The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study

Yangyang Zhao* — Nanjing University, China — njuzhyy@gmail.com
Alexander Serebrenik — Eindhoven U of Technology, The Netherlands — a.serebrenik@tue.nl
Yuming Zhou — Nanjing University, China — zhouyuming@nju.edu.cn
Vladimir Filkov — UC Davis, USA — filkov@cs.ucdavis.edu
Bogdan Vasilescu* — Carnegie Mellon University, USA — vasilescu@cmu.edu

Abstract—**Continuous Integration (CI) has become a disruptive innovation in software development: with proper tool support and adoption, positive effects have been demonstrated for pull request throughput and scaling up of project sizes. As any other innovation, adopting CI implies adapting existing practices in order to take full advantage of its potential, and "best practices" to that end have been proposed. Here we study the adaptation and evolution of code writing and submission, issue and pull request closing, and testing practices as TRAVIS CI is adopted by hundreds of established projects on GITHUB. To help essentialize the quantitative results, we also survey a sample of GITHUB developers about their experiences with adopting TRAVIS CI. Our findings suggest a more nuanced picture of how GITHUB teams are adapting to, and benefiting from, continuous integration technology than suggested by prior work.**

I. INTRODUCTION

The DEVOPS movement, made popular in recent years, is a paradigm shift [1]–[4]. It aims to get changes into production as quickly as possible, without compromising software quality. While no standard definitions exist (the term is often overloaded), here we refer to DEVOPS as a culture that emphasizes *automation of the processes of building, testing, and deploying software*. In practice, DEVOPS is supported by a multitude of tools for configuration management, cloud-based continuous integration, and automated deployment, which enjoy widespread open-source [5] and industrial adoption [6], [7].

In this study we focus on Continuous Integration (CI), the key enabler of DEVOPS. CI is a well known concept in Extreme Programming, promulgated in Martin Fowler's 2000 blog post [8]. As a practice, CI is seeing broad adoption with the increasing popularity of the GITHUB pull-based development model [9] and the plethora of open-source, GITHUB-compatible, cloud-based CI tools, such as TRAVIS CI, CLOUDBEES, and CIRCLECI. In a decentralized, social coding context such as GITHUB, CI is particularly relevant. By automatically building and testing a project's code base, in isolation, with each incoming code change (*i.e.*, push commit, pull request), CI has the potential to: (i) speed up development (code change throughput) [5], [10], [11]; (ii) help maintain code quality [12], [13]. Clearly, CI promises to be a disruptive technology in distributed software development.

For it to be effective, CI must allow for a seamless back and forth between development, testing (*e.g.*, unit, integration, code review), and deployment. However, the road to efficiency is riddled with choices and trade-offs. For example, working in large increments may lead to more meaningful change sets, but it may also complicate synchronization between team members and, if necessary, reverting changes. Conversely, more frequent changes facilitate merging, but they also require more computing infrastructure for CI, since by default the code is built and all tests are executed with *every* change. Moreover, while CI runs on smaller, more frequent changes would provide earlier feedback on potential problems, they may also lead to process "noise", where developers start to ignore the CI build status due to information overload, irrespective of whether the build is clean or broken [14].

Several CI "best practices" have been proposed, *e.g.*, by Fowler in his influential blog post [8], such as *Everyone Commits To the Mainline Every Day*, *Fix Broken Builds Immediately*, and *Keep the Build Fast*. However, despite the large scale adoption of CI, we know relatively little about the state of the practice in using this technology and whether developers are aligning their practices with Fowler's proposed "best practices". Such knowledge can help developers to optimize their practices, project maintainers to make informed decisions about adopting CI, and researchers and tool builders to identify areas in need of attention.

In this paper we report on a study of a large sample of GITHUB open-source projects that adopted TRAVIS CI, by far the most popular CI infrastructure used on GITHUB [5]. In particular, we focus on the *transition to using TRAVIS CI*, and investigate how development practices changed following this shift. To this end, we introduce *regression discontinuity design analyses* to quantitatively evaluate the effect of an intervention, in our case adoption of TRAVIS CI, on the transition toward expected behaviors in the above three practices (measured from trace data in a large sample of GITHUB projects, appropriately selected). Qualitatively, to help essentialize the quantitative results, we survey a sample of GITHUB developers

*These authors contributed equally to this work about their experiences with adopting TRAVIS CI. Applying this mixed methodology, we find that:

- the increasing number of merge commits aligns with the "commit often" guideline of Fowler, but is likely to be further encouraged by the shift to a more distributed workflow with multiple branches and pull requests;
- the "commit small" guideline, however, is followed only to some extent, with large differences between projects in terms of adherence to this guideline;
- the expected increasing trend in the number of closed pull requests manifests itself after the introduction of TRAVIS CI, and even then only after the initial plateau period;
- the pull request latency increases despite the code changes becoming smaller;
- while the number of issues closed increases, this trend is unexpectedly slowed down by TRAVIS CI;
- after initial adjustments when adopting TRAVIS CI, test suite sizes seem to increase.

II. DEVELOPMENT OF RESEARCH QUESTIONS

Transitioning to an integrated CI platform, like TRAVIS, involves adaptation of established processes to the new environment. During this transition, some developers will experience a more streamlined process evolution trajectory than others. Studying those trajectories can provide lessons learned.

Continuous integration encourages developers to "break down their work into small chunks of a few hours each", as smaller and more frequent commits helps them keep track of their progress and reduces the debugging effort [15], [16]. Miller has observed that, on average, Microsoft developers committed once a day, while off-shore developers committed less frequently due to network latencies [17]; Stolberg expects everyone to commit every day [10] and Yüksel reports 33 commits per day after introduction of CI [18]. On a related note, the interviewees in the study of Leppanen *et al.* [19] saw higher release frequency as an advantage and reported that the CI "radically decreased time-to-market". Hence, we ask in $\mathbf{RQ}_1$: *Are developers committing code more frequently?*

Frequent commits can be expected to be smaller in size: indeed, the quote from Fowler's blog post refers to "small chunks" [15], [16]. Hence, we formulate $\mathbf{RQ}_2$: *Are developers reducing the size of code changes in each commit post CI adoption? Do they continue to do so over time?*

In the GITHUB-common pull-based development model, wherein project outsiders (and sometimes insiders too) submit changes in the form of pull requests (PRs), evaluation throughput is key. Indeed, popular GITHUB projects receive increasingly many PRs, all of which need to be tested and reviewed pre-merge. One of the most commonly advocated benefits of TRAVIS CI is the ability to "scale up" development, by automating PR testing and delegating the workload to cloud-based infrastructure; in this sense, Vasilescu *et al.* [12] report that GITHUB teams are able to process (*i.e.*, close) more PRs per unit time after adopting TRAVIS CI. This high level of automation should also impact the speed, not just the volume, of PR evaluations. Hence, we ask in $\mathbf{RQ}_3$: *Are pull requests closed more quickly post CI adoption?*

For CI, and DEVOPS in general, to have the stated benefits, effective coordination between all project contributors is important; a common mechanism for this are issue reports. Similarly to the pull requests, the number of closed issues per time period may change after CI adoption, in response to increased project activity or efficiency. Hence, we ask in $\mathbf{RQ}_4$: *Are more issues being closed after adopting CI?*

Finally, CI is closely related to the presence of automated tests [15]. Duvall even claims that CI without such tests should not be considered CI at all [16], while Cannizzo *et al.* deem an extensive suite of unit and acceptance tests to be an essential first step [20]. Moreover, CI is frequently introduced together with test automation [18]. However, developing tests suited for automation requires a change in developers' mindset, and presence of a comprehensive set of tests incurs additional maintenance costs [21]. Naturally, more automated testing will expose more errors, and CI makes it possible to track and react to different error types differently [22]. Therefore, we formulate $\mathbf{RQ}_5$: *How does the usage of automated testing change over time after CI adoption?*

III. METHODS

We collected and statistically analyzed data from a large sample of open-source projects on GITHUB, that adopted TRAVIS CI at some point during their history. We further surveyed a sample of those projects' TRAVIS CI adopters.

A. Data Gathering

Data collection involved mining multiple sources: GHTORRENT [23], the GITHUB API, project version control (git) logs, and the TRAVIS CI API. The goal was to select nontrivial projects that adopted TRAVIS CI, and had sufficient activity both before and after adoption, in order to observe potential transition effects. Note that for the purpose of this study we don't distinguish between "project" and "repository"; the term "project" has also been used to refer to a collection of interrelated repositories in the literature [24].

Candidate Projects: We started by identifying GITHUB projects that use TRAVIS. To our knowledge, no such list exists (TRAVISTORRENT [25] is restricted to Java and Ruby projects), so we wrote a script to iterate over non-fork GHTORRENT projects (oldest to newest) and poke the TRAVIS CI API (cf. [26]) to determine if the project used TRAVIS; we ran and stopped the script after identifying approximately half (165,549) of all GITHUB projects that ever used TRAVIS CI.[1]

Next, we cloned the GITHUB repositories of all these projects locally, and extracted their main-branch commit histories using PERCEVAL, an open-source repository mining tool part of the GRIMOIRELAB tool suite. Then, we traversed each project's commit history to determine when maintainers introduced TRAVIS CI, by identifying the earliest commit

[1]At the time of writing, TRAVIS self reports being used in over 318,000 open source GITHUB projects, see https://travis-ci.org.

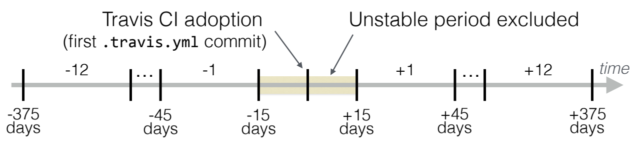

Fig. 1. Overview of our time series assembly method.

to the `.travis.yml` configuration file, and recorded its authored timestamp as the timestamp of the TRAVIS adoption.

Time Series: We proceeded to aggregate data about the different practices considered in 30-day windows, 12 on each side around the TRAVIS CI adoption event (Figure 1).

During initial exploration, we saw several occasions of TRAVIS being adopted as part of a larger restructuring effort, which lasted several days. Examples include changing the build system from Ant to Gradle and, consequently, reorganizing project folders; updated library dependencies; and restructured tests. Based on this anecdotal evidence, we concluded that the activity immediately prior to and immediately following the introduction of TRAVIS CI might not be representative of the project's overall trends. Therefore, to limit the risk of bias due to extraordinary project activity in the transition period, we excluded one month of data centered around the adoption event in our quantitative analyses below.

Measures: We collected global and time-window measures:

- **total number of commits** in a project's history, as a proxy for project size / activity.
- **total number of commit authors**, as a proxy for the size of a project's community; commit authors include both core developers, who are also committers, and external contributors, without "write" access, whose commits are merged in by the core developers. Since it is common for open-source developers to contribute under different aliases (name-email tuples), *e.g.*, as a result of using different machines and changes in git settings over time, we first performed *identity merging*. We used heuristics that match first and last names, email prefixes, and email domains, cf. prior work [27], [28], and found an average of 1.15 (max 7) aliases per person in our dataset.
- **project age** at the time of adopting TRAVIS CI, in months, computed since the earliest recorded commit.
- **main programming language**, automatically computed by GITHUB based on the contents of each repository and extensions of file names therein.
- **number of non-merge commits**, **number of merge commits** per time window. Since git is a distributed version control system, developers can work locally, in increments, before pushing their changes to GITHUB or opening a pull request. E.g., a developer can choose to partition a large change into many smaller ones, and make multiple local commits to better manage the work. This would have no effect on CI runs, since CI is only triggered by GITHUB *push* events, and events happening after (and including when) a pull request is opened. Consequently, to study *Commits To the _Mainline_ Every Day* (Fowler's best practice), as opposed to potentially local git commits, we distinguish between *non-merge commits* and *merge commits* as a proxy. We recognize non-merge commits as those having at most one parent, and merge commits otherwise.
- **mean commit churn** per time window. Churn is the total number of lines added plus the total number of lines removed per commit (in git modified lines appear as first removed, then added), extracted from git logs. The mean is computed over all commits in that time window.
- **number of issues opened / closed**, **number of pull requests opened / closed** per time window, extracted using the GITHUB API.
- **mean pull request latency** per time window, in hours, computed as the difference between the timestamp when the PR was closed and that when it was opened. The mean is computed over all PRs in that time window.
- **number of tests executed per build**. Each TRAVIS build runs at least one job, corresponding to a particular build/test environment (*e.g.*, jdk version). Once the job starts, a log is generated, recording the detailed information of the build lifecycle, including installation steps and output produced by the build managers. On a sample of Java projects that used Maven, Ant, or Gradle as their build systems, for which we could make reasonable assumptions about the structure of their build log files, we parsed the TRAVIS build logs and extracted information about the number of tests executed (we take the maximum number of tests across jobs as the test count for a build), and the reasons causing builds to break (similar to [29]).

Filtering: As a large fraction of projects on GITHUB are small and not highly active [9], we filtered out projects inconsistently active during our 24-month observation period, to avoid biasing our conclusions due to an inflation of zero values in our data. Depending on the research question, this means either requiring at least one merge and one non-merge commit on the main branch / closed pull request / closed issue in *each* of the 24 time windows of observation. Furthermore, our multivariate regression analysis below requires enough variance along each of the dimensions being modeled, thus we additionally filter out programming languages not represented by many projects each. The resulting dataset spans seven popular programming languages: C, Java, Ruby, PHP, JavaScript, C++, and Python. Table I contains an overview.

B. Time Series Analysis Method

We use data visualization and statistical modeling to discover longitudinal patterns indicative of CI adoption effects. As one of our contributions, we introduce the statistical modeling framework of *regression discontinuity design* [30] to assess the existence and extent of a longitudinal effect, associated with the TRAVIS CI adoption.

To evaluate the effect of a treatment, *e.g.*, a new drug, on a disease progression, randomized experimental trials are

TABLE I
PROJECTS PER LANGUAGE, FOR DIFFERENT FILTERS.

Language	Commits	24 active periods with Pull reqs	Issues	All
C	30	13	13	6
Java	57	26	30	6
Ruby	54	26	37	5
PHP	71	32	33	14
JavaScript	75	38	69	18
C++	80	40	36	13
Python	100	55	44	15
Total	467	230	262	77

usually conducted: the experimental cohort is randomly split into a treatment group, *i.e.*, those given the treatment, and a control group, *i.e.*, those not given the treatment; then, the effect is evaluated based on the difference in disease progression between the two groups. In the absence of randomized trials, as is often the case with software engineering trace data, weaker techniques such as quasi-experiments are employed.

Among quasi-experimental designs to evaluate longitudinal effects of an intervention, regression discontinuity designs (RDDs) [31] are the most powerful. RDD is used to model the extent of a discontinuity of a function between its values at points just before/after an intervention. It is based on the assumption that in the absence of the intervention, the trend of the function would be continuous in the same way as prior to the intervention. Therefore, one can statistically assess how much an intervention (in our case the TRAVIS adoption) changed an outcome of interest, immediately and over time; and, if implemented using multiple regression, also evaluate whether the change could be attributed to other factors than the intervention. Figure 2 illustrates a discontinuity; the RDD approach, in a nutshell, aims to uncover the different regression lines before and after the discontinuity.

There are different formalizations of RDD, most prominently sharp RDD and fuzzy RDD [30]. To model the effect of CI adoption on developer practices, here we chose one implementation of the simpler, sharp RDD approach: *segmented regression analysis of interrupted time series data* [32]. We summarize our approach next, following the description by Wagner *et al.* [32], and refer to Figure 2. Let Y be the outcome variable in which we are looking for a discontinuity, *e.g.*, commit churn per month. We can specify the following linear regression model to estimate the level and trend in Y before CI adoption, and the changes in level and trend after CI adoption:

$$\begin{aligned} y_i =& \alpha + \beta \cdot \text{time}_i + \gamma \cdot \text{intervention}_i \\ &+ \delta \cdot \text{time\_after\_intervention}_i + \epsilon_i, \end{aligned}$$

where *time* indicates time in months at time i from the start of the observation period; *intervention* is an indicator for time i occurring before (intervention $= 0$) or after (intervention $= 1$) CI adoption, which was at month 0 based on our encoding in Figure 1; and *time after intervention* counts the number of months at time i after the intervention, coded 0 before CI adoption and (time $- 12$) after CI adoption.

This model encapsulates two separate regressions. For points before the treatment, the resulting regression line has a slope of β, and after the treatment $\beta+\delta$. The size of the effect of the treatment is the difference between the two regression values of y_i at the intervention time, and is equal to γ.

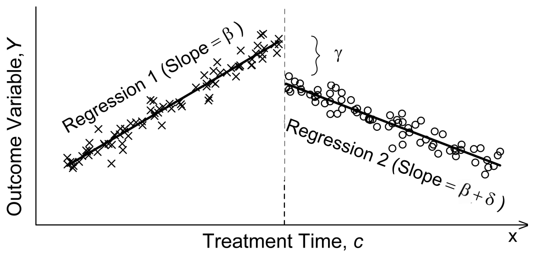

Fig. 2. RDD: the treatment effect (γ) is negative, and there is an interaction effect ($\delta \neq 0$) which changes the slope of the regression after the treatment.

Our goal is to capture, using the RDD model above, changes in trends after CI adoption across our sample of projects, while controlling for confounding variables (*e.g.*, project size, age, and programming language). Our data is centered at the time of CI adoption, and has an equal number of points, 12, on each side. Since the data is inherently nested (each project contributes multiple observations; similarly for programming language), we implement the RDD model as a mixed-effects linear regression (functions `lmer` and `lmer.test` in R) with a random-effects term for *project* and another random effects term for *programming language*; this way, we can capture project-to-project and language-to-language variability in the response. All other variables were modeled as fixed effects.

Solving the regression gives us the coefficients, which, if significant, can help us reason about the treatment and its effects, if any. We report on the models having significant coefficients in the regressions ($p < 0.05$) as well as their effect sizes, obtained from ANOVA analyses. Model fit was evaluated using a marginal (R^2_m) and a conditional (R^2_c) coefficient of determination for generalized mixed-effects models [33], [34]; R^2_m describes the proportion of variance explained by the fixed effects alone; R^2_c describes the proportion of variance explained by the fixed and random effects together. To improve robustness, the top 1% of the data was filtered out as outliers. Finally, we check for multicolinearity using VIF (below 3).

C. Survey

To obtain additional insights into the adoption of TRAVIS CI we conducted a survey with a sample of the maintainers responsible for introducing TRAVIS. We randomly selected 335 projects from our dataset, stratified by how much of a discontinuity in commit activity we detected at TRAVIS adoption time. For each project we identified the developer responsible for introducing TRAVIS, i.e., committing the first version of `.travis.yml`, and removed duplicates (same developer in different projects). In the invitation email (delivery of 23 messages failed; we received 55 responses, or 17.63% response rate) we explicitly stated the name of the project. We asked three questions: what made developers decide to start using CI and TRAVIS CI; whether they had to change anything in their development process to accommodate CI/TRAVIS; and how did their development process change with time, if at all, to use CI/TRAVIS CI efficiently. No question was mandatory.

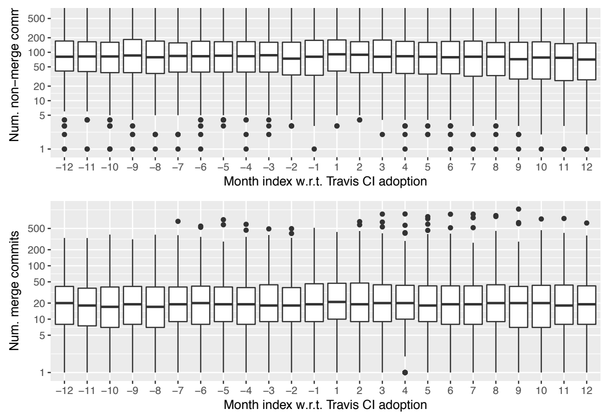

Fig. 3. Commit frequency before and after the TRAVIS CI adoption.

IV. RESULTS AND DISCUSSION

We discuss changes in development practices after CI adoption along four dimensions: commit frequency, code churn, pull requests resolution efficiency, issue tracking, and testing.

A. RQ_1: Trends in Commit Frequency

The first practice we examine is commit frequency. As we investigate the "Commits to the Mainline Every Day" practice, it is important to distinguish non-merge from merge commits. Indeed, local git commits, in a developer's offline repository, happen in isolation and can be seen as simply a mechanism to partition the work. However, what matters for TRAVIS CI are pushes and pull requests to the blessed GITHUB repository, *i.e.*, the "main" repository of the project as only then TRAVIS would be triggered. Push events are not readily accessible in our data, hence we analyze merge commits on the main development branch as a proxy for work that would have been subjected to CI by TRAVIS. Since not all our projects have recorded merge commits in *all* 24 periods, we restrict this analysis to only 575 projects that had at least one merge commit in each time window.

Exploratory Study: Figure 3 shows the boxplots of per-project number of non-merge commits (top) and number of merge commits (bottom), respectively, for each of twelve consecutive 30-day time intervals before and after the TRAVIS CI adoption. Note the log scale. Recall that due to instability, we omit the 30-day time interval centered around $t = 0$, when the earliest commit to the TRAVIS configuration file, which signals adoption, was recorded (Figure 1). The horizontal line in each boxplot is the median value across all projects.

First, we observe relative stability in the number of non-merge commits prior to the TRAVIS CI adoption (Figure 3, top), with the across-projects medians around 78 commits, and a slight decreasing trend after the adoption, with the across-projects median dropping to 67 commits in period 12. Second, after an initial dip in periods -12 to -10, the merge commits (Figure 3, bottom) appear to display a slight increasing trend prior to the TRAVIS CI adoption, with the across-projects median reaching 18 commits immediately prior to the adoption period, followed by apparent stabilization after that. We also observe a discontinuity at $t = 0$: the across-projects median is 21 commits right after the adoption period. Note, in both plots, the large variance in the data.

TABLE II
COMMIT FREQUENCY MODEL. THE RESPONSE IS **LOG(NUMBER OF MERGE COMMITS)** PER MONTH. $R^2_m = 0.58$. $R^2_c = 0.83$.

	Coeffs (Errors)	Sum Sq.
(Intercept)	-0.958 $(0.232)^{***}$	
log(TotalCommits)	-0.016 (0.032)	0.07
log(NumNonMergeCommits)	0.777 $(0.007)^{***}$	3095.91^{***}
AgeAtTravis	-0.001 (0.001)	0.96
log(NumAuthors)	0.120 $(0.026)^{***}$	5.67^{***}
time	0.019 $(0.002)^{**}$	30.39^{***}
interventionTRUE	-0.024 (0.027)	0.21
time_after_intervention	-0.016 $(0.003)^{**}$	10.96^{***}

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Statistical Modeling Study: We fitted a mixed-effects RDD model, as described before, to model trends in the *number of merge commits* per project, over time, as a function of TRAVIS CI adoption, and while controlling for confounds; most notably, we control for the *number of non-merge commits*, as these may have not all been subjected to CI, since they appear to display a decreasing trend with time.

We modeled a random intercept for *programming language*, to allow for language-to-language variability in the response (*i.e.*, code in some languages being naturally more verbose than in others, resulting in different ways to split the work across commits; or community-level norms for committing); we also modeled a random *intervention* slope and intercept for *project*, to allow for project-to-project variability in the response and the possibility that, on average, projects with lower initial activity may be less strongly affected by adopting TRAVIS CI than high-frequency projects. Recall also that the coefficient for *time* is the slope before adoption, the coefficient for *intervention* is the size of the effect of CI introduction, the coefficient for *time after intervention* is the divergence in the slopes before and after TRAVIS CI adoption, and the sum of the coefficients for *time* and *time after intervention* is the slope of the linear trend after TRAVIS CI adoption.

Table II summarizes the results. In addition to the model coefficients and corresponding standard errors, the table shows the sum of squares, a measure of variance explained, for each variable. The statistical significance is indicated by stars. The fixed-effects part of the model fits the data well ($R^2_m = 0.58$). There is also a considerable amount of variability explained by the random effects, *i.e.*, project-to-project and language-to-language differences, not explicitly modeled by our fixed effects ($R^2_c = 0.83$). Among the fixed effects, we observe that the *number of non-merge commits*, our main control, explains most of the variability in the response. The coefficient is positive, *i.e.*, the direction of the correlation is expected: other things constant, the more non-merge commits there are, the more merge commits there will be as well. Neither *project size* (total number of commits over the entire history) nor *project age* at adoption time have any statistically significant effects.

Next we turn to our TRAVIS-related variables. The model confirms a statistically significant, positive, baseline trend in the response with *time* (with a small effect), as suggested by

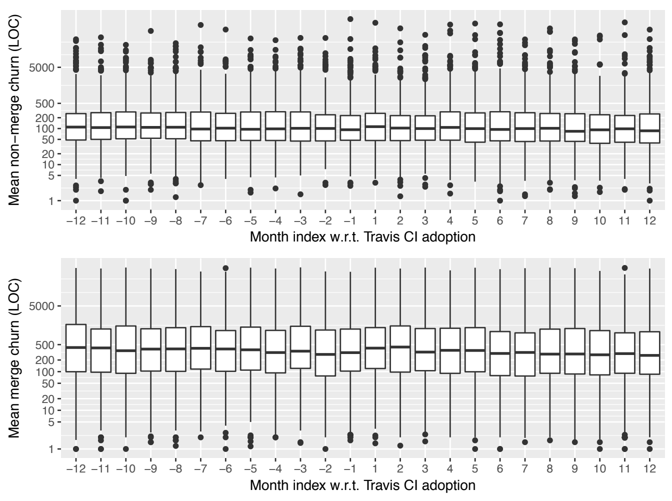

Fig. 4. Mean code churn per commit per project, before and after TRAVIS.

our exploratory study. The model does not detect any discontinuity at adoption time, since the coefficient for *intervention* is not statistically significant. Post adoption, there is a decrease in the slope of the time trend, but the overall trend remains ascending (the sum of the coefficients for *time* and *time after intervention* is positive): more merge-commits as time passes.

Discussion: The exploratory study suggests a slight decreasing trend in the number of non-merge commits with time. Separately modeling the trend in *number of non-merge commits* (not shown) using a similar approach as above confirmed this overall decreasing trend. This is consistent with the common observation that, as projects move past the initial development hurdle and age, development generally slows down, and the focus shifts to bug-fixing rather than adding new features, noted, *e.g.*, by Brindescu *et al.* [35].

In contrast, the statistical modeling study above revealed a consistent increase in *merge commits* with time, including post-CI adoption, albeit with a slowdown. This overall trend aligns with the expected increase in commit frequency after switching to CI, as expressed and encouraged by Fowler. Given the overall decreasing trend with time in the number of non-merge commits, the increase in merge commits is noteworthy. One explanation is that projects are switching to more distributed development workflows, using branches and pull requests. Indeed, in our survey, R38 has indicated that their project "shifted towards pull-request - merge development strategy as it made the distributed development more manageable." The development process change reported by R38 is not exceptional. Indeed, the idea of a shift towards more focused development in separated branches has been voiced by R32 ("shorter lived (and generally single contributor) branches") and R49 ("feature branches"), R6, and R47.

B. RQ$_2$: Trends in Code Churn

Next we examine code churn. As before, we distinguish between merge and non-merge commits.

TABLE III
COMMIT CHURN MODEL 1. THE RESPONSE IS **LOG(MEAN NON-MERGE COMMIT CHURN)**. $R^2_m = 0.09$. $R^2_c = 0.48$.

	Coeffs (Errors)	Sum Sq.
(Intercept)	$1.336\ (0.386)^{***}$	
log(TotalCommits)	$0.529\ (0.049)^{**}$	68.03^{***}
AgeAtTravis	$-0.003\ (0.001)^{*}$	0.03
log(NumAuthors)	$-0.233\ (0.041)^{**}$	14.42^{***}
time	$-0.007\ (0.004)$	1.57
interventionTRUE	$0.071\ (0.047)$	1.92
time_after_intervention	$-0.009\ (0.006)$	7.86^{*}

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

TABLE IV
COMMIT CHURN MODEL 2. THE RESPONSE IS **LOG(MEAN MERGE-COMMIT CHURN)**. $R^2_m = 0.20$. $R^2_c = 0.51$.

	Coeffs (Errors)	Sum Sq.
(Intercept)	$-1.297\ (0.484)^{**}$	
log(TotalCommits)	$1.113\ (0.062)^{**}$	625.20^{***}
AgeAtTravis	$-0.005\ (0.001)^{**}$	18.43^{**}
log(NumAuthors)	$-0.522\ (0.052)^{**}$	196.48^{***}
time	$-0.012\ (0.005)^{*}$	8.86^{*}
interventionTRUE	$0.220\ (0.066)^{**}$	21.75^{***}
time_after_intervention	$-0.022\ (0.008)^{**}$	15.51^{**}

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

Exploratory Study: Figure 4 shows boxplots of per-project per-commit mean code churn (medians behave similarly) for each of twelve consecutive 30-day time intervals before, and after, the TRAVIS CI adoption. The horizontal line in each boxplot is the median value over all projects.

In the non-merge commits, the medians are quite stable across the entire interval, from before to after the TRAVIS CI adoption, dancing around 100 lines of code churn per commit on average, with large variance. In comparison, in the merge-commits, we observe more than usual variance in the two months preceding the adoption, as well as a slight downward trend in the medians following adoption, which drop to about 263 lines of code churn per commit on average in the last period (12), compared to about 405 right after adoption (period 1). The variance around the medians is still large in all periods.

In conclusion, non-merge commits seem mostly unaffected by time passing and the TRAVIS CI adoption, while merge commits seem to be getting smaller with time.

Statistical Modeling Study: Guided by our exploratory observations above, we proceed to quantify the trends we observed using two mixed-effects RDD models, for non-merge and merge commits, respectively, as described before.

Table III summarizes the RDD model for non-merge commits. First, we observe that only the combined fixed-and-random effects model fits the data well ($R^2_c = 0.48$ compared to $R^2_m = 0.09$), *i.e.*, most of the explained variability in the data is attributed to project-to-project and language-to-language variability, rather than any of the fixed effects.

Next, we turn to the fixed effects. From among the controls, we note that overall bigger projects (*TotalCommits*) tend to churn more, other things held constant; and projects with a larger contributor base (*NumAuthors*), which typically indicate many occasional contributors, also tend to churn less. None of the TRAVIS-related predictors have statistically significant effects, *i.e.*, the churn trend in non-merge commits is stationary over time, and remains unaffected by the TRAVIS CI adoption.

Table IV summarizes the RDD model for merge commits. Similarly as with the previous model of churn in non-merge commits, most of the explained variability in the data is attributed to project-to-project and language-to-language variability, rather than any of the fixed effects variables. The controls also behave similarly as before: older projects tend to churn less, perhaps as they have reached maturity and a more stationary stage in their evolution; projects with a larger contributor base also tend to churn less, which is in line with the expectation that occasional contributors to open source projects are generally less active than core developers; small pull request contributions would be reflected here.

After controlling for confounds, we move on to the main time series predictors, all of which now have statistically significant effects. The coefficient for *time* is negative, suggesting a small decreasing baseline trend in commit churn before CI adoption; the *intervention* coefficient signals a discontinuity in the time series at the time of the TRAVIS CI adoption; the negative coefficient *time_after_intervention* signals an acceleration in the baseline trend. Together, this confirms a change in merge commit churn at the time of adoption, followed by an accelerated decreasing trend after adoption.

Discussion: Our modeling study revealed a statistically significant discontinuity in the merge commit churn time series when adopting TRAVIS CI; and a statistically significant decreasing linear trend with time in merge commit churn after adopting TRAVIS CI. However, churn in non-merge commits remains unaffected by either time or switching to TRAVIS CI.

The discontinuity is not unexpected, as one can reasonably expect more maintenance work in preparation for transitioning to TRAVIS CI, and some adjustment/cleanup period right after. The relative instability in code churn on both sides near the CI adoption time is also indication of this, as are the survey results. Indeed, when asked about the introduction of TRAVIS CI, respondents frequently refer to test automation being introduced around the same time as TRAVIS "as contributors couldn't be trusted to run test suite on their own" (R25). Furthermore, respondents indicate that TRAVIS has been introduced as "a part of automated package/release effort" (R38) and to "deploy artifacts to S3 on each commit, as part of our deployment process using Amazon CodeDeploy" (R34).

The decreasing trend in code churn post TRAVIS is visible only among merge commits, which are more likely to be affected by switching to a fast-paced CI workflow than non-merge commits, which may live on local, isolated developer branches for some time before being merged to the remote. This finding is consistent with Fowler's recommended good practices of CI, of committing smaller changes to the code. The expected decrease in the size of the commits is also echoed by one of the survey respondents: "commits became smaller and more frequent, to check the build; pull requests became easier to check" (R4). However, the decreasing trend we observed is not particularly steep. Survey responses provide a possible explanation: several developers referred to pull request integration as the reason for introducing TRAVIS CI,

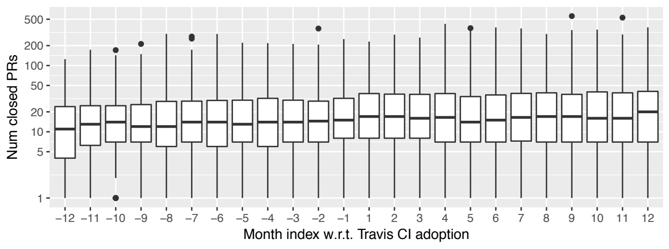

Fig. 5. Closed pull requests before and after the TRAVIS CI adoption.

TABLE V
PULL REQUEST MODEL. THE RESPONSE IS **LOG(NUMBER OF PULL REQUESTS CLOSED)** PER MONTH. $R^2_m = 0.32$. $R^2_c = 0.65$.

	Coeffs (Errors)	Sum Sq.
(Intercept)	$-3.132\ (0.356)^{***}$	
log(TotalCommits)	$0.605\ (0.049)^{**}$	71.17^{***}
AgeAtTravis	$-0.008\ (0.001)^{***}$	14.17^{***}
log(NumAuthors)	$0.136\ (0.043)^{**}$	4.64^{**}
time	$0.032\ (0.004)^{***}$	32.98^{***}
interventionTRUE	$0.005\ (0.055)$	0.00
time_after_intervention	$-0.033\ (0.005)^{***}$	18.06^{***}

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

and while R31 and R50 state that TRAVIS CI is used both for commits and for pull requests, R11, R21, and R37 explicitly state that in their projects push commits are outside the scope of TRAVIS CI. In other words, if in a project not all commits are subjected to TRAVIS CI, then there may be less incentive to follow Fowler's recommendations related to commit churn.

At the same time, a different respondent indicated that TRAVIS CI discourages him/her from making trivial commits (R11), suggesting instead that the commits he/she makes are likely to be larger; therefore, our global decreasing trend may be weakened by local trends in the opposite direction. Our model found evidence for strong project-specific effects: project-to-project and language-to-language differences, as captured by our random effects, contribute substantially to explaining the overall data variability. This suggests that any phenomena giving rise to pressures to increase or reduce code churn are perhaps subordinate in magnitude to other, more pressing phenomena of local, *i.e.*, project-specific, character.

Alternatively, the decreasing trend in commit churn is also consistent with the observation that as projects age, bug-fixing commits, which on average are smaller, become more common than new-feature commits, which on average are larger [35].

Our results suggest that while Fowler's good practice of small commits is followed to some extent, the project-to-project and language-to-language differences are more important and might overshadow the overall trend.

C. *RQ$_3$: Trends in Pull Request Closing*

Next we consider pull request closing, which we analyze along two dimensions: the number of pull requests closed, and the average pull request latency (*i.e.*, time to close) per time window. Since TRAVIS CI runs every time a pull request is submitted, the developers are rapidly notified whether their pull requests pass the TRAVIS quality gate, and can then rapidly react by correcting the pull request source code. Hence,

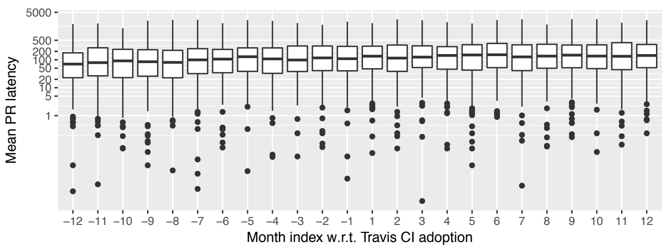

Fig. 6. Mean pull request latency before and after the TRAVIS CI adoption.

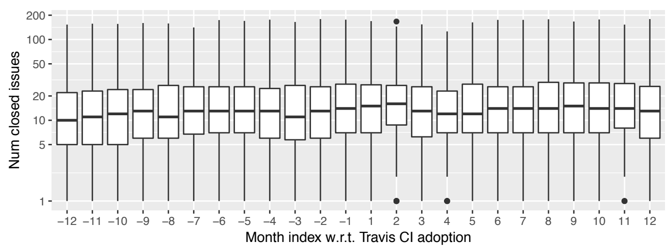

Fig. 7. Number of closed issues before and after the TRAVIS CI adoption.

TABLE VI
PULL REQUEST LATENCY MODEL. THE RESPONSE IS **LOG(MEAN PULL REQUEST LATENCY)** PER MONTH, IN HOURS. $R^2_m = 0.05$. $R^2_c = 0.31$.

	Coeffs (Errors)	Sum Sq.
(Intercept)	0.052 (0.519)	
log(TotalCommits)	0.157 $(0.068)^{*}$	7.89^{*}
AgeAtTravis	-0.002 (0.002)	1.00
log(NumAuthors)	0.225 $(0.062)^{**}$	19.72^{***}
time	0.030 $(0.012)^{*}$	8.60^{*}
interventionTRUE	-0.160 (0.153)	1.65
time_after_intervention	-0.022 (0.019)	2.03

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

TABLE VII
ISSUES MODEL. THE RESPONSE IS **LOG(NUMBER OF ISSUES CLOSED)** PER MONTH. $R^2_m = 0.17$. $R^2_c = 0.53$.

	Coeffs (Errors)	Sum Sq.
(Intercept)	-1.680 $(0.364)^{***}$	
log(TotalCommits)	0.443 $(0.049)^{**}$	42.54^{***}
AgeAtTravis	-0.006 $(0.001)^{***}$	8.83^{***}
log(NumAuthors)	0.119 $(0.039)^{**}$	4.72^{**}
time	0.021 $(0.004)^{***}$	16.00^{***}
interventionTRUE	0.030 (0.049)	0.19
time_after_intervention	-0.019 $(0.005)^{***}$	5.99^{***}

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

we expect that TRAVIS CI shortens the pull request latency and increases the number of closed pull requests.

Exploratory Study: Figure 5 shows boxplots of per-project number of closed pull requests. We observe that the median number of closed pull requests per months fluctuates between 11 and 15 before TRAVIS introduction, and between 14 and 20 after. We note an apparent increasing trend pre-TRAVIS, which also seems to continue post-TRAVIS. Both pre-TRAVIS and post-TRAVIS we see large variance around the medians.

Figure 6 provides a complementary perspective on the efficiency of pull request resolution, namely the pull request latency. The median latency seems to increase prior to introduction of TRAVIS CI, and continues to do so afterwards.

Statistical Modeling Study: We apply RDD as above. The statistical models for the number of pull requests closed and the mean pull request latency are summarized in Tables V and VI. As above, the combined fixed-and-random effects models fit the data much better than the basic fixed-effect models, indicating that the project-to-project and language-to-language variability are responsible for most of the variability explained.

Regarding closed pull requests (Table V), we note an increasing *time* baseline trend pre-adoption; no statistically significant discontinuity at the adoption time; and an apparent neutralization of the aforementioned time trend post-adoption, as β (*time*) $+ \gamma$ (*time_after_intervention*) $\simeq 0$. Turning to the pull request latency model (Table VI), we note a statistically significant, increasing baseline *time* trend, unaffected by the TRAVIS CI adoption, as neither coefficient for discontinuity and change in trend has statistically significant effects.

Discussion: The statistical modeling study generated two noteworthy findings. First, despite a visually apparent increasing trend in the number of pull requests closed per time period, across the entire interval under observation, our model suggests that, after controlling for project size, project age, and size of developer community, as well as local project-to-project and language-to-language differences, the increasing trend manifests itself, in aggregate, only pre-TRAVIS. Moreover, surprisingly, we find that the initial trend is flattened post-TRAVIS, resulting in relatively stationary behavior, on average. This paints a more nuanced picture of how GITHUB teams are adapting to, and benefiting from, continuous integration technology than suggested by prior work [12], and speaks to the strength of time-series-based analysis methods, such as the RDD we use here, at detecting fine resolution effects.

Second, our model finds that, on average, pull requests are taking longer to close over time, and this trend is unaffected by the switch to TRAVIS CI. This, again, is quite surprising, as we have seen in RQ$_2$ above that the size of code changes becomes, on average, smaller over time; in turn, this would imply that changes are also easier (quicker) to evaluate. One possible explanation for the increased latency is the TRAVIS' slowness, reported by some survey participants (R1, R27, R53).

D. RQ$_4$: Trends in Issue Closing

Next, we examine issue closing trends.

Exploratory Study: We follow the same approach as above and compare the medians in the number of issues per time period, in the months before and after CI adoption. Fig. 7 shows the boxplots of the number of issues per unit time period for 12 consecutive 30-day time intervals, before and after TRAVIS CI adoption. We note that the month immediately preceding and the two following the adoption exhibit the highest number of issues. Besides those, before CI adoption the median number of issues per period seems to vary between 10 and 13, with most of them being in between. Following CI adoption, we note a slight increasing trend in the median of the number of issues closed, with the median between 12 and 14.

Statistical Modeling Study: We apply RDD as above. The statistical model for the number of issues closed is summarized in Table VII. As above, the combined fixed-and-random effects models fit the data much better than the basic fixed-effect

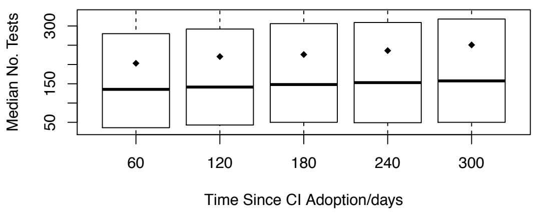

Fig. 8. Unit tests per build following CI adoption.

models, indicating that the project-to-project and language-to-language variability are responsible for most of the variability explained. We note an increasing *time* baseline trend pre-adoption; no statistically significant discontinuity at the adoption time; and a slight positive trend post-adoption, as β (*time*) + γ (*time_after_intervention*) > 0.

Discussion: The statistical modeling study confirmed the visually apparent increasing trend in the number of issues closed per time period, across the entire interval under observation, although the trend slows down following TRAVIS CI adoption, and is smaller than expected from the visual study, indicating the moderating influence of the control variables. Thus, our model finds that, on average, more issues are being closed over time, but this trend slows with the switch to TRAVIS CI.

Survey participants experience TRAVIS as beneficial for bug detection: "I think we produce less bugs" (R45), "there was an immediately noticeable improvement in terms of the number of serious bugs in production" (R51). An interesting insight into this matter is provided by R16: while several survey respondents have indicated that TRAVIS CI is not consistent in terms of performance and relatively slow, those performance aspects provided R16 with means to detect "more flaky issues" "which where only visible on TRAVIS".

E. RQ_5: Trends in Testing

The last development practice we examine is testing. Here we only consider a sample of Java projects, for which we could confidently identify their build system and, consequently, parse their TRAVIS log files. The sample consists of 250 projects, each with at least 100 builds, 90+% of builds executing tests.

Exploratory Study: As before, we first look for general trends. Fig. 8 shows a boxplot of per-project median number of tests per build, for each of five consecutive 60-day time intervals before CI adoption. We aggregated the data here in 60 day intervals to make it easier to visualize the trend, since the differences are small. The horizontal line in each boxplot is the median value of all per-project medians, and the black dot is their average value. We observe a monotonically increasing trend in both the medians, from 140 to 160 tests per build, and the means, from 205 to 245, *i.e.*, 15% to 20% increase.

To ascertain if the complexity of builds increases over time, we also calculated the average number of jobs per build. The median is 1 for all five time intervals. These two findings suggest strongly that there is an increase in the number of unit tests per build over time. This, coupled with our finding that builds are not getting more complex over time indicates that developers are likely spending more time on automated testing since CI adoption. This is consistent with Fowler's "good practices" proposal.

Error Types Study: We also looked at the evolution of the error types in builds over time after CI adoption. For all 250 projects above, we looked at the builds that resulted in errors, and deconvolved the errors into 8 types after open coding. We did this over 3 intervals: 0-60 days, 61-180 days, and 181-300 days (*i.e.*, corresponding to the first, third, and fifth interval in Figure 8). We find an apparent upward trend over time in most error type categories, most notably compile errors, execution errors, failed tests, and skipped test (median remained 2). All these are consistent with an increase in the amount of code being built and tested per build, as well as an increasing management of errors by skipping tests, likely to aid in debugging. On the other hand, we find a decreasing trend among errors related to missing files/dependencies, and timeouts, both consistent with those errors being less of an issue over time, as developers are acculturating to the CI mediated processes. Thus, overall we find an expected adjustment to the automated testing and error handling, with an indication that debugging complexity grows over time.

Improving software quality through test automation has been repeatedly mentioned by the survey respondents as a reason to switch to TRAVIS CI. By using TRAVIS CI they became more aware of testing-related aspects of their development process (R32, R52), spent more time and effort on designing testing strategies (R5, R53), and encourage other projects to embrace TRAVIS (R26). The efficiency of TRAVIS CI for long-running tests is, however, a concern (R10).

V. RELATED WORK

Impact of CI has attracted quite some attention from the research community. The earlier studies [36], [37] interpret CI as *distributed development and obligation to integrate one's own contributions*. Under this definition no commit size reduction has been observed in 5,122 Ohloh projects [37].

More recently, the ease of use of the TRAVIS CI [38] system led to its popularity on GITHUB, and triggered a series of research studies [5], [12], [22], [26], [39], [40]. This line of research is closer to the current work as it performs empirical analysis of TRAVIS CI data. Moreover, similarly to Vasilescu *et al.* [26] and Hilton *et al.* [5] our work can be seen as related to adoption of TRAVIS CI, and similarly to Beller *et al.* [22] we study build failures. Closest in spirit to ours is probably the work of Vasilescu *et al.* [12], who report, using different methodology, increases in a project team's ability to integrate outside contributions following adoption of TRAVIS CI. In our work we adopt a similar evolutionary perspective, focusing on changes in development practices in the projects before and after adoption of TRAVIS CI; most prior work has compared projects that adopt CI with others that do not.

Gousious *et al.* studied work practices and challenges in pull-based development on GITHUB [13], [41]. They report that 75% of the projects surveyed use CI tools to evaluate code quality [13] and that more than 60% of the surveyed contributors employ automatic testing [41]. Importance of tooling facilitating the testing tasks has been already recognized by Pham *et al.* [11]: back in 2012 when the authors conducted the interviews, TRAVIS CI has only started to support mainstream languages such as Java and lack of such tooling has been reported as an important challenge. These findings further emphasize importance of CI in modern software development.

A March 2016 survey of 1,060 IT professionals indicated that 81% of the enterprises and 70% small and midsize businesses implement DEVOPS [6]. Not surprisingly, industrial adoption of CI has attracted substantial research attention [7], [19], [42]–[46] and is a subject of recent literature survey by Eck *et al.* [47]. However, this line of work is based on interviews or surveys rather than on analysis of repository data and, as such, is to a larger extent susceptible to perception bias: *e.g.*, Leppanen *et al.* report that CI introduction is beneficial for productivity [19], Eck *et al.* stress that productivity is likely to decrease before positive effects can be observed [47], and Ståhl and Bosch validated this hypothesis only partially [46].

Automated testing is an important factor that affects the cost-effectiveness of CI and much effort has been devoted to improve the quality and efficiency of automated testing in CI. Campos et al. [48] enhanced CI with automated test generation. Elbaum *et al.* [49] and Hu *et al.* [50] applied test case selection and test case prioritization to test suites. Dösinger *et al.* [51] proposed the continuous change impact analysis technique to improve the effectiveness of automated testing. Long *et al.* [52] designed tools to support collaborative testing between testers. Nilsson et al. [53] developed a technique to visualize end-to-end testing activities in CI processes. All these efforts aim at improving the cost-effectiveness of CI.

VI. THREATS TO VALIDITY

We focus on construct and internal validity [54], as we do not claim generalizability.

Construct Validity: One of our constructs is a project's "CI adoption time," operationalized as "first commit to `.travis.yml`". A more precise operationalization would have involved reconciling two additional timestamps: registration of the repository with TRAVIS CI, and first build; the three timestamps may not coincide, *e.g.*, because TRAVIS CI can also start a build using some default environment settings (Ruby) *without* `.travis.yml` being present. Hence, validity of the "CI adoption time" construct might have been threatened by our operationalization. We reduce this threat by excluding the period immediately before and after our $t = 0$ from all analyses. Another construct is "size of a code change". We operationalize this as the number of churned lines, customarily interpreted as the sum of the number of added and removed lines [55]. In this way, moved lines are counted twice, as being added and as being removed. Moreover, we do not distinguish between lines of source code and other lines, since it is non-trivial when dealing with many languages.

Internal Validity: To reduce these threats we have opted for RDD [30], a sound approach to statistical modeling of discontinuity in time series. Application of RDD to SE data has been recently advocated by Wieringa [56].

Multiple data filtering steps have been applied above. We tested the robustness of our models by varying the data filtering criteria (*e.g.*, 9 month windows instead of 12), and observed similar phenomena. We encourage independent replications to further assess the robustness of our results.

VII. CONCLUSION

This paper focused on the switch to continuous integration (CI): while several guidelines exist, relatively little has been done to evaluate the state of practice. We empirically studied the impact of adopting TRAVIS CI on development practices in a collection of GITHUB projects, and surveyed the developers responsible for introducing TRAVIS in those projects.

We find that the reality of adopting TRAVIS CI is much more complex than suggested by previous work. The increasing number of merge commits aligns with the "commit often" guideline, but is likely to be further encouraged by the shift to a more distributed workflow with multiple branches and pull requests (RQ_1). The "commit small" guideline, however, is followed only to some extent, with large variation between projects (RQ_2). As expected, we observe a general increasing trend in the number of issues closed over time; however, it was surprising that this trend slows down after TRAVIS CI is introduced (RQ_4). In terms of testing, we find that after some (expected) initial adjustments, the amount (and potentially the quality) of automated tests seems to increase (RQ_5).

The most interesting observations relate to pull request (PR) evaluations: While we also find that, in aggregate, more PRs are being closed after adopting TRAVIS, as did prior work [12], our time-series-based analysis suggests that the expected increasing trend in PRs closed over time manifests itself *only before* adopting TRAVIS; after, the number of closed PRs remains relatively stable (RQ_3). At the same time, PR latencies increase steadily over time, despite the code changes getting relatively smaller. Future work should employ qualitative methods to understand potential causes for this effect. We can only speculate here that even with the high level of automation provided by CI, the ability of teams to scale up distributed development is limited by the availability of human resources for manual code review (*i.e.*, the project integrators). This calls for a more profound investigation of how GITHUB teams change their PR review practices in response to the introduction of TRAVIS CI, as well as additional tool support, *e.g.*, for automatic prioritization and automatic post-merge defect prediction, which may help. Future work should also explicitly consider the design decisions and trade-offs between seemingly equivalent CI pipeline implementations. Project specific concerns may drive individual implementations, usage, and practices, as the high amount of variance explained by our project random effects suggest.

Acknowledgements. BV and VF are supported by the NSF (1717415, 1717370). YZ is supported by the National Natural Science Foundation of Jiangsu Province (BK20130014).

References

[1] D. DeGrandis, "DevOps: So you say you want a revolution?" *Cutter IT Journal*, vol. 24, no. 8, p. 34, 2011.

[2] M. Loukides, *What is DevOps?* O'Reilly Media, Inc., 2012.

[3] J. Humble and J. Molesky, "Why enterprises must adopt DevOps to enable continuous delivery," *Cutter IT Journal*, vol. 24, no. 8, p. 6, 2011.

[4] J. Roche, "Adopting DevOps practices in quality assurance," *Communications of the ACM*, vol. 56, no. 11, pp. 38–43, 2013.

[5] 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.

[6] RightScale, "State of the cloud report: DevOps trends," http://www.rightscale.com/blog/cloud-industry-insights/new-devops-trends-2016-state-cloud-survey, 2016.

[7] M. Hilton, N. Nelson, D. Dig, T. Tunnell, D. Marinov *et al.*, "Continuous integration (CI) needs and wishes for developers of proprietary code," Corvallis, OR: Oregon State University, Dept. of Computer Science, Tech. Rep., 2016.

[8] M. Fowler, "Continuous integration," http://martinfowler.com/articles/originalContinuousIntegration.html, 2000, accessed: 2016-10-8.

[9] G. Gousios, M. Pinzger, and A. v. Deursen, "An exploratory study of the pull-based software development model," in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 345–355.

[10] S. Stolberg, "Enabling agile testing through continuous integration," in *Agile Conference (AGILE)*. IEEE, 2009, pp. 369–374.

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

[12] B. Vasilescu, Y. Yu, H. Wang, P. T. 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.

[13] 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.

[14] Y. Bugayenko, "Continuous integration is dead," http://www.yegor256.com/2014/10/08/continuous-integration-is-dead.html, 2014, accessed: 2016-10-8.

[15] M. Fowler, "Continuous integration," http://martinfowler.com/articles/continuousIntegration.html, 2006, accessed: 2016-10-8.

[16] P. Duvall, S. M. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[17] A. Miller, "A hundred days of continuous integration," in *Agile Conference (AGILE)*, 2008, pp. 289–293.

[18] H. M. Yüksel, E. Tüzün, E. Gelirli, E. Bıyıklı, and B. Baykal, "Using continuous integration and automated test techniques for a robust C4ISR system," in *International Symposium on Computer and Information Sciences*, 2009, pp. 743–748.

[19] M. Leppanen, S. Makinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Mannisto, "The highways and country roads to continuous deployment," *IEEE Software*, vol. 32, no. 2, pp. 64–72, 2015.

[20] F. Cannizzo, R. Clutton, and R. Ramesh, "Pushing the boundaries of testing and continuous integration," in *Agile Conference (AGILE)*, 2008, pp. 501–505.

[21] M. Coram and S. Bohner, "The impact of agile methods on software project management," in *International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, 2005, pp. 363–370.

[22] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An analysis of Travis CI builds with GitHub," *PeerJ PrePrints*, vol. 4, p. e1984, 2016.

[23] G. Gousios and D. Spinellis, "GHTorrent: GitHub's data from a firehose," in *Working Conference on Mining Software Repositories (MSR)*. IEEE, 2012, pp. 12–21.

[24] B. Vasilescu, K. Blincoe, Q. Xuan, C. Casalnuovo, D. Damian, P. Devanbu, and V. Filkov, "The sky is not the limit: Multitasking on GitHub projects," in *International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 994–1005.

[25] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration," in *International Conference on Mining Software Repositories (MSR)*. ACM, 2017.

[26] 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 *International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2014, pp. 401–405.

[27] C. Bird, A. Gourley, P. Devanbu, M. Gertz, and A. Swaminathan, "Mining email social networks," in *International Workshop on Mining Software Repositories (MSR)*. ACM, 2006, pp. 137–143.

[28] B. Vasilescu, A. Serebrenik, and V. Filkov, "A data set for social diversity studies of GitHub teams," in *Working Conference on Mining Software Repositories (MSR)*. IEEE, 2015, pp. 514–517.

[29] 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 *International Conference on Mining Software Repositories (MSR)*. IEEE, 2017.

[30] G. W. Imbens and T. Lemieux, "Regression discontinuity designs: A guide to practice," *Journal of Econometrics*, vol. 142, no. 2, pp. 615–635, 2008.

[31] T. D. Cook, D. T. Campbell, and A. Day, *Quasi-experimentation: Design & analysis issues for field settings*. Houghton Mifflin Boston, 1979, vol. 351.

[32] 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, no. 4, pp. 299–309, 2002.

[33] S. Nakagawa and H. Schielzeth, "A general and simple method for obtaining r^2 from generalized linear mixed-effects models," *Methods in Ecology and Evolution*, vol. 4, no. 2, pp. 133–142, 2013.

[34] P. C. Johnson, "Extension of nakagawa & schielzeth's r^2_{GLMM} to random slopes models," *Methods in Ecology and Evolution*, vol. 5, no. 9, pp. 944–946, 2014.

[35] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *International Conference on Software Engineering (ICSE)*. ACM, 2014, pp. 322–333.

[36] J. Holck and N. Jørgensen, "Continuous integration and quality assurance: a case study of two open source projects," *Australasian J. of Inf. Systems*, vol. 11, no. 1, 2003.

[37] A. Deshpande and D. Riehle, *Continuous Integration in Open Source Software Development*. Boston, MA: Springer US, 2008, pp. 273–280.

[38] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, 2014.

[39] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on GitHub," in *Working Conference on Mining Software Repositories (MSR)*. IEEE, 2015, pp. 367–371.

[40] 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, no. 8, pp. 1–14, 2016.

[41] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: the contributor's perspective," in *International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 285–296.

[42] E. Laukkanen, M. Paasivaara, and T. Arvonen, "Stakeholder perceptions of the adoption of continuous integration – a case study," in *Agile Conference (AGILE)*, 2015, pp. 11–20.

[43] A. Debbiche, M. Dienér, and R. Berntsson Svensson, *Challenges When Adopting Continuous Integration: A Case Study*. Cham: Springer International Publishing, 2014, pp. 17–32.

[44] D. Ståhl and J. Bosch, "Automated software integration flows in industry: A multiple-case study," in *International Conference on Software Engineering (ICSE) Companion*. ACM, 2014, pp. 54–63.

[45] ——, "Modeling continuous integration practice differences in industry software development," *Journal of Systems and Software*, vol. 87, pp. 48–59, 2014.

[46] ——, "Experienced benefits of continuous integration in industry software product development: A case study," in *IASTED International Conference on Software Engineering*, 2013, pp. 736–743.

[47] A. Eck, F. Uebernickel, and W. Brenner, "Fit for continuous integration: How organizations assimilate an agile practice," in *20th Americas Conference on Information Systems (AMCIS)*. AIS, 2014.

[48] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, “Continuous test generation: enhancing continuous integration with automated test generation,” in *International Conference on Automated Software Engineering (ASE)*. ACM, 2014, pp. 55–66.

[49] S. Elbaum, G. Rothermel, and J. Penix, “Techniques for improving regression testing in continuous integration development environments,” in *International Symposium on Foundations of Software Engineering (FSE)*. ACM, 2014, pp. 235–245.

[50] J. Hu and Y. Li, “Implementation and evaluation of automatic prioritization for continuous integration test cases,” Master’s thesis, Chalmers University of Technology — University of Gothenburg, 2016.

[51] S. Dösinger, R. Mordinyi, and S. Biffl, “Communicating continuous integration servers for increasing effectiveness of automated testing,” in *International Conference on Automated Software Engineering (ASE)*. ACM, 2012, pp. 374–377.

[52] T. Long, “Collaborative testing across shared software components (doctoral symposium),” in *International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2015, pp. 436–439.

[53] A. Nilsson, J. Bosch, and C. Berger, “Visualizing testing activities to support continuous integration: A multiple case study,” in *International Conference on Agile Software Development*. Springer, 2014, pp. 171–186.

[54] D. E. Perry, A. A. Porter, and L. G. Votta, “Empirical studies of software engineering: a roadmap,” in *International Conference on Software Engineering (ICSE)*. ACM, 2000, pp. 345–355.

[55] E. Giger, M. Pinzger, and H. C. Gall, “Comparing fine-grained source code changes and code churn for bug prediction,” in *Working Conference on Mining Software Repositories (MSR)*. ACM, 2011, pp. 83–92.

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