Quality and Productivity Outcomes Relating to Continuous Integration in GitHub

Bogdan Vasilescu[†*], Yue Yu[‡†*], Huaimin Wang[‡], Premkumar Devanbu[†], Vladimir Filkov[†]

[†]Department of Computer Science
University of California, Davis
Davis, CA 95616, USA
{vasilescu, ptdevanbu, vfilkov}@ucdavis.edu

[‡]College of Computer
National University of Defense Technology
Changsha, 410073, China
{yuyue, hmwang}@nudt.edu.cn

ABSTRACT

Software processes comprise many steps; coding is followed by building, integration testing, system testing, deployment, operations, among others. Software process integration and automation have been areas of key concern in software engineering, ever since the pioneering work of Osterweil; market pressures for Agility, and open, decentralized, software development have provided additional pressures for progress in this area. But do these innovations actually help projects? Given the numerous confounding factors that can influence project performance, it can be a challenge to discern the effects of process integration and automation. Software project ecosystems such as GitHub provide a new opportunity in this regard: one can readily find large numbers of projects in various stages of process integration and automation, and gather data on various influencing factors as well as productivity and quality outcomes. In this paper we use large, historical data on process metrics and outcomes in GitHub projects to discern the effects of one specific innovation in process automation: *continuous integration*. Our main finding is that continuous integration improves the productivity of project teams, who can integrate more outside contributions, without an observable diminishment in code quality.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Experimentation, Human Factors

Keywords

Continuous integration, GitHub, pull requests

*Bogdan Vasilescu and Yue Yu are both first authors, and contributed equally to the work.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786850

1. INTRODUCTION

Innovations in software technology are central to economic growth. People place ever-increasing demands on software, in terms of features, security, reliability, cost, and ubiquity; and these demands come at an increasingly faster rate. As the appetites grow for ever more powerful software, the human teams working on them have to grow, and work more efficiently together.

Modern games, for example, require very large bodies of code, matched by teams in the tens and hundreds of developers, and development time in years. Meanwhile, teams are globally distributed, and sometimes (*e.g.*, with open source software development) even have no centralized control. Keeping up with market demands in an agile, organized, repeatable fashion, with little or no centralized control, requires a variety of approaches, including the adoption of technology to enable process automation. Process Automation *per se* is an old idea, going back to the pioneering work of Osterweil [32]; but recent trends such as open-source, distributed development, cloud computing, and software-as-a-service, have increased demands for this technology, and led to many innovations. Examples of such innovations are distributed collaborative technologies like `git` repositories, forking, pull requests, continuous integration, and the DEVOPS movement [36]. Despite rapid changes, it is difficult to know how much these innovations are helping improve project outcomes such as productivity and quality. A great many factors such as code size, age, team size, and user interest can influence outcomes; therefore, teasing out the effect of any kind of technological or process innovation can be a challenge.

The GitHub ecosystem provides a very timely opportunity for study of this specific issue. It is very popular (increasingly so) and hosts a tremendous diversity of projects. GitHub also comprises a variety of technologies for distributed, decentralized, social software development, comprising version control, social networking features, and process automation. The development process on GitHub is more democratic than most open-source projects: *anyone* can submit contributions in the form of *pull requests*. A pull request is a candidate, proposed code change, sometimes responsive to a previously submitted modification request (or *issue*). These pull requests are reviewed by project insiders (*aka* core developers, or integrators), and accepted if deemed of sufficient quality and utility. Projects that are more popular and widely used can be expected to attract more interest, and more pull requests; these will have to be

built, tested, and reviewed by core developers prior to actual inclusion. This process can slow down, given the limited bandwidth of core developers; thus popular, innovative, and agile projects need process automation. One key innovation is the idea of *continuous integration* ($\mathcal{CI}$); essentially, $\mathcal{CI}$ attempts to automatically build and deploy the software in a "sandbox", and automatically run a collection of tests when the pull request is received. By automating these steps, a project can hope to gain both productivity (more pull requests accepted) and quality (the accepted pull requests are prescreened by the automation provided by $\mathcal{CI}$).

Starting from a newly mined data set of the usage of $\mathcal{CI}$ in GitHub projects, in this paper we looked at the software engineering outcomes which present differentially with the introduction of $\mathcal{CI}$ versus without. In particular, our contributions are:

- We collected a comprehensive data set of 246 GitHub projects which at some point in their history added the Travis-CI functionality to the development process. Our data is available online at `https://github.com/yuyue/pullreq_ci`.
- We found that after $\mathcal{CI}$ is added, more pull requests from core developers are accepted, and fewer are rejected; and fewer submissions from non-core developers get rejected. This suggests that $\mathcal{CI}$ both improves the handling of pull requests from insiders, and has an overall positive effect on the initial quality of outside submissions.
- Despite the increased volume of pull requests accepted, we found that introduction of $\mathcal{CI}$ is *not* associated with any diminishment of user-reported bugs, thus suggesting that user-experienced quality is not negatively affected. We did see an increase in developer reported bugs, which suggests that $\mathcal{CI}$ is helping developers discover more defects.

2. BACKGROUND

The focus of our work is the effect of continuous integration in the context of open source projects that use the pull request based model of development. We begin with some background.

2.1 Continuous Integration

The concept of $\mathcal{CI}$ is often attributed to Martin Fowler based on a 2000 blog entry [12]. The basic notion is that all developers' work within a team is continually compiled, built, and tested. This process is a perpetual check on the quality of contributed code, and mitigates the risk of "breaking the build", or worse, because of major, incompatible changes by different people or different sub-teams. Arguably, $\mathcal{CI}$ originated from the imperatives of agility [11], *viz.*, responding quickly to customer requirements. It can be viewed as a type of process automation; rather than wait for some sort of human-controlled gate-keeping of code prior to building and integration testing, automated mechanisms are incorporated into the development environment to carry out these steps continually and automatically. In software engineering, continuous integration is viewed as a paradigm shift, "perhaps as important as using version control" [22]. Without $\mathcal{CI}$, software is considered broken until proven to work, typically during a testing or integration stage. With $\mathcal{CI}$, assuming a comprehensive automated test suite, software is proven to work with every new change, and serious regressions can be detected and fixed immediately.

In the context of distributed, globalized development, cultural, geographical and time differences raise the spectre of process variation and non-repeatability, and thus amplify the imperatives to adopt process automation. This applies even more strongly to open source software (OSS) projects where, in addition to the above issues, volunteers are involved, and there is also a lack of centralized control [10, 20, 21]. $\mathcal{CI}$ has become quite popular in OSS projects, and many projects in GitHub are using it [19]. Numerous tools that support $\mathcal{CI}$ exist [29]. Therefore, one might expect that these teams are seeing benefits from adopting $\mathcal{CI}$, and that one could obtain quantitative evidence of these benefits.

2.2 Pull-based Software Development

The pull-based development model [2], used to integrate incoming changes into a project's codebase, is becoming the de facto contribution model in distributed software teams [19]. Enabled by `git`, pull-based development means that contributors to a software project can propose changes without the need for them to share access to a central repository; instead, contributors can work locally in a different branch or fork (local clone) of the central repository and, whenever ready, request to have their changes merged into the main branch by submitting a *pull request*.

Compared to patch submission and acceptance via mailing lists and issue tracking systems, which has been the traditional model of collaboration in open source [5, 15], the pull-based model offers several advantages, including centralization of information (*e.g.*, the contributed code—the patch—resides in the same source control management system as the rest of the system, therefore authorship information is effortlessly maintained; on modern collaborative coding platforms such as BitBucket, Gitorius, and GitHub, a wealth of data about the submitter's track record is publicly available to project managers via user profile pages [8,28]) and process automation (*e.g.*, GitHub provides integrated functionality for pull request generation, automatic testing, contextual discussion, in-line code review, and merger).

By "decoupling the development effort from the decision to incorporate the results of the development in the code base" [17], the pull-based model also offers an unprecedentedly low barrier to entry for potential contributors (*i.e.*, the so-called "drive-by" commits [35]), since anyone can fork and submit pull requests to any repository. Therefore, projects can use pull requests complementarily to the shared repository model, such that core team members push their changes directly, and outside contributors submit changes via pull requests. However, projects can also use pull requests in many scenarios beyond basic patch submission, *e.g.*, for conducting code reviews, and discussing new features [19]. As a result, in many projects all contributions are submitted as pull requests, irrespective of whether they come from core developers with write access to the repository or from outsiders, which ensures they adhere to the same evaluation process (*e.g.*, only reviewed code gets merged) [19].

The pull-based model is widely used. For example, on GitHub alone, almost half of all collaborative projects use pull requests [19], and this number is only expected to grow. *Ruby on Rails*,[1] one of the most popular projects on GitHub,

[1] `https://github.com/rails/rails`

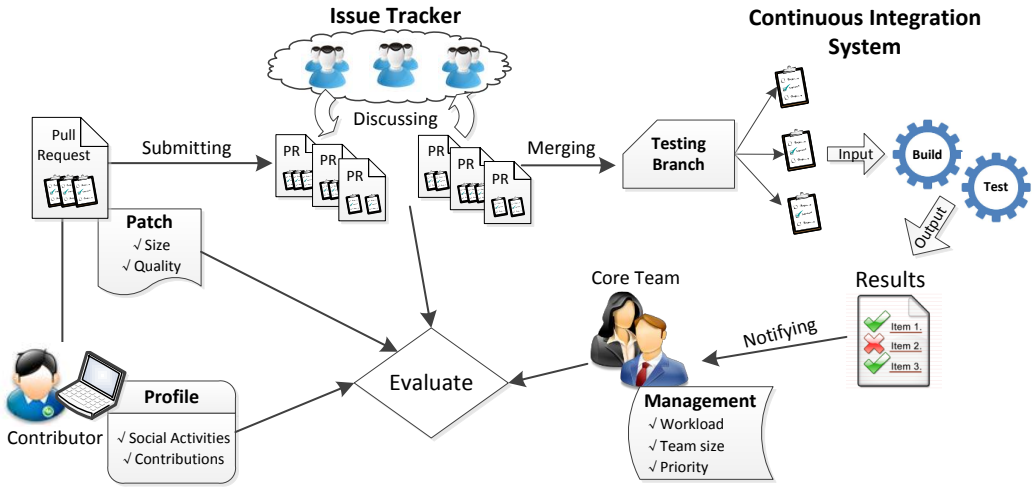

Figure 1: Overview of the pull request evaluation process

receives upwards of three hundred new pull requests each month. The high volume of incoming pull requests poses a serious challenge to project integrators, *i.e.*, the core team members responsible for evaluating the proposed changes, deciding whether to apply them or not, and integrating them into the main development branch [19, 35, 45]. Integrators play a crucial role in pull-based development [9,19]. Dabbish *et al.* [9] identify management of pull requests as the most important project activity on GitHub. Gousios *et al.* [19] consider integrators to be "guardians for the project's quality". They must ensure not only that pull requests are evaluated in a timely matter and eventually accepted, to secure the project's growth, but also that all contributions meet the project's quality standards.

2.3 Pull Request Evaluation

Prior work on pull request evaluation on social coding platforms like GitHub [17, 19, 45, 46] points to a complex process influenced by a multitude of social and technical factors. The level of transparency available on GitHub, where a number of social signals become more readily available, has shaped the way developers make inferences about each other and their work [8,28]. For example, the integrated social media features (*e.g.*, *following* other developers, *watching* repositories and *commenting* on pull requests) enable participants in these communities to build social relationships [3, 8, 40, 52, 53], while public profile pages make salient information about one's track record as developer [8, 19, 27] and even demographic features (*e.g.*, gender [47, 48]). Integrators use social signals to build trust in the submitted contributions, *e.g.*, they build mental profiles of the submitters' competence by evaluating their track record, and they base judgements of the contributions on personal relationships with the submitters [19, 28, 45]. Prior work has also uncovered which technical factors are associated with contribution acceptance, *e.g.*, code quality [19, 45], adherence to coding styles and project conventions [19], existence of testing code in the pull request [19, 45], and how active the particular project area affected by the pull request is [17].

In trying to handle pull requests efficiently without compromising software quality, especially when faced with an increasing volume of incoming pull requests, integrators often resort to automated testing, as supported by $\mathcal{CI}$ services [19, 35]. On GitHub, 75% of projects that use pull requests frequently also use $\mathcal{CI}$, either in hosted services (*e.g.*, Travis-CI [49], Jenkins [29]) or in standalone setups [19]. In prior work [51], we found that presence of $\mathcal{CI}$ plays a prominent role in the pull request evaluation process, being a strong positive predictor of pull request evaluation latency. The following is a simplified description of how $\mathcal{CI}$ is involved in pull request evaluation (from [51]). Whenever a new pull request is received by a project using $\mathcal{CI}$, the contribution is merged automatically into a testing branch, and the existing test suites are run. If tests fail, the pull request is typically rejected (closed and not merged in GitHub parlance) by one of the integrators, who may also comment on why it is inappropriate and how it can be improved. If tests pass, core team members proceed to do a team-wide code review, by commenting inline on (parts of) the code, including requests for modifications to be carried out by the submitter (who can then update the pull request with new code), if necessary. After a cycle of comments and revisions, and if everyone is satisfied, the pull request is closed and merged. In rare cases, pull requests are merged even if (some) tests failed. Only core team members (integrators) and the submitter can close (to merge or reject—integrators; to withdraw—submitter) and reopen pull requests. The process is summarized in Figure 1.

The continuous application of quality control checks (as imposed by $\mathcal{CI}$) aims to speed up the development process and to ultimately improve software quality [11]. For example, Addam Hardy, a developer active on GitHub, explains in his blog:[2]

> *[CI] enables us to automate more of our process which frees us up to focus on the important things — like implementing and shipping features! [...] [The integration of CI in GitHub] enables the team to rapidly find integration errors or regression failures in the test suite. This tightens the feedback loop and not only enables more defect free code, but greatly speeds up our process.*

[2] http://addamhardy.com/blog/2013/09/28/automate-all-the-things-continuous-integration-and-continuous-deployment-at-revunit/

2.4 Research Questions

Prior work in evaluating such beliefs has been mostly qualitative and survey based. Based on the conceptual foundation laid by these studies, reviewed above, the central aim of this paper is to quantitatively explore effects associated with the adoption of $\mathcal{CI}$ in GitHub projects that use the pull request model. We explicate our research questions. To begin, there is a strong and immediate expectation that $\mathcal{CI}$ *should improve productivity* of both teams and individuals. Staal and Bosch [42], for example, argue with some survey-based evidence [41], that build and test automation saves programmers time for more creative work, and thus should increase productivity. Stolberg [43] argues that $\mathcal{CI}$ leads to improved integration intervals, and thus faster delivery of software. Miller [30] reports the experiences with $\mathcal{CI}$ of one distributed team at Microsoft in 2007, and estimates that moving to a $\mathcal{CI}$-driven process may achieve a 40% reduction in check-in overhead when compared to a check-in process that doesn't leverage CI, for the same code base and product quality. Bhattacharya [4] reports on an industrial case study from an insurance company, where developer productivity increases through the usage of $\mathcal{CI}$. However, not all studies agree on effects associated with adoption of $\mathcal{CI}$, *e.g.*, Parsons *et al.* [33] find no clear benefits of $\mathcal{CI}$ on either productivity or quality. These prior studies prompt a large-scale, quantitative analysis to determine the productivity effects of $\mathcal{CI}$ on distributed teams. We ask:

RQ1. Productivity.
How is the productivity of teams affected by $\mathcal{CI}$?

A key aspect of $\mathcal{CI}$ is *testing*; integration, and unit (and perhaps other) tests are run automatically with every change. Testing is, after all, all about finding bugs. Indeed, Fowler, one of early proponents of $\mathcal{CI}$, has emphasized the *benefits to software quality* [12]. Other researchers make supporting claims about automated testing. For example, Karhu *et al.* [24] report on an industrial case study conducted in five commercial organizations, and find, using interview data, that testing automation leads to fewer defects. Rady and Coffin [38] claim, without providing additional details, that "many software development teams have recognized the design and quality benefits of creating automated test suites". The literature also points to indirect benefits of using $\mathcal{CI}$ on software quality, related to test quality increases [38], *e.g.*, through test coverage increases [29]. Fortunately, in GitHub there are a range of projects that have adopted $\mathcal{CI}$ to varying degrees, making it possible to study the effects of $\mathcal{CI}$ on quality. We ask:

RQ2. Quality.
What is the effect of $\mathcal{CI}$ on software quality?

It should be noted here that there are many factors that could influence productivity and quality in software projects, and if we are to tease out the effect of $\mathcal{CI}$ *per se*, we shall need enough data to provide adequate controls for all these factors. Furthermore, the introduction of $\mathcal{CI}$ could be expected to affect productivity and quality in a variety of complex ways, including second-order effects. For example, the use $\mathcal{CI}$ might attract more people to the project; if pull requests are accepted expediently, more contributions might arrive, and more people might be induced to contribute. The test-automation convenience of $\mathcal{CI}$ might encourage people to write more tests; contrariwise, the very ability of $\mathcal{CI}$ to detect more defects might lead people to inaccurately perceive a diminishment of quality; it might also encourage people to submit higher-quality pull requests, for fear of failing a lot of tests, and thus losing hard-won community status.

For the purposes of this study, we adopt two experimental postures (discussed in more detail below). First, we gather metrics on a large number of projects, over a significant period of time, focusing on a broad set of aspects that are known to affect the rate of growth of projects' source base, and the quality thereof. By controlling for several known factors that affect productivity and quality, we hope to discern the effects of $\mathcal{CI}$ *per se*. Second, in this paper we consider specifically the overall quality and productivity effects of $\mathcal{CI}$, without delving into further causal analysis as to whether the effects are first- or second-order; that is left for future work.

3. METHODS

3.1 Data Collection

3.1.1 Selecting Projects

Our goal was to identify projects with sufficiently long historical records, which had also switched to continuous integration at some point; this would enable us to model the effects of $\mathcal{CI}$. We began with the GHTorrent [16] dump dated 10/11/2014. We focused on main-line projects (*i.e.*, not forks) written in the most popular languages[3][4] on GitHub: Ruby, Python, JavaScript, PHP, Java, Scala, C and C++. We excluded projects with fewer than 200 pull requests; we only focused on projects where pull requests were integral to the development effort, rather than being an infrequent modality adopted by occasional external contributors.

This set of selection criteria resulted in a candidate set of 1,884 projects. Then, for each of them, we checked whether it used a $\mathcal{CI}$ service or not. There are two popular $\mathcal{CI}$ services used by projects in GitHub, Travis-CI and Jenkins [29] (in addition to others, used less frequently). Travis-CI is a hosted service (*i.e.*, it is supported by a remote build server) integrated with GitHub (*i.e.*, pull requests can be tested automatically by Travis-CI, and the GitHub pull request UI is updated automatically with test results), therefore all projects using it operate in the same context. The entire build history of projects using Travis-CI is available through the Travis-CI API. Jenkins is a self-hosted system, *i.e.*, projects using it set up their $\mathcal{CI}$ service locally. Typically, only data on recent builds is stored by Jenkins. To reduce potentially confounding effects based on variations in how Jenkins is implemented by each project, and in order to have access to complete build histories (to be able to discern effects associated with *adoption* of $\mathcal{CI}$), we restricted our attention in this study to the projects using Travis-CI.

We used the available Travis-CI API to detect whether a given project used TRAVIS-CI or not [49]. We found 918 projects, 48.7% of 1,884, that used Travis-CI. For each project, we collected data about closed pull requests (*i.e.*, ignoring pull requests that are still open) from GHTorrent. The data we collected included metadata (*e.g.*, number of comments on the pull request), as well as the title, descrip-

[3] http://redmonk.com/dberkholz/2014/05/02/github-language-trends-and-the-fragmenting-landscape/
[4] http://githut.info/

Table 1: Programming language statistics for the selected 246 GitHub projects. Approximately half of all pull requests closed (column *pull_reqs*) arrived after projects adopted Travis-CI (*pull_reqs_after_CI*); most of these underwent CI testing (*pull_reqs_CI_tested*).

Language	n	*pull_reqs*	*pull_reqs after_CI*	*pull_reqs CI_tested*
JavaScript	65	45,781	23,010	22,453
Python	60	46,787	24,510	23,916
Ruby	36	26,559	15,058	14,634
PHP	24	22,907	10,227	9,836
Java	22	9,682	5,021	4,892
C++	16	16,543	7,464	7,284
C	16	11,097	6,158	6,028
Scala	7	2,700	1,509	1,466
Total	246	182,056	92,957	90,509

tion, body, and the actual code contents, collected separately from the GitHub API; we also collected Travis-CI data (*e.g.*, the timestamps and outcomes of each build).

A further step of filtering was required to select projects where the use of Travis-CI had endured long enough to reach some kind of steady state. Some pull-request-based projects had used $\mathcal{CI}$ since the outset, while others had only been using it a few months at the time of our observation; furthermore, some projects made inconsistent use of $\mathcal{CI}$ on their pull requests, *i.e.*, despite adopting Travis-CI, they only used it sporadically (on few of their incoming pull requests).[5] We therefore selected projects that had a good level of activity *after* the adoption of $\mathcal{CI}$. Specifically, we selected projects where between 25% and 75% of the pull requests were received after the adoption of CI (to ensure sufficient history both before and after $\mathcal{CI}$), *and* 88.4% of the pull requests received after the adoption of $\mathcal{CI}$ were actually tested using $\mathcal{CI}$. The 88.4% threshold was chosen to cover 75% of the projects that were plausibly in a steady-state use of $\mathcal{CI}$. The final dataset consisted of 246 projects (65 JavaScript, 60 Python, 36 Ruby, 24 PHP, 22 Java, 16 C++, 16 C, and 7 Scala), balanced with respect to use of $\mathcal{CI}$: 51.1% (92,957 out of 182,056) of pull requests were submitted after the adoption of Travis-CI (computed as the date of the earliest pull request tested by $\mathcal{CI}$), and 48.9% before, as shown in Table 1.

3.1.2 Collecting Data on Source and Test Files

Starting with the creation date of each project, we collected 3-month snapshots[6] of their source code repository (by obtaining the identifying commit hashes—SHA values—for the most recent commit in each snapshot, and performing subsequent `git reset` commands). For each snapshot, we identified source and test files using standard filename extension conventions on GitHub [18, 54]. Finally, we used *CLOC*[7] to calculate the number of files and the number of executable lines of code.

[5]Travis-CI does not attempt to build pull requests having the `[ci skip]` flag anywhere in the commit message.

[6]We resorted to this approximation due to the otherwise much greater computational effort required to reset each project's repository to its state at the time of each incoming pull request, before computing the metrics.

[7]`http://cloc.sourceforge.net/`

Table 2: Summary statistics on the various metrics related to productivity, for the selected 246 GitHub projects, for a period of 24 months centered around adoption of Travis-CI (4,148 rows of monthly project data; outliers removed as described in Section 3.2).

Statistic	Mean	St. Dev.	Min	Median	Max
proj_age	23.84	14.16	1	22	78
n_src_loc	33,675.45	54,727.58	1	12,842.5	454,899
n_test_loc	9,120.58	15,867.94	0	3,380.5	113,977
n_stars	784.15	1,128.05	0	314	5,453
n_forks	141.68	175.13	0	80	1,126
ci_use	0.52	0.50	0	1	1
team_size	3.44	2.39	1	3	12
n_pr_open	19.63	21.31	0	12	125
n_pr_merged	15.96	18.96	0	9	120
n_pr_rejected	3.68	5.54	0	2	83
n_pr_core	10.35	17.17	0	3	121
n_core_merged	9.32	15.77	0	2	119
n_core_rejected	1.04	3.05	0	0	62
n_issues_open	11.97	15.59	0	7	95

3.1.3 Collecting Productivity Data

To reason about *productivity*, generally understood as the amount of output per unit input [14] (*e.g.*, work per unit time), we focused on integrator productivity, *i.e.*, the number of pull requests merged per month. We computed various related metrics at monthly intervals for a period of 24 months, centered on the adoption date of Travis-CI (*i.e.*, 12 months prior, and 12 months after adoption of $\mathcal{CI}$; we ignored the actual adoption month). We recorded: the project age (**proj_age**; in months); the project size (**n_src_loc**, the number of source lines of code, in source files; **n_test_loc**, the number of source lines of code, in test files);[8] the number of forks (**n_forks**) and the number of stars (**n_stars**) of the project's main repository, as measures of popularity (cumulative count since the project's creation until the current month); whether the project uses $\mathcal{CI}$ at this time (**ci_use**, binary);[9] the team size (**team_size**, the number of core developers active each month);[10] the number of pull requests received and the number of issues opened during the current month, as measures of activity (**n_pr_open**, **n_issues_open**); the number of merged (**n_pr_merged**) and rejected (**n_pr_rejected**) pull requests; we further distinguished between pull requests submitted by core developers and those submitted by external contributors. The data is summarized in Table 2.

3.1.4 Collecting Quality Data

We operationalizes *code quality* by the number of bugs per unit time, a commonly used measure [25]. There are two main ways to identify the incidence of bugs when mining software repositories. First, given that it is common practice for developers to describe their work in commit messages, one way to reason about the rate of bugs is to identify *bug-*

[8]**n_src_files**, the number of source files, and **n_test_files**, the number of test files, yielded qualitatively similar results, therefore we exclude them from presentation.

[9]We also recorded a project's $\mathcal{CI}$ age—**ci_age**—in months, ranging from −12 to +12, but did not find its effects to be statistically significant.

[10]We identified core developers as those developers who either had write access to a project's code repository, or had closed issues and pull requests submitted by others.

Table 3: Summary statistics on the various metrics related to quality, for the 42 GitHub projects we found to use issue tagging consistently, for a period of 24 months centered around adoption of Travis-CI (562 rows of monthly project data; outliers removed as described in Section 3.2).

Statistic	Mean	St. Dev.	Min	Median	Max
proj_age	16.12	9.70	1	14.5	45
n_src_loc	48,712.02	47,770.60	117	40,912	196,436
n_test_loc	7,969.43	9,689.29	0	3,419	43,406
n_stars	165.32	254.38	0	43	1,105
n_forks	41.66	55.82	0	19	294
ci_use	0.55	0.50	0	1	1
n_pr_open	572.05	407.49	210	447	2,783
n_issues_open	20.49	18.42	1	15	96
n_bug_issues	5.96	7.52	0	3	34
n_core_bugs	5.02	6.81	0	2	33
n_user_bugs	0.94	2.04	0	0	13

fix commits, *e.g.*, by searching for defect-related keywords (*error*, *bug*, *fix*, etc.) in commit messages [31]. However, this approach may have low accuracy, even when augmented with a topic-based prediction model [13]. Second, in some projects developers assign labels to issues reported in the project's issue trackers (*e.g.*, *bug*, *feature request*) for easier coordination and management [44], which may enable a more accurate identification of bugs. However, on GitHub and its integrated issue tracker, the use of tagging is not enforced and, consequently, only few projects tag their issue reports [6]. In the current study we adopted a conservative stance, and traded scale for accuracy; relying on heuristics for identifying bug-fix commits would have allowed us to perform the analysis on more projects (larger scale), but would have arguably been less accurate; instead, we chose to investigate a smaller sample of GitHub projects, *i.e.*, those that used issue tagging consistently, in trying to maximize data quality (accuracy).

From our list of candidate projects that adopted Travis-CI, we selected those that used the GitHub issue tracker (at least 100 issues reported) and tagged their issues (at least 75% of issues have tags), for a total of 42 projects.[11] For each project we separated *bug* issues (indicative of quality) from other issue types (*e.g.*, discussions, feature requests), as follows. Since tagging is project specific, we started by manually reviewing how project managers used tags to label bugs in some highly active GitHub projects (*e.g.*, `rails`, `scipy`), and compiled a list of bug-related keywords, *i.e.*, *defect*, *error*, *bug*, *issue*, *mistake*, *incorrect*, *fault*, and *flaw*, after lowercasing and Porter stemming. We then searched for these stems in issue tags in all projects, and labeled the issue as *bug* if any of its tags (potentially multiple) contained at least one stem.

We further distinguished between *core developer bugs*, *i.e.*, those reported internally, by core developers, and *user bugs*, *i.e.*, those reported externally, by users. As discussed in Section 2, adoption of $\mathcal{CI}$ is expected to shorten the feedback loop, and allow developers to discover bugs quicker. At the same time, the effects of this should be positive on external software quality, *i.e.*, users should not experience an increasing number of defects. The distinction between internally-reported and externally-reported bugs should allow us to more clearly observe effects associated with adoption of $\mathcal{CI}$ on external quality. Note that we again adopted a conservative stance, and labeled as core developer bugs those issues reported by people who eventually became core developers for that project, even if they were not yet core developers at the time of reporting the issue. This ensured that *user bugs* are in fact bugs reported by people that have never been directly affiliated with the project (*i.e.*, part of its core team), at least by the end date of our data collection efforts.

Then, similarly as before, we computed monthly project-level data for a period of 24 months centered around the adoption date of Travis-CI, for a total of 562 rows of data summarized in Table 3. In addition to measures described above (Table 2), we recorded the number of bugs reported during the current month (**n_bug_issues**), broken down by reporter (**n_core_bugs**; **n_user_bugs**).

3.2 Analysis

To answer each of our research questions, we used multiple regression modeling to describe the relationship between a set of explanatory variables (predictors, *e.g.*, usage of $\mathcal{CI}$) and a response (outcome, *e.g.*, number of bugs reported per unit time), while controlling for known covariates that might influence the response.

Our data is challenging in two ways: (i) most of our predictors and all our response variables are counts (*e.g.*, of bugs reported, pull requests merged, developers, etc.); some outcome variables are over-dispersed, *i.e.*, the variance is much larger than the mean; and (ii) some response variables present an excess number of zeros (*e.g.*, when modeling the number of bugs reported by external developers in a given project per month, as a measure of quality, there is an over-abundance of months during which no bug issues had been submitted, *i.e.*, an excess number of zero values compared to non-zero values). To deal with the former, we used negative binomial (NB) regression, a class of generalized linear models used to model non-negative integer responses, suitable for modeling over-dispersed count data [1].

In the latter case, fitting a single model on the whole data would assume that both the zero and non-zero values come from the same distribution, which may not be the case (*e.g.*, zeros might be due to core developers not being active during those months, or to them being busy with other activities). We dealt with this issue by using zero-inflated models [26], *i.e.*, mixture models capable of dealing with excess zero counts, by combining a count component (we use NB) and a point mass at zero (we use a logit model to model which of the two processes the zero outcome is associated with). In R [37], zero-inflated models are implemented in the `pscl` package [23]. To ensure that using zero-inflated models was necessary (having many zeros doesn't necessarily mean that they were generated by different processes, *i.e.*, that they come from different distributions), we compared the fit of the zero-inflated model to that of a conventional NB model using Vuong's test of non-nested model fit [50].

Where appropriate, we also log transformed predictors to stabilize the variance and improve model fit [7]. To avoid multicollinearity between explanatory variables, we considered the VIF (variance inflation factor) of the set of predictors, comparing against the recommended maximum of 5 [7] (in our case all remained well below 3, indicating absence of multicollinearity).

[11] 43 projects met these criteria, but one was additionally removed because we did not find any of its test files.

In regression analysis, few data points may sometimes have disproportionate effects on the slope of the regression equation, artificially inflating the model's fit. This is a common occurrence when explanatory variables have highly skewed distributions, as most of ours do (*e.g.*, very few projects have an extremely large number of forks). Whenever one of our variables x was well fitted by an exponential distribution, we identified, and rejected as outliers, values that exceeded $k(1 + 2/n)median(x) + \theta$ [34], where θ is the exponential parameter [39], and k is computed such that not more than 3% of values are labeled as outliers. This reduced slightly the size of the data sets onto which we built the regression models, but ensured that our models are robust against outliers.

4. RESULTS AND DISCUSSION

4.1 Team Productivity (RQ1)

The first question we address pertains to team-level (*i.e.*, project-level) productivity, understood here as the ability of core team members (integrators) to handle pull requests effectively and efficiently. As discussed in Section 2, a desirable pull-based development workflow should be both *effective*, in that pull requests are eventually accepted (*i.e.*, merged into the codebase), and *efficient*, in that a large volume of incoming pull requests can be handled quickly by core developers, without compromising quality [17].

To address this question, we model the *number of pull requests merged per month* as a response against explanatory variables that measure test coverage (measured as a count of the *number of source lines of code in test files*; **n_test_loc**) and *usage of CI* (**ci_use**, binary variable that encodes whether or not the current project month falls after the date of adoption of Travis-CI), while controlling for various confounds, which we recall below (see also the discussion in Section 3.1.3).

team_size is the number of developers on the project's core team, at that time; larger teams are expected to be able to handle a higher volume of pull requests.

n_forks, the number of project forks at that time; we use this as a proxy measure for the size of the contributor base; contributors create forks, and contributions in the form of pull requests arise from forks.

n_src_loc, the number of source lines of code in source code files at that time, after excluding documentation files, test files, etc; larger projects are expected to receive, and merge, more pull requests.

proj_age, the number of months since the project's creation; growth dynamics may be different in younger vs. older projects.

n_stars, the number of stars received by the project's main repository by that time, used as a proxy for project popularity. Starring is one of the GitHub-specific social media features, that allows anyone to flag, *i.e.*, star, projects they find interesting; more popular projects can be expected to receive more pull requests.

Modeling Considerations.

Our primary measure of project performance is the number of pull requests received and processed by the project's core team. There are two possible outcomes of each pull-request: *merged* and *rejected*. These are two distinct process outcomes; it is quite reasonable to expect that the processes

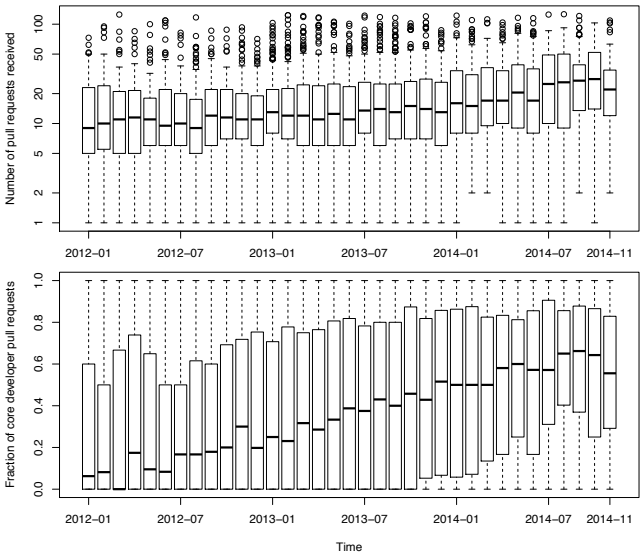

Figure 2: *Top*: Number of pull requests received monthly per project, excluding outliers (log y-axis). *Bottom*: Fraction of core developer pull requests.

that lead to them, and their determinants, are different, *viz.*, drawn from different samples. Therefore, we create separate models for *merged* (accepted) and *rejected* pull requests.

In addition, we expect that pull request processing would function differently for insiders and outsiders. Prior work [19, 28,45,51] suggests that the strength of the social relationship between submitter and evaluators plays an important role in contribution acceptance, and that pull requests submitted by core developers are treated preferentially. With time, an increasingly larger fraction of the pull requests received each month by projects are submitted by core developers (Figure 2). We therefore split the data into two groups by pull request submitter (*core developers* versus non-core developers, or *external contributors*), and present separate sets of models for each group.

Results and Discussion.

Table 4 presents the zero-inflated negative binomial (ZINB) core developer models, for merged (left) and rejected (right) pull requests. Similarly, Table 5 presents the zero-inflated negative binomial (ZINB) external contributor (*i.e.*, non core developer) models, for merged (left) and rejected (right) pull requests. In both cases, all models (*Merged* and *Rejected*) fit the data significantly better than the corresponding null models (*i.e.*, the intercept-only models), as evident from chi-squared tests on the difference of log likelihoods. Additionally, both models represent improvements over a standard NB regression, as confirmed by Vuong tests (ZINB, NB) in each case.

Each model consists of two parts, a count part (the columns "Count"), and a zero-inflation part (the columns "Zero-infl"). In the count part, we present the negative binomial regression coefficients for each of the variables, along with their standard errors. In the inflation part, similarly, we present the logit coefficients for predicting excess zeros, along with their standard errors. The statistical significance of coefficients is indicated by stars.

Table 4: Zero-inflated **core developer** pull request models. The response is the number of core developer pull requests merged (left) and rejected (right) per month.

	Merged PRs		*Rejected PRs*	
	Count	Zero-infl	Count	Zero-infl
(Intercept)	0.625***	−0.834	−0.945***	2.886***
	(0.134)	(0.498)	(0.225)	(0.620)
team_size	0.212***	−1.259***	0.163***	−1.398***
	(0.009)	(0.095)	(0.015)	(0.165)
proj_age	−0.010***	0.026***	0.012***	0.032***
	(0.002)	(0.005)	(0.003)	(0.007)
log(n_stars+0.5)	−0.031	0.375***	0.038	0.101
	(0.016)	(0.059)	(0.028)	(0.069)
log(n_forks+0.5)	−0.155***	−0.028	−0.099*	−0.070
	(0.021)	(0.066)	(0.039)	(0.091)
log(n_src_loc+0.5)	0.130***	0.073	0.008	−0.030
	(0.015)	(0.046)	(0.026)	(0.067)
log(n_test_loc+0.5)	0.059***	−0.022	0.085***	−0.012
	(0.009)	(0.024)	(0.017)	(0.046)
ci_useTRUE	0.187***	−0.894***	−0.353***	−0.714***
	(0.049)	(0.144)	(0.076)	(0.215)
Log(theta)	−0.124***		−0.595***	
	(0.037)		(0.064)	
AIC	22079.594	22079.594	9249.581	9249.581
Log Likelihood	−11022.797	−11022.797	−4607.790	−4607.790
Num. obs.	4148	4148	4148	4148

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

Table 5: Zero-inflated **external contributor** pull request models. The response is the number of external contributor pull requests merged (left) and rejected (right) per month.

	Merged PRs		*Rejected PRs*	
	Count	Zero-infl	Count	Zero-infl
(Intercept)	0.355**	−1.109	−0.791***	−2.101**
	(0.129)	(1.048)	(0.142)	(0.801)
team_size	0.015	0.074	0.053***	−0.000
	(0.008)	(0.049)	(0.009)	(0.038)
proj_age	−0.008***	0.010	−0.007***	0.004
	(0.002)	(0.013)	(0.002)	(0.010)
log(n_stars+0.5)	0.010	−0.208**	0.003	−0.307***
	(0.013)	(0.065)	(0.014)	(0.047)
log(n_forks+0.5)	0.096***	−0.841***	0.221***	−0.655***
	(0.018)	(0.128)	(0.020)	(0.076)
log(n_src_loc+0.5)	0.172***	−0.095	0.069***	0.196**
	(0.013)	(0.111)	(0.014)	(0.073)
log(n_test_loc+0.5)	−0.060***	0.177*	0.044***	0.210**
	(0.007)	(0.082)	(0.008)	(0.071)
ci_useTRUE	0.085	1.335**	−0.232***	0.569*
	(0.045)	(0.471)	(0.050)	(0.247)
Log(theta)	−0.224***		−0.052	
	(0.031)		(0.046)	
AIC	23914.789	23914.789	16824.317	16824.317
Log Likelihood	−11940.394	−11940.394	−8395.158	−8395.158
Num. obs.	4148	4148	4148	4148

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

We discuss the effects of each explanatory variable across all models. The coefficient for ***team size*** is statistically significant in the core developer models (Table 4). As expected, we can see a positive relationship between team size and number of pull requests merged: the expected increase in the response for a one-unit increase in team size is 1.236 ($e^{0.212}$), holding other variables constant. Stated differently, adding one member enables the team to merge 23.6% more core developer pull requests. Team size also has a significant positive relationship with the number of pull requests rejected, *i.e.*, larger teams also tend to reject more pull requests. Taken together, the two results confirm the expected effect of team size: larger teams are able to process more pull requests, irrespective of whether these will be eventually merged or rejected. Interestingly, this is not the case in the external contributor models (Table 5), where the coefficient for team size is not significant in the *Merged* part, *i.e.*, larger teams do not also merge more external contributions; in fact, they reject more. In the core developer models (Table 4), team size is also significant, and has a negative effect, in the zero inflation models. The log odds of being an excessive zero for merged pull requests would decrease by 1.26 (1.4 for rejected pull requests) for every additional team member. Overall, the larger the team, the less likely that the zero would be due to core developers not submitting pull requests. Put plainly, the larger the team, the more likely that core developers would submit pull requests, and also the more likely that these would be merged.

In the core developer models (Table 4), ***project age*** has a significant negative effect on merged pull requests, and a significant positive effect on rejected ones. Older projects accept fewer / reject more pull requests from core members. The expected decrease in the number of pull requests merged for a one-month increase in project age holding other variables constant is, however, very small ($\sim 1\%$). In both zero inflation parts, the relationship is positive and significant, suggesting also that it is more likely that core developers would not submit pull requests in older projects. This can be explained, perhaps, by individual core developer activity generally diminishing with time. In the external pull request models (Table 5), project age has a significant negative effect on both merged and rejected pull requests. The older the projects, the fewer external pull requests they merge, and also the fewer they reject. Stated differently, the older the projects, the fewer external pull requests they receive.

Project popularity (n_stars) has a significant effect *only* in the merged inflation part in the core developer models (Table 4). The log odds of being an excessive zero would increase in more popular projects. There is no significant effect of popularity on rejected PRs, in either count or zero-inflation; nor on the number of merged PRs. Only the number of periods with zero merged pull-requests increases. This suggests that the more popular the project, the less the core developers are likely to submit pull requests. This can be explained by the presence of larger pools of external contributors in these projects. Indeed, we can see in the inflation parts of the external contributor pull request models (Table 5) that external contributors are more likely to submit pull requests in more popular projects (the coefficients are negative and significant in the inflation parts of both the *Merged* and the *Rejected* models).

The negative effect of ***number of forks*** in the count part in both core developer models (Table 4) can be explained by the relatively small size of a project's core team compared to the size of its community of external contributors, who create forks of the project in order to submit pull requests. The more forks a project has, the fewer pull requests are submitted by core developers. Indeed, we can see very clearly in the external contributor models (Table 5) that the more forks a project has, the more pull requests are submitted by non-core developers.

Project size (n_src_loc) has an expected positive effect on the number of core developer pull requests being merged (Table 4). Naturally, the project size grows as a result of core

developers integrating their contributions. In the *Rejected* model, project size is not significant. In contrast, project size has a significant positive effect on both merged and rejected pull requests by external contributors (Table 5), *i.e.*, larger projects simply receive more external contributions. However, the disparate significance of project size in the *Merged* and *Rejected* models confirms that core developer pull requests are accepted preferentially.

Next, we turn our attention to the main explanatory variables, $\mathcal{CI}$ use and test coverage. Naturally, the effectiveness of $\mathcal{CI}$ depends on the availability and scale of existing test suites. In the core developer models (Table 4), the ***size of test files*** (n_test_loc) has a significant positive effect on the pull request count in both the *Merged* and the *Rejected* models. That is, having more tests allows team members to process more core developer pull requests, but there is no differential effect on their acceptance and rejection due to testing alone. Interestingly, the external contributor models (Table 5) show that having more tests is associated with fewer pull requests by non-core members being accepted, and more being rejected. This suggests that the availability and coverage of tests enables team members to find more defects in code submitted by external contributors, which results in fewer of their pull requests being accepted. However, these effects are mediated by the presence of $\mathcal{CI}$, as we discuss next.

$\mathcal{CI}$ use has a significant positive effect on the number of merged pull requests, and a significant negative effect on the number of rejected pull requests, in the core developer models (Table 4). Holding other variables constant, the expected increase in the number of merged core developer pull requests in projects that use $\mathcal{CI}$ (*i.e.*, a one-unit increase in **ci_use**, from FALSE, encoded as 0, to TRUE, encoded as 1) is 20.5%. Similarly, the expected decrease in the number of rejected pull requests in projects that use $\mathcal{CI}$ is 42.3%. In the external contributor models (Table 5), $\mathcal{CI}$ use has a negative effect on the count of pull requests rejected, with a sizable 26% effect. This suggests that external contributors may also benefit from the availability of $\mathcal{CI}$: by being able to receive immediate feedback on whether their contributions can be merged (*i.e.*, pass integration testing), they can more easily improve their code if necessary (*i.e.*, if tests fail) and update the pull request, resulting in overall fewer of their pull requests being rejected.

Result 1: *Teams using $\mathcal{CI}$ are significantly more effective at merging pull requests submitted by core members. Availability of $\mathcal{CI}$ is also associated with external contributors having fewer pull requests rejected.*

4.2 Code Quality (RQ2)

To address this question, we model the *number of bug reports* (*i.e.*, issues clearly labeled as bugs, as discussed in Section 3.1.4) raised in a project each month, as a response against explanatory variables that measure test coverage (measured as a count of the *number of source lines of code in test files*; **n_test_loc**) and *usage of $\mathcal{CI}$* (**ci_use**, binary variable that encodes whether or not the current project month falls after adoption of Travis-CI). Similarly as in the previous research question, we control for various confounds: the size of of the project's contributor base (*number of project forks*; **n_forks**), the size of the project (*number of source lines of code in source code files*; **n_src_loc**), the *project's age* (**proj_age**), and the project's popularity (*number of stars*; **n_stars**). In addition, we control for the *number of non-bug issue reports* received that month (**n_non_bug_issues**), as a measure of activity or general interest in the project.

We model separately the counts of bugs reported by core developers, and those reported by external contributors, since we expect adoption of $\mathcal{CI}$ may impact these sub-populations differently. For example, core developers may report bugs regularly as part of their development process, while external contributors may be more inclined to report bugs when they experience defects while using the software.

Table 6 presents the zero-inflated negative binomial (ZINB) models for bugs reported by core developers (left, denoted *Core Dev. Bugs*) and external contributors (right, denoted *External Bugs*). Both models provide a significantly better fit for the data than the corresponding null models (*i.e.*, the intercept-only models), as evident from chi-squared tests on the difference of log likelihoods. Additionally, both models represent improvements over a standard NB regression, as confirmed by Vuong tests (ZINB, NB) in each case.

We start by discussing the effects associated with our controls. As expected, the ***number of non-bug issue reports*** has a significant and positive effect on the response, in both models. Projects with an increased activity on the issue tracker are more likely to receive more bug reports, both from external contributors as well as from core developers.

Project age has a significant negative effect on the count of bugs reported by core developers. The older the project, the fewer bug reports it receives from core developers. Similarly, the ***project's popularity*** has a significant negative effect on the count of bugs reported by core developers, *i.e.*, the more popular the project, the fewer bug reports it receives from internal developers. These two results indicate an increasing reliance of the core team on external contributors, with time, and as the project becomes more popular and has access to a larger pool of external contributors or, perhaps, a shifting focus in the core team, from fixing bugs to implementing new features. The ***number of forks*** has a significant positive effect on the bug report count in both models, supporting the popularity result. The more forks a project has, the more contributors is has (both internal and external), and therefore the more bug reports it receives.

The ***size of test files*** is significant in the inflation part of the external bugs model, and has a negative effect. The log odds of being an excessive zero for external bug reports would decrease by 1.96 for every unit increase in the log of the number of test lines. In other words, the larger the test suite, the less likely that the zero would be due to external contributors not submitting bug reports, *i.e.*, the more likely that external contributors would report bugs. We attribute this association to the *post hoc* addition of test cases by developers: when a bug is reported by users, it is good practice to augment the repaired code with an automated test to ensure that this bug doesn't re-occur by progression. Since we cumulate our variables monthly, we observe an association between increased test cases, and the presence (or rather, non-absence) of user-reported bugs.

$\mathcal{CI}$ use has a significant positive effect on the count of bug reports by core developers. Holding other variables constant, the expected increase in the number of bug reports by core developers in projects that use $\mathcal{CI}$ is 48%. In contrast, $\mathcal{CI}$

Table 6: Zero-inflated project quality models. The response is the number of bugs reported per project each month by core developers (left) and external contributors (right).

	Core Dev. Bugs		*External Bugs*	
	Count	Zero-infl	Count	Zero-infl
(Intercept)	2.012***	3.185	0.001	1.385
	(0.397)	(1.975)	(0.673)	(2.115)
n_non_bug_issues	0.039***	−0.197*	0.033***	−0.364
	(0.005)	(0.078)	(0.008)	(0.242)
proj_age	−0.026**	−0.031	−0.002	0.049
	(0.008)	(0.075)	(0.010)	(0.045)
log(n_stars+0.5)	−0.122**	−0.161	0.006	−0.386
	(0.041)	(0.244)	(0.083)	(0.457)
log(n_forks+0.5)	0.155**	−0.311	0.332**	0.219
	(0.057)	(0.347)	(0.116)	(0.339)
log(n_src_loc+0.5)	−0.068	−0.084	−0.104	0.475
	(0.046)	(0.252)	(0.080)	(0.249)
log(n_test_loc+0.5)	−0.023	−0.211	−0.071	−0.674***
	(0.026)	(0.155)	(0.069)	(0.197)
ci_useTRUE	0.392**	−0.899	0.164	−0.201
	(0.120)	(0.965)	(0.206)	(0.936)
Log(theta)	−0.035		−0.518***	
	(0.103)		(0.155)	
AIC	1309.820	1309.820	2827.910	2827.910
Log Likelihood	−637.910	−637.910	−1396.955	−1396.955
Num. obs.	562	562	562	562

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

use does not have any effect on the count of bug reports by external contributors. That is, all other things being equal, adoption of $\mathcal{CI}$ enables team members to discover more bugs, but this does not seem to have any influence on the project's external quality, as indicated by the count of bug reports by non-core developers. As observed earlier, the overall number of pull requests managed (both merged and rejected) increases after $\mathcal{CI}$; and this increased volume is being managed by the core developers without a corresponding increase in the number of user-reported bugs. This suggests that $\mathcal{CI}$ allows an increase in productivity (as per Result 1) without a significant negative effect on user-experienced quality.

Result 2: *Core developers in teams using $\mathcal{CI}$ are able to discover significantly more bugs than in teams not using $\mathcal{CI}$. This does not come at a cost to external software quality, as external contributors do not experience an increasing number of defects.*

5. CONCLUDING REMARKS

Process integration and automation have been an important topic of late, specially with the rise of DEVOPS. In this paper, we leverage the growth and diversity of the projects on GitHub to study the effect of one aspect of process integration and automation: the effect of introducing Continuous Integration to the pull request process. Our findings clearly point to the benefits of $\mathcal{CI}$: more pull requests get processed; more are being accepted and merged, and more are also being rejected. Moreover, this increased productivity doesn't appear to be gained at the expense of quality.

This must be considered a preliminary study. The exact mechanisms that allow developers to process more pull requests need to be better understood, perhaps *via* detailed case studies. However, our models yield results of quantitative significance, indicating that the findings are robustly manifested in our data.

Several threats should be noted. First, some of the relevant properties, such as popularity of projects with users, and developer interest in the project, are clearly confounds that affect our outcomes. More user attention will certainly lead to more bugs; and more (non-core) developer interest will increase project productivity. However, both are difficult to measure directly, so we measure both indirectly. User attention is measured using the stars awarded in GitHub (**n_stars**); we assume that more stars are a good proxy for more user interest. Likewise, we use the number of forks (**n_forks**) as a proxy for developer interest. While these measures are intuitively justifiable, the use of such indirect measures is a potential internal validity threat.

Second, one might like to take the experimental posture that $\mathcal{CI}$ introduction is an independent, causal factor, and seek to identify the effects thereof. However, some projects may have introduced $\mathcal{CI}$ *because* they were experiencing high interest; also, they may have introduced $\mathcal{CI}$ because they already had a strong quality culture. It's also possible that the introduction of $\mathcal{CI}$ causes people to behave differently, for example to maintain reputations in the face of more rigorous testing. Our study cannot distinguish between these various effects; all we can study with our modeling is the quality and productivity effects *associated* with the introduction of $\mathcal{CI}$. Perhaps in the end, whatever be the modality of the effects, our findings, that $\mathcal{CI}$ use appears to be associated with productivity gains, without significantly sacrificing quality, is *per se* good enough to support its use.

Third, the data we gathered comes from a relatively small number of projects, compared to the size of GitHub. One of the reasons was that we set on the Travis-CI system; while others are available and in use in GitHub, we wanted to make sure the comparison was fair and even across projects. The amount of data we did get guarantees sufficient power for our models and results; however it is always possible that our sample was biased in some unknown way, thus diminishing the generalizability of our results; independent replication remains the best way to mitigate this threat.

6. ACKNOWLEDGEMENTS

BV, PD, and VF are partially supported by NSF under grants 1247280 and 1414172. YY and HW gratefully acknowledge support from the National Science Foundation of China (grants 61432020 and 61472430).

7. REFERENCES

[1] P. D. Allison and R. P. Waterman. Fixed–effects negative binomial regression models. *Sociological Methodology*, 32(1):247–265, 2002.

[2] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and isolated development with branches. In *FASE*, pages 316–331. Springer, 2012.

[3] A. Begel, J. Bosch, and M.-A. Storey. Social networking meets software development: Perspectives from GitHub, MSDN, Stack Exchange, and TopCoder. *IEEE Software*, 30(1):52–66, 2013.

[4] A. Bhattacharya. Impact of continuous integration on software quality and productivity. Master's thesis, Ohio State University, 2014.

[5] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open borders? Immigration in open source projects. In *MSR*, pages 6–6. IEEE, 2007.

[6] T. F. Bissyande, D. Lo, L. Jiang, L. Reveillere, J. Klein, and Y. Le Traon. Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub. In *ISSRE*, pages 188–197. IEEE, 2013.

[7] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013.

[8] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in GitHub: Transparency and collaboration in an open software repository. In *CSCW*, pages 1277–1286. ACM, 2012.

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

[10] B. J. Dempsey, D. Weiss, P. Jones, and J. Greenberg. Who is an open source software developer? *CACM*, 45(2):67–72, 2002.

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

[12] M. Fowler. Continuous integration, 2006. `http://martinfowler.com/articles/continuousIntegration.html`.

[13] Y. Fu, M. Yan, X. Zhang, L. Xu, D. Yang, and J. D. Kymer. Automated classification of software change messages by semi-supervised Latent Dirichlet Allocation. *Information and Software Technology*, 57:369–377, 2015.

[14] R. Gadagkar. Evolution of eusociality: the advantage of assured fitness returns. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 329(1252):17–25, 1990.

[15] M. Gharehyazie, D. Posnett, B. Vasilescu, and V. Filkov. Developer initiation and social interactions in OSS: A case study of the Apache Software Foundation. *Emp. Softw. Eng.*, pages 1–36, 2014.

[16] G. Gousios. The GHTorrent dataset and tool suite. In *MSR*, pages 233–236. IEEE, 2013.

[17] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *ICSE*, pages 345–355. ACM, 2014.

[18] G. Gousios and A. Zaidman. A dataset for pull-based development research. In *MSR*, pages 368–371. ACM, 2014.

[19] G. Gousios, A. Zaidman, M.-A. Storey, and A. Van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *ICSE*. IEEE, 2015.

[20] A. Hars and S. Ou. Working for free? Motivations of participating in open source projects. In *HICSS*, pages 1–9. IEEE, 2001.

[21] J. Holck and N. Jørgensen. Continuous integration and quality assurance: A case study of two open source projects. *Australasian Journal of Information Systems*, 11(1), 2007.

[22] J. Humble and D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[23] S. Jackman. pscl: Classes and methods for R developed in the Political Science Computational Laboratory, 2008.

[24] K. Karhu, T. Repo, O. Taipale, and K. Smolander. Empirical observations on software testing automation. In *ICST*, pages 201–209. IEEE, 2009.

[25] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do faster releases improve software quality? An empirical case study of Mozilla Firefox. In *MSR*, pages 179–188. IEEE, 2012.

[26] D. Lambert. Zero-inflated Poisson regression, with an application to defects in manufacturing. *Technometrics*, 34(1):1–14, 1992.

[27] J. Marlow and L. Dabbish. Activity traces and signals in software developer recruitment and hiring. In *CSCW*, pages 145–156. ACM, 2013.

[28] J. Marlow, L. Dabbish, and J. Herbsleb. Impression formation in online peer production: activity traces and personal profiles in GitHub. In *CSCW*, pages 117–128. ACM, 2013.

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

[30] A. Miller. A hundred days of continuous integration. In *Agile*, pages 289–293. IEEE, 2008.

[31] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *ICSME*, pages 120–130. IEEE, 2000.

[32] L. Osterweil. Software processes are software too. In *ICSE*, 1987.

[33] D. Parsons, H. Ryu, and R. Lal. The impact of methods and techniques on outcomes from agile software development projects. In *Organizational Dynamics of Technology-Based Innovation: Diversifying the Research Agenda*, pages 235–249. Springer, 2007.

[34] J. K. Patel, C. Kapadia, and D. B. Owen. *Handbook of statistical distributions*. M. Dekker, 1976.

[35] R. Pham, L. Singer, O. Liskin, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *ICSE*, pages 112–121. IEEE, 2013.

[36] B. Phifer. Next-generation process integration: CMMI and ITIL do devops. *Cutter IT Journal*, 24(8):28, 2011.

[37] R Development Core Team. R: A language and environment for statistical computing. *R Foundation for Statistical Computing, Vienna, Austria*, 2008.

[38] B. Rady and R. Coffin. *Continuous Testing: with Ruby, Rails, and JavaScript*. Pragmatic Bookshelf, 2011.

[39] P. J. Rousseeuw and C. Croux. Alternatives to the median absolute deviation. *Journal of the American Statistical Association*, 88(424):1273–1283, 1993.

[40] J. Sheoran, K. Blincoe, E. Kalliamvakou, D. Damian, and J. Ell. Understanding watchers on GitHub. In *MSR*, pages 336–339. ACM, 2014.

[41] D. Ståhl and J. Bosch. Experienced benefits of continuous integration in industry software product development: A case study. In *The 12th IASTED International Conference on Software Engineering, (Innsbruck, Austria, 2013)*, pages 736–743, 2013.

[42] D. Ståhl and J. Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87:48–59, 2014.

[43] S. Stolberg. Enabling agile testing through continuous integration. In *Agile Conference, 2009. AGILE'09.*, pages 369–374. IEEE, 2009.

[44] M.-A. Storey, C. Treude, A. van Deursen, and L.-T. Cheng. The impact of social media on software engineering practices and tools. In *FSE/SDP Workshop on Future of Software Engineering Research*, pages 359–364. ACM, 2010.

[45] J. Tsay, L. Dabbish, and J. Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *ICSE*, pages 356–366. ACM, 2014.

[46] J. Tsay, L. Dabbish, and J. Herbsleb. Let's talk about it: Evaluating contributions through discussion in GitHub. In *FSE*, pages 144–154. ACM, 2014.

[47] B. Vasilescu, V. Filkov, and A. Serebrenik. Perceptions of diversity on GitHub: A user survey. In *CHASE*, 2015.

[48] B. Vasilescu, D. Posnett, B. Ray, M. G. J. van den Brand, A. Serebrenik, P. Devanbu, and V. Filkov. Gender and tenure diversity in GitHub teams. In *CHI*, pages 3789–3798. ACM, 2015.

[49] 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 *ICSME*, pages 401–405. IEEE, 2014.

[50] Q. H. Vuong. Likelihood ratio tests for model selection and non-nested hypotheses. *Econometrica*, pages 307–333, 1989.

[51] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu. Wait for it: Determinants of pull request evaluation latency on GitHub. In *MSR*. IEEE, 2015.

[52] Y. Yu, H. Wang, G. Yin, and C. Ling. Reviewer recommender of pull requests in GitHub. In *ICSME*, pages 609–612. IEEE, 2014.

[53] Y. Yu, H. Wang, G. Yin, and C. Ling. Who should review this pull request: Reviewer recommendation to expedite crowd collaboration. In *APSEC*, pages 335–342. IEEE, 2014.

[54] J. Zhu, M. Zhou, and A. Mockus. Patterns of folder use and project popularity: A case study of GitHub repositories. In *ESEM*, pages 30:1–30:4. ACM, 2014.