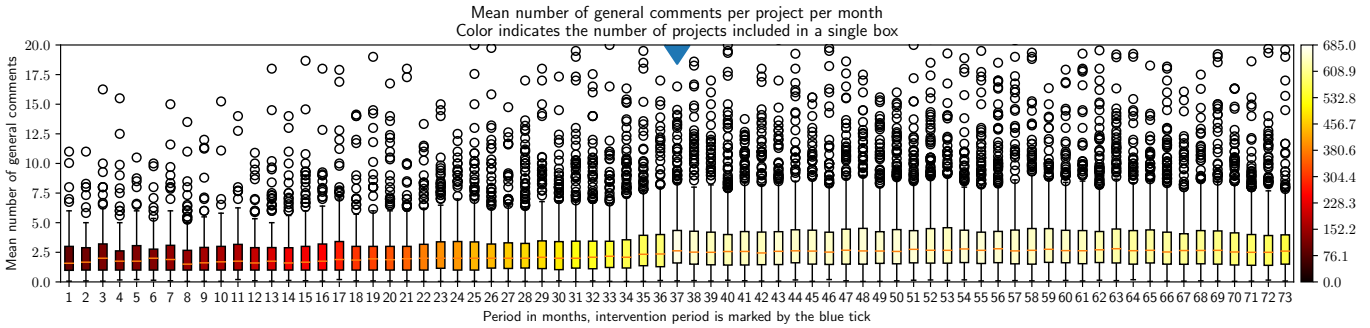


Figure 3: Mean number of pull request *general comments* per month per project around the introduction of CI (indicated by the blue tick). The color denotes how many projects had at least one pull request with comments that month.

Given a pull-request with review comments and commits, we order the comments and commits such that the oldest comment or commit is first, then we iterate over them, from oldest to newest.

- If the item is a comment:
  - Record the file and line on which the comment has been placed.
- If the item is a commit:
  - If the lines and files modified by the commit intersect with any of the recorded comments:
    - Record all of the intersecting comments as *effective comments*. Remove all of the intersecting comments, so they cannot be considered twice.
  - For every recorded comment we update its position based on the files modified by the commit.
    - If lines have been added by the commit above a recorded comment the current position of the comment is moved down by the number of added lines. This is done for all recorded comments.
    - If lines have been removed by the commits above a recorded comment the current position of the comment is moved up by the number of removed lines. This is done for all recorded comments.

Figure 4: Algorithm used to extract *effective-comments*.

given project, that were opened during that calendar month. However, we use the data to determine the number of monthly windows to consider. Indeed, not all projects are continuously active. To avoid statistical artifacts due to missing data, we typically exclude all projects that are not active for the entire length of the observation period, *e.g.*, those that do not have at least one pull request in each period. Having a longer observation period is useful, to increase the reliability of the inferred trends. However, the longer the observation period, the fewer projects can be included in the analysis, as not all projects will have been sufficiently active.

To determine an appropriate number of windows, we follow Imbens and Lemieux [42] and plot heatmaps of the number of projects that would remain part of our analysis for different lengths of the observation period, and use these to inform our decision. In the plots we include a six year time period per project, where three years (36 months) lie before the point at which CI was introduced, and three years (36 months) lie after the introduction of CI. Each three-year time period is then further divided into month-long buckets, and for each project all pull requests are distributed over the buckets based on the time the pull request was opened, *e.g.*, if a pull request for project p was opened one month after the project introduced CI, it falls in the 38^{th} bucket.

Figure 3 illustrates this process for the mean number of general pull request comments per project per period. Overall, we observe that more data is available post-CI than pre-CI. However, this does not mean that not enough data is available pre-CI, as for at least a year pre-CI there are 220 projects per bucket. This is still a large enough amount of data to fit an RDD model. Therefore, in our subsequent analysis we model a period of 12 months before and 12 months after the introduction of Travis-CI. In addition, similarly to Zhao *et al.* [8] we exclude the period ranging from 15 days before to 15 days after the adoption of Travis-CI, to increase model robustness to any instability during this transition period. Figure 5 shows a trend plot for each dependent variables, where the trend line is split around the month where Travis-CI is introduced.

D. Modeling Considerations

Regression Discontinuity Design (RDD) is a quasi-experimental technique that is well suited to model the impact of an intervention on a population, in a setting where controlled experiments are not possible [43], [44]. To accurately determine the impact of continuous integration on code reviews we want to separate the actual *intervention* from other effects. Using RDD we model our variables of interest as a function of time, and we search for the presence or absence of discontinuity around the point in time where CI was introduced. To account for different behavior observed

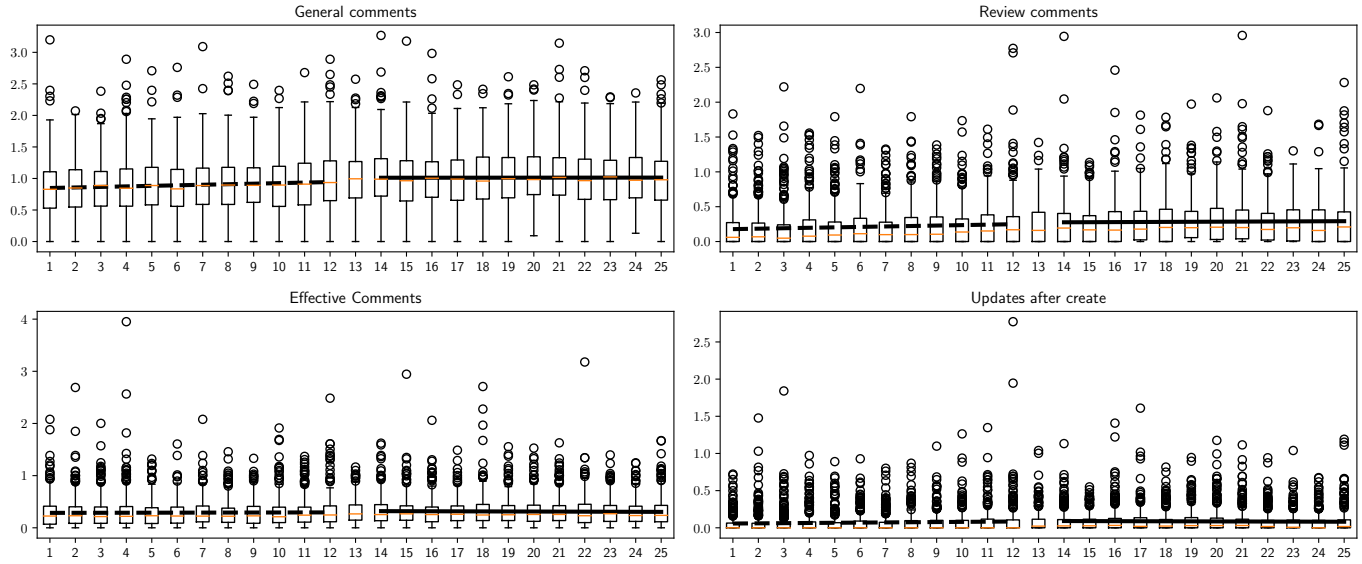


Figure 5: Plots for each of the dependent variables, showing the trend of the variable one year before and after the introduction of Travis-CI. Each datapoint represents a month of activity for a single project. The 13th month is the month in which Travis-CI was introduced. Each plot has two trend lines, the dashed one capturing the trend before the introduction of Travis-CI, and the solid one capturing the trend after the introduction of Travis-CI.

across projects or programming languages [8], we model these factors as random effects [45].

To model the adoption of Travis-CI, we model a boolean flag *intervention*, set to 0 for the pre-CI periods, and to 1 for the post-CI periods [46]. To capture trends before, and changes in trend after the adoption of Travis-CI, we include two additional independent variables in the regression model [46]: *time* and *time_after_intervention*. The variable *time* models the pre-CI trend as a discrete numerical variable. *time* captures the period index, ranging from 1 to 12 pre-CI, and having a value of 0 post-CI. The variable *time_after_intervention* is used to model the trend post-CI in a similar fashion as *time*; the value of the variable is 0 for all periods before the adoption of CI, and ranges from 1 to 12 after the intervention.

We then estimate the model coefficients using the `lmerTest` R package [47]. The model coefficients for the different variables and the statistical significance of these coefficients are used to determine whether the introduction of CI had an immediate impact on the dependent variables (immediate increase or drop, if the coefficient for *intervention* is statistically significant and positive or negative, respectively), and how the one-year trends in these variables changed after the introduction of CI compared to before. For example, a statistically significant and positive coefficient for *time* would indicate an increasing trend in the dependent variable over the 12 months prior to adopting CI. Similarly, a statistically significant and positive coefficient for *time_after_intervention* would indicate an increasing trend in the dependent variable over the 12 months after adopting CI. The difference in magnitude between the coefficients for *time* and *time_after_intervention* indicates how the pre-CI trend, if any, changed after adopting CI.

As standard [48], we evaluate model fit using the marginal R^2_m (only fixed effects) and conditional R^2_c scores (combined fixed and random effects, analogous to the traditional R^2 value in linear models). Furthermore, we use ANOVA to estimate the effect size of each variable, *i.e.*, the fraction of the total variance explained by the model that can be attributed to each individual variable [49]. To prevent collinearity from affecting our coefficient estimates, we exclude variables for which the VIF score is higher than 5, as recommended by Sheater [50].

V. Results

Recall, we model the change in response variables capturing review effort (**RQ$_1$**) and review outcome (**RQ$_2$**) after the adoption of Travis-CI. In this section we present the results of the statistical models fit.

A. RQ$_1$: Code Review Communication

a) General comments: We start by analyzing the trend in *general* pull request comments. For this model, the dependent variable is the mean number of *general* pull request comments per repository per month, regressed on the other variables as independent variables. These include the mean lines of code added, the users assigned and the number of commits.

Table I shows the statistical significance and estimated coefficients for each of the variables. In addition to that, it also shows the sum of squares, or the amount of variance explained by each variable. The marginal R^2_m goodness of fit score (fixed effects only) is 0.1168, while the conditional R^2_c goodness of fit score (fixed and random effects) is 0.5922, therefore, when taking into account per repository and programming

Table I: Estimated coefficients and statistical significance levels for the RDD model of *general* pull request comments.

Feature	Coeffs.	Sum sq.
Intercept***	0.7698	
Additions***	0.0464	2.272
Deletions***	-0.0330	1.280
Commits***	-0.0726	1.225
Assignees***	0.3057	2.705
ChangedFiles*	-0.0419	0.390
CommitsAfterCreate***	0.4922	47.699
TotalPrs***	0.0006	1.155
time***	0.0077	2.600
intervention**	0.0588	0.756
time_after_intervention***	-0.0073	1.071

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

language variability (the random effects) the model fits the general comments data well.

From Table I we observe that all three RDD variables, used to model trends over time and the intervention itself, have statistically significant coefficients, indicating that the adoption of Travis-CI is associated with changes in the number of general pull request comments. This effects manifests itself as a slight increase over time in the number of general comments leading to the adoption of CI (the positive coefficient for the *time* variable) and a level increase directly after the adoption (the positive coefficient for the *intervention* variable), after which the number of general comments decreases slightly over time (the negative coefficient for the *time_after_intervention* variable).

We also observe that all independent variables have statistically significant coefficients in the model. The *CommitsAfterCreate* variable is especially notable, as it explains the largest amount of variability in the model, indicating that development activity during the lifetime of the pull request is strongly associated with the amount of communication during the pull request review. This is expected (hence the inclusion of *CommitsAfterCreate* as a control variable), since updates to a pull request after creation, through subsequent commits on that respective branch, would likely generate additional discussion.

Overall, the model suggests that on average and while controling for other variables such as pull request updates after creation, there is less general discussion of pull requests over time after adopting CI.

b) Review comments: Another operationalization of code review communication effort is the amount of *review* comments in pull requests, *i.e.*, those comments attached to specific lines of code in the pull request commits. Here we model the mean number of review comments per month as the dependent variable, while the other features are modeled as independent variables. As above, project name and programming language are modeled as random effects, to account for potential per project and per language variability.

Table II presents the results of the fitted RDD model for review comments. The marginal R^2_m score of the model is 0.148, while the conditional R^2_c score is 0.593, again indicating good model fit. Analyzing the results in the table we observe similar associations as in the case of the general pull request comments model above. Specifically, there is an increasing trend in the number of review comments before the adoption of CI, which gets approximately reversed after CI is adopted (comparable magnitude of positive and negative estimated coefficients, before and after the intervention, respectively).

Table II: Estimated coefficients and statistical significance levels for the RDD model of line-level *review* comments.

Feature	Coeffs.	Sum sq.
Intercept*	0.0421	
Additions***	0.0488	3.056
Deletions	-0.0050	0.030
Commits***	-0.0440	0.460
Assignees	0.0300	0.027
ChangedFiles***	-0.0424	0.407
CommitsAfterCreate***	0.3672	29.973
TotalPrs**	0.0003	0.302
time***	0.0057	1.437
intervention	0.0133	0.045
time_after_intervention**	-0.0040	0.350

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

Other variables behave similarly as above. For example, we note the strong effect of the *CommitsAfterCreate* variable. This is expected as a likely trigger for modifications during the lifetime of a pull request is the presence of review comments by project maintainers, requesting changes from the author.

Overall, our conclusion is similar:

A previously increasing trend in the number of line-level review comments made during pull request reviews is, on average, reversed after CI adoption.

c) Effective comments: We also modeled the change in the number of effective comments after adopting Travis-CI, *i.e.*, those review comments that lead to a direct change in the code under review [35], expressed as subsequent commits on the same pull request branch. We fit an RDD model as described above, having as dependent variable the mean number of effective comments per repository per month, and with similar independent and control variables as above. As before, the project name and the dominant programming language have been modeled as random effects.

Table III shows the slope estimates, the statistical significance levels, and the sum of squares for the fitted RDD model. R^2_m is 0.107, while R^2_c is 0.425, indicating that the model fits the data well.

The significant control variables in Table III are the features which capture the activity seen within a single month. As expected this indicates that when more files or more lines of code are modified, and a pull request becomes more complex, that more effective comments are being posted, indicating the pull requests require more review.

Table III: Estimated coefficients and statistical significance levels for the RDD model of *effective* comments.

Feature	Coeffs.	Sum sq.
Intercept***	-0.1312	
Additions***	0.0273	0.986
Deletions***	-0.0153	0.287
Commits***	0.1073	4.324
Assignees	0.0024	<0.001
ChangedFiles***	-0.0267	0.167
GeneralComments***	0.0581	2.380
TotalPrs	< 0.0001	0.021
time***	0.0024	0.260
intervention	0.0009	<0.001
time_after_intervention***	-0.0030	0.196

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

Table IV: Estimated coefficients and statistical significance levels for the RDD model of pull request commits.

Feature	Coeffs.	Sum sq.
Intercept***	-0.4357	
Additions***	0.0294	1.166
Deletions	0.0060	0.045
Commits***	0.5237	105.467
Assignees**	0.0830	0.273
ChangedFiles***	-0.0749	1.348
TotalPrs	<0.0001	0.033
GeneralComments***	0.1505	16.903
time*	0.0019	0.155
intervention	0.0028	0.002
time_after_intervention	-0.0017	0.067

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$

Two of the three coefficients used to model the impact of CI are also statistically significant. No immediate discontinuity is observed after projects adopt CI. However, a positive trend in the number of effective comments before the adoption of Travis-CI is reversed: there is a steady decrease in the number of effective comments after the adoption of CI.

A previously increasing trend in the number of change-inducing review comments is, on average, reversed after CI adoption.

B. RQ_2: Code Changes During Review

The results presented in the previous subsection paint a consistent picture: the amount of discussion during pull request reviews goes down over time after adopting CI. But could that be explained by pull requests becoming less complex over time, and thus requiring fewer changes?

The number of changes (commits) made during a pull request can be considered as the net effect of the pull request review. Typically, by opening the pull request the author signals that they are done with the work. Any changes made during the review are either at the request of reviewers, or might be triggered by continuous integration when a build fails. Therefore, we model whether CI impacts the number of commits made during a pull request review. As before, we fit an RDD model with general activity metrics as controls and random effects for project name and dominant programming language. The results of the fitted model are shown in Table IV. The marginal goodness of fit score R^2_m has a value of 0.547, while the conditional goodness of fit score R^2_c has a value of 0.658, indicating that the model fits well.

From Table IV we observe that the introduction of Travis-CI is not statistically associated with the number of added commits during a pull request review. No immediate discontinuity is observed for repositories on the introduction of CI. Notably, unlike the previous models of discussion comments discussed above, there is no decreasing trend after the adoption of CI. We conclude that:

On average, the outcome of the pull request review process, measured in terms of subsequent code changes made during a pull request review, does not change after adopting CI.

VI. Discussion

By modeling both the pre and post CI adoption code review history of a large sample of open-source repositories we find that the communication between reviewers and author starts to gradually decrease after the adoption of Travis-CI (**RQ_1**). In a way, the reviewers hand-off a set of their tasks to the automated checks performed by CI. The output of these checks is visible through user interface signals on GitHub (*e.g.*, ✗ or ✓, build failing or build passing) and accessible through the CI service, instead of a comment manually posted by a reviewer.

In different models related to the code review communication (**RQ_1**) we see that $R^2_m \simeq 0.1$, while R^2_c ranges between 0.4 and 0.6. This means that project-to-project and language-to-language differences, modeled as random effects, are responsible for most of the explained variability.

Using the estimated coefficients of the models we can compute the net communication effort savings effect of using CI. The regression discontinuity design framework allows us to assume that the pre-intervention trend would have continued unchanged after the adoption of continuous integration, if CI had no effect. In addition, the multiple regression modeling framework allows us to assume that the other independent variables remain constant. Therefore, we construct two surrogate data points, where one data point is constructed such that it simulates the progression of reviewing practices before CI adoption, and the second data point simulates the impact of the adoption of CI on the reviewing practices. We then use the previously fitted models to predict the mean number of general and review comments for those two data points. The delta in number of comments between these two data points measures, thus, the impact of CI.

To this end, to construct these two surrogate data points we take the median value for the numerical variables. Furthermore

we assign both data points to the same random programming language and project, to ensure that the already fitted models can account for the random effects. For the non-CI data-point we set *time* to 24, *intervention* to 0 and *time_after_intervention* to 0, such that we simulate a year of continued code reviews with no CI. Conversely, the second data point has *time* set to 0, *intervention* set to 1, and *time_after_intervention* set to 12, simulating the progression of a year of code review activity post-CI. Applying this technique for general comments we find that a year post-CI the median pull request review has approximately .85 general comments less, and approximately .23 line-level review comments less. That is, **on average CI saves up to one review comment per pull request**. To provide some context for these savings, the average pull-request for the set of projects that was analyzed has a mean number of 3 general comments, and 1.5 review comments. Therefore, the reduction in the number of comments is on average quite substantial for the projects in the dataset.

This apparent hand-off gives rises to the idea of **continuous integration as a silent helper**, where some of the tasks traditionally executed by human reviewers are now carried out by CI, leading to less discussions in code reviews. Furthermore, from earlier work we know that projects can process more outside contributions after the adoption of CI without any change in code quality [7]. This would indicate that code quality is safeguarded while adopters of CI can review and process more pull requests. The fact that we find that less communication is required for maintainers to process pull requests could be the reason why more outside contributions can be processed. Furthermore, the fact that less time is spent reviewing can further help maintainers prioritize pull requests, and further improve open-source projects [12].

Secondly, we observe that the number of change-inducing requests by maintainers also starts to decrease post CI (**RQ$_1$**), while the number of changes made to the pull request code during the review process does not decrease (**RQ$_2$**). This difference in effects observed further supports the idea of CI as a silent helper; less comments by reviewers are required to maintain the same level of quality as continuous integration supports maintainers by automating certain checks. Hence, maintainers can save themselves effort by adopting CI. The work that they then save because continuous integration performs some of the tasks traditionally executed by manual reviewers can be invested in other activities.

Moreover, Ram *et al.* state that conformance of the code under review to the project style guides improves reviewability of the source-code [51]. Linters, and other static code analysis tools, can be integrated in the automated checks performed by the CI service. A manual inspection of some of the CI configurations shows that there are indeed projects which use static analysis tools. Furthermore, Zampetti *et al.* [52] found that static code analysis tools are used as part of the CI process for several Java projects. Conformance of the source code to the project style guide can be enforced using static analysis tools, and this could further help shift effort from manual review efforts to automated checks.

VII. Threats to validity

We note several threats to the internal, external, and construct validity of our study [53].

A. Internal Validity

For this study we applied quantitative methods. Using this approach we found that over time fewer comments are posted after the introduction of continuous integration. Our results could be strengthened with qualitative methods, *e.g.*, interviews, to determine how reviewers and pull request authors experience the role and benefits of CI. However, even though we used only quantitative methods, our statistical analysis follows state-of-the-art methods (RDD) and modeling best practices, as advocated by Wieringa [54]. Furthermore, we measure the conditional model fit to verify that the fitted models match the data well and can indeed be used to test the impact of CI on the dependent variables.

To ensure that there are no confounding factors several controls that might influence the independent variables have been added. However, in addition to the already identified dependent variables there might also be other factors that influence code reviewing practices. These factors could include an increase in testing, or the adoption of TDD, which might also influence code reviewing practices. Moreover, we exclude projects that adopt other CI services, however, it has been found that Travis-CI is less suited for for instance Windows development [6].

Moreover the operationalization chosen to measure whether a review comment is effective, *i.e.*, change inducing, is dependent on how well GitHub preserves code review history. Our algorithm to determine whether a review comment is change inducing recreates a timeline of events that occur during a pull request review such that the review comments that triggered a commit can be found. However, during a pull request review practitioners can directly use `git` to rewrite the public history, for instance by force pushing or squashing commits. If this occurs during the lifetime of a pull request on GitHub, and GitHub does not preserve the original commits, then the algorithm may not accurately find effective comments. Therefore, for projects which regularly apply public history rewriting during code reviews the number of change-inducing comments reported might be lower than the actual number of change-inducing comments.

B. External Validity

Code review data has been collected from a single platform, GitHub. However, GitHub is not the only platform with publicly accessible code review data. For example, other platforms such as `BitBucket`, `GitLab`, and on-premise `Gerrit` installations all have code review data available. Secondly the project selection and sampling strategy serves to ensure only active, collaborative, open-source software projects are analyzed.

Both these choices may impact the generalizability of this study. For instance, if another platform integrates CI differently from GitHub, other effects might be observed.

Secondly, the criteria used to filter out inactive, or non-collaborative projects may introduce bias; Work by Munaiah *et al.* [55] has shown that the recall of such filtering techniques can be 30% or lower. Therefore, there could be a class of smaller yet active and collaborative open-source software projects which we are not represented in our study. These projects may use a different continuous integration configuration, or they may use a more informal code review process, potentially experiencing the benefits of CI observed in this study differently. Therefore, both the platform selection and project filtering might impact the external validity of this study.

However, our study is representative for many projects hosted on GITHUB; moreover, platforms such as `BitBucket` and `GitLab` offer similar features related to code reviews and CI, hence we expect also similar code review processes on these platforms. Finally, the filtering criteria ensure that the projects we have analyzed are representative of larger, active and collaborative projects. While our findings might not generalize to all software projects that adopt CI, they may well generalize to the larger and more active open-source projects.

C. Construct Validity

The variables used in the RDD models are proxies used to model the interactions that occur during a pull request review. However, such variables as the number of review and general comments do not necessarily capture all communication, and therefore these proxies might not completely capture the full effort of reviewers. For instance, we do not take into account the length of a comment, nor do we test whether developers communicate over other channels such as instant messaging apps.

Secondly, we also use the number of code changes during a review as a proxy for reviewing effort. However, we do not take into account the actual effort that went into authoring the change, as this information is not available on GITHUB. Moreover, the number of additions, deletions, and changed files as reported by GITHUB also include changes that do not affect the source code. This could include whitespace changes or modifications to non source-code files.

The notion of effective comments has been adapted from the work by Bosu *et al.* [35]. However, to the best of our knowledge no other work has attempted to adapt the notion of effective comments to GITHUB repositories. Due to several differences between the code review tool at Microsoft and the pull requests of GITHUB it was not possible to maintain the definition proposed by Bosu *et al.* exactly; instead, we have opted to simplify their definition. This simplification might introduce both false positives and false

VIII. CONCLUSION

In this paper we present an exploratory empirical study investigating the effects of Continuous Integration on open-source code reviews. Literature has described both the modern code review process as it has been adopted by open-source software projects, and the impacts of adopting continuous integration on development practices. To the best of our knowledge the intersection of these topic has not yet been analyzed. We believe that because of the social aspect of code reviews the impact of continuous integration on code reviews is particularly interesting. To understand this impact we have analyzed a large sample of open-source software projects hosted on GITHUB.

We formalized two research questions focusing on whether communication during a code review is affected by continuous integration, and whether the number of changes made during a code review is affected by Continuous Integration. By modeling code review data around the introduction of Continuous Integration we find that the number of comments per code review decreases after to the adoption of continuous integration, while the number of changes made during a code review remains constant. Using the fitted models we determine that on average a code review a year after the introduction of continuous integration has roughly one less general comment (**RQ$_1$**). Meanwhile, we find no such effect for changes made during a code review (**RQ$_2$**).

Our findings are important in determining the benefit of Continuous Integration for practitioners. It is known that reviewers can be overwhelmed by the number of changes that require a review, and that this overload leads to external contributions being ignored [12], [56]. Therefore, for projects that struggle with the workload imposed by having to review external contributions adopting continuous integration can be a valuable time-saver.

The role fulfilled by continuous integration for open-source projects has been reported on before [4], [7], [8], [38]. Furthermore, given the signals associated with continuous integration, and the fact that continuous integration on average saves a general comment per code review, we posit that continuous integration fulfills the role of a silent helper. Through the signals posted by Travis-CI it is present for all code reviews, and more importantly continuous integration helps reviewers save a measurable and significant amount of effort. However, continuous integration itself does not become a part of the discussion, as Travis-CI does not post any comments to code reviews.

Acknowledgements. Vasilescu gratefully acknowledges support from the National Science Foundation (award 1717415).

REFERENCES

[1] M. Fowler, "Continuous integration (original version)," https://martinfowler.com/articles/originalContinuousIntegration.html, 2000.

[2] R. O. Rogers, "Scaling continuous integration," in *International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, 2004, pp. 68–76.

[3] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.

[4] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs, and benefits of continuous integration in open-source projects," in *International Conference on Automated Software Engineering (ASE)*. ACM, 2016, pp. 426–437.

[5] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 197–207.

[6] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "A conceptual replication of continuous integration pain points in the context of Travis CI," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, pp. 647–658.

[7] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in github," in *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 805–816.

[8] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu, "The impact of continuous integration on other software development practices: A large-scale empirical study," in *International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 60–71.

[9] C. Vassallo, S. Proksch, H. C. Gall, and M. Di Penta, "Automated reporting of anti-patterns and decay in continuous integration," in *International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 105–115.

[10] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 81–90.

[11] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, pp. 2146–2189, 2016.

[12] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *International Conference on Software Engineering (ICSE)*. IEEE, 2015, pp. 358–368.

[13] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ACM, 2018, pp. 181–190.

[14] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 712–721.

[15] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "Confusion in code reviews: Reasons, impacts, and coping strategies," in *26th IEEE International Conference on Software Analysis, Evolution and Reengineering*, X. Wang, D. Lo, and E. Shihab, Eds. IEEE, 2019, pp. 49–60.

[16] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: transparency and collaboration in an open software repository," in *ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW)*. ACM, 2012, pp. 1277–1286.

[17] A. Trockman, S. Zhou, C. Kästner, and B. Vasilescu, "Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem," in *International Conference on Software Engineering (ICSE)*. ACM, 2018, pp. 511–522.

[18] B. Vasilescu, S. van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. J. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from github," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 401–405.

[19] S. Stolberg, "Enabling agile testing through continuous integration," in *2009 Agile Conference*, 2009, pp. 369–374.

[20] M. Hilton, N. Nelson, T. Tunnell, D. Marinov, and D. Dig, "Trade-offs in continuous integration: Assurance, security, and flexibility," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 197–207.

[21] R. Pham, L. Singer, O. Liskin, F. F. Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 112–121.

[22] T. Rausch, W. Hummer, P. Leitner, and S. Schulte, "An empirical analysis of build failures in the continuous integration workflows of java-based open-source software," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 345–355.

[23] Y. Yu, G. Yin, T. Wang, C. Yang, and H. Wang, "Determinants of pull-based development in the context of continuous integration," *Science China Information Sciences*, vol. 59, p. 080104, 2016.

[24] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, pp. 182–211, 1976.

[25] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections: an effective verification process," *IEEE Software*, vol. 6, 1989.

[26] M. E. Fagan, *Advances in Software Inspections*. Springer Berlin Heidelberg, 1986, pp. 335–360.

[27] P. M. Johnson, "Reengineering inspection," *Commun. ACM*, vol. 41, pp. 49–52, 1998.

[28] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "Investigating technical and non-technical factors influencing modern code review," *Empirical Software Engineering*, vol. 21, pp. 932–959, 2016.

[29] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on github," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 367–371.

[30] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "Confusion detection in code reviews," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 549–553.

[31] M. Beller, A. Bacchelli, A. Zaidman, and E. Juergens, "Modern code reviews in open-source projects: which problems do they fix?" in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, p. 202211.

[32] J. Tsay, L. Dabbish, and J. Herbsleb, "Let's talk about it: Evaluating contributions through discussion in GitHub," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2014, pp. 144–154.

[33] A. Alami, M. L. Cohn, and A. Wasowski, "Why does code review work for open source software communities?" in *International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1073–1083.

[34] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim, "How do software engineers understand code changes?: An exploratory study in industry," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, pp. 51:1–51:11.

[35] A. Bosu, M. Greiler, and C. Bird, "Characteristics of useful code reviews: An empirical study at microsoft," in *Proceedings of the 12th Working Conference on Mining Software Repositories*. IEEE Press, 2015, pp. 146–156.

[36] L. MacLeod, M. Greiler, M. Storey, C. Bird, and J. Czerwonka, "Code reviewing in the trenches: Challenges and best practices," *IEEE Software*, vol. 35, pp. 34–42, 2018.

[37] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 1028–1038.

[38] M. M. Rahman and C. K. Roy, "Impact of continuous integration on code reviews," in *International Conference on Mining Software Repositories*, 2017, pp. 499–502.

[39] F. Zampetti, G. Bavota, G. Canfora, and M. Di Penta, "A study on the interplay between pull request review and continuous integration builds," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 38–48.

[40] A. Rahman, A. Agrawal, R. Krishna, and A. Sobran, "Characterizing the Influence of Continuous Integration: Empirical Results from 250+ Open Source and Proprietary Projects," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*. ACM, 2018, pp. 8–14.

[41] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "Communicative intention in code review questions," in *2018 IEEE International Conference on Software Maintenance and Evolution*, 2018, pp. 519–523.

[42] "Regression discontinuity designs: A guide to practice," *Journal of Econometrics*, vol. 142, pp. 615 – 635, 2008.

[43] D. L. Thistlethwaite and D. T. Campbell, "Regression-discontinuity analysis: An alternative to the ex post facto experiment." *Journal of Educational Psychology*, vol. 51, pp. 309–317, 1960.

[44] T. Cook and D. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin, 1979.

[45] A. Galecki and T. Burzykowski, *Linear Mixed-Effects Models Using R: A Step-by-Step Approach*. Springer Publishing Company, Incorporated, 2013.

[46] A. K. Wagner, S. B. Soumerai, F. Zhang, and D. Ross-Degnan, "Segmented regression analysis of interrupted time series studies in medication use research," *Journal of Clinical Pharmacy and Therapeutics*, vol. 27, pp. 299–309, 2002.

[47] A. Kuznetsova, P. B. Brockhoff, and R. H. B. Christensen, "lmerTest package: Tests in linear mixed effects models," *Journal of Statistical Software*, vol. 82, no. 13, pp. 1–26, 2017.

[48] S. Nakagawa and H. Schielzeth, "A general and simple method for obtaining r2 from generalized linear mixed-effects models," *Methods in Ecology and Evolution*, vol. 4, pp. 133–142, 2013.

[49] J. Kaufmann and A. Schering, *Analysis of Variance ANOVA*. American Cancer Society, 2014.

[50] S. Sheather, *A Modern Approach to Regression with R*, ser. Springer Texts in Statistics. Springer New York, 2009. [Online]. Available: https://books.google.nl/books?id=zS3Jiyxqr98C

[51] A. Ram, A. A. Sawant, M. Castelluccio, and A. Bacchelli, "What makes a code change easier to review: An empirical investigation on code change reviewability," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 201–212.

[52] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 334–344.

[53] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: A roadmap," in *Proceedings of the Conference on The Future of Software Engineering*. ACM, 2000, pp. 345–355.

[54] R. J. Wieringa, *Abductive Inference Design*, 2014, pp. 177–199.

[55] N. Munaiah, S. Kroh, C. Cabrey, and M. Nagappan, "Curating github for engineered software projects," *Empirical Software Engineering*, vol. 22, pp. 3219–3253, 2017.

[56] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *2011 33rd International Conference on Software Engineering*, 2011, pp. 541–550.