"Doing" Agile versus "Being" Agile

Empirical Results from 250+ Projects

Akond Rahman, Amritanshu Agrawal, Rahul Krishna, Alexander Sobran*, and Tim Menzies
North Carolina State University, IBM Corporation*
[aarahman,aagrawa8,rkrish11]@ncsu.edu,asobran@us.ibm.com,tim@menzies.us

ABSTRACT

In numerous occasions Agile practitioners have warned about the negative aspects of adopting Agile tools and techniques, without implementing the primary practices of Agile. They have coined this observation as *"doing" Agile, but not "being" Agile.* However such warnings are opinion-based, as Agile practitioners have provided little to no empirical evidence that supports their recommendations. We mine 150 open source software (OSS) and 123 proprietary projects to investigate if empirical evidence exists for the phenomenon: "doing" Agile, but not "being" Agile. In particular, we investigate if the Agile technique of continuous integration (CI) influences bug and issue resolution, as well as commit patterns. According to our empirical analysis, for OSS projects, we observe the expected benefits after CI adoption, i.e., more bugs are resolved, and more issues are resolved. However, for the proprietary projects, we cannot make similar observations. Furthermore, we observe proprietary projects to "do" Agile, but not "be" Agile, as these projects use CI tools, without implementing the primary Agile practices for example, making frequent commits. We recommend practitioners not to use Agile techniques such as CI, without adopting the necessary practices.

CCS CONCEPTS

• **Software and its engineering → Agile software development**;

KEYWORDS

Continuous Integration, DevOps, GitHub

ACM Reference format:
Akond Rahman, Amritanshu Agrawal, Rahul Krishna, Alexander Sobran*, and Tim Menzies. 2018. "Doing" Agile versus "Being" Agile. In *Proceedings of International Conference on Mining Software Repositories, Gothenburg, Sweden, May 2018 (MSR'18),* 11 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In numerous blog posts [31] [15] [23] [40], Agile practitioners caution that some organizations pretend to "do Agile" without actually "being Agile"; which implies the organization's use of Agile techniques and tools without actually adopting the necessary Agile practices. While their cautions seem sensible, the Agile practitioners do not provide any empirical evidence which demonstrates how well-founded these concerns are. Accordingly, this paper checks empirically for two observations:

- Are some projects "doing" Agile without "being" Agile?
- Do we observe any detrimental effects for projects that are "doing" Agile without "being" Agile?

To investigate these two observations, we investigate the usage of continuous integration (CI), one of the primary techniques to implement agility for software projects [2]. CI systems integrate code changes by automatically compiling, building, and executing test cases upon submission of code changes. In recent years, usage of CI tools has become increasingly popular both for open source software (OSS) projects [4] [19] [43] as well as for proprietary projects [42]. Influence and adoption of CI has also gained interest amongst researchers. Hilton et al. [19] have investigated adoption rate of CI in OSS Github projects. Other researchers have investigated if CI influences commit patterns [47], and issue resolution [47] for OSS Github projects. But, how does CI influence bug and issue resolution for proprietary projects? This paper compares the influence of adopting CI between OSS and proprietary projects by mining data from 150 OSS projects and 123 proprietary projects. We obtain this collection of 273 projects from a cloud-programming development environment hosted by our industrial partner.

Our goal is to quantify the influences of CI on bug resolution, collaboration, commit patterns, and issue resolution for both OSS and proprietary projects. Similar to prior work [43] [47], we focus on the influences of CI on bug resolution, issue resolution, and commit patterns. We also focus on collaboration, as use of CI, being one of the primary Extreme Programming (XP) techniques, is expected to benefit collaboration amongst team members [41]. In this paper we answer the following research questions:

RQ1: How does adoption of continuous integration influence issue resolution? Significantly more issues are resolved after adoption of CI for OSS projects, but not for proprietary projects.
RQ2: How does adoption of continuous integration influence bug resolution? Unlike proprietary projects, significantly more bugs are resolved after adoption of CI within OSS projects.
RQ3: How does adoption of continuous integration influence collaboration amongst team members? After adopting CI, collaboration significantly increases for both, OSS and proprietary projects. The increase in collaboration is more observable for OSS projects than proprietary projects.
RQ4: Does adoption of continuous integration influence commit patterns? Commit frequency and sizes significantly increases for OSS projects after CI adoption but not for proprietary projects.

Hence, based on our empirical findings, we advise:

> *Do not use the tools from another community without also adopting the practices of that community.*

MSR'18, May 2018, Gothenburg, Sweden
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

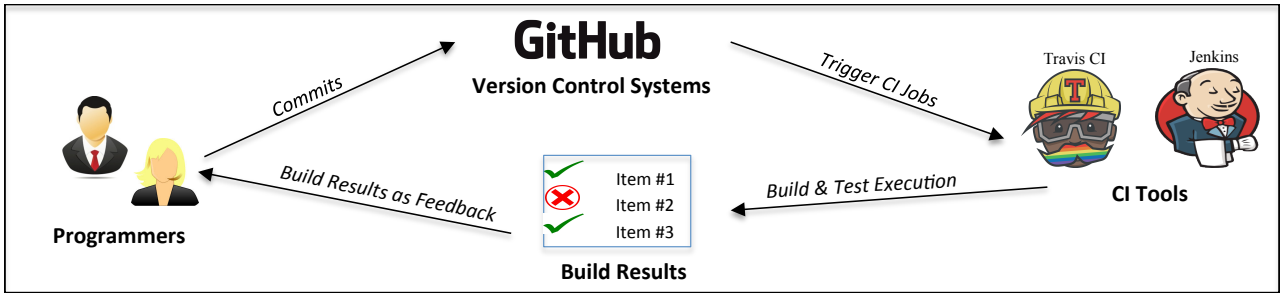


Figure 1: An example work-flow of the continuous integration (CI) process.

The rest of this paper is organized as follows. The next section discusses background and related work. Our data analysis methodology is explained in Section 3 which is followed by our findings in Section 4. These findings are discussed in Section 5, followed by notes on the validity of our conclusions in Section 6.

Before all that, it is important to say that there may well be benefits to CI which are not captured by our metrics (for e.g., cultural benefits in introducing CI tools to an existing infrastructure). Regardless, our main point remains: it is most unwise to hype CI usage promising that it will (for example) instantly increase bug resolution. Findings from our paper can dampen overoptimistic expectations, thus enabling better discussions between management and developers about CI (and other Agile methods and tools).

2 BACKGROUND

We first briefly describe the concept of "doing" Agile vs. "being" Agile. Next, we provide a brief background on CI and prior research work related to CI.

2.1 "Doing" Agile vs "Being" Agile

In numerous blog posts, practitioners have discussed the subtle differences between "doing" Agile and "being" Agile. "Doing" Agile implies a team's adoption of Agile tools and techniques such as daily scrums and CI, whereas, "being" Agile refers to the proper adoption of the agile practices such as frequently releasing software to end-users [31]. As a hypothetical example, a team may adopt a CI tool as part of their daily work-flow, but if they release software infrequently, then that would imply that the team is "doing" Agile, and not "being" Agile. Some agile enthusiasts provides the following rule of thumb: "doing" Agile leads to 20% added benefits, whereas, "being" Agile leads to 200% added benefits [15]. One of the co-authors of the Agile Manifesto [3] stresses the importance of "being" Agile and stated "stop doing Agile and start being Agile" [17]. However, "being" Agile is non-trivial, as suggested by an Agile practitioner who stated "doing Agile can be achieved over night, being Agile is a journey." [23].

In popular programming literature we observe practitioners to discuss about "doing" vs. "being" Agile in details. However, we find little empirical evidence that supports/extends, the observation of "doing" Agile vs. "being" Agile. We use 273 projects to investigate the observation of "doing" Agile vs. "being" Agile by quantifying the influences of CI on bug/issue resolution, collaboration, and commit patterns. To the best of our knowledge, this is largest such study of it's kind.

2.2 About Continuous Integration (CI)

In XP, CI is identified as one of the primary practices to implement agility for software projects [2]. Humble and Farley [21] attributes the introduction of CI to Beck [2]. According to Duvall et al. [14], CI originated from the imperatives of agility, in order to respond to customer requests quickly. When building the source code, CI tools can execute unit and integration tests to ensure quality of the integrated source code. If the tests do not pass, CI tools can be customized to give feedback to team members. Even though the concept of CI was introduced in 2006, initial usage of CI was not popular amongst practitioners [12]. However, since 2011, with the advent of CI tools such as Travis CI [10], usage of CI has increased in recent years [19]. CI is also considered to be a pre-requisite to implement continuous deployment [24].

When a software team adopts CI, the team has to follow a set of practices [14]. According to the CI methodology all programmers has to check-in their code daily, which are integrated daily [14]. Unlike, traditional methodologies such as waterfall, in CI, programmers get instant feedback on their code via build results. To implement CI, the team must maintain its source code in a version control system (VCS), and integrate the VCS with the CI tool so that builds are triggered upon submission of each commit [14]. Figure 1 provides an example on how a typical CI process works. Programmer make commits in a repository maintained by a VCS such as, GitHub, and these commits trigger CI jobs on a CI tool such as Travis CI which executes, builds, tests, and produces build results. These build results are provided to the programmers as a feedback either through e-mails, or phone alerts [14] on their submitted code changes. Based on the build results, programmers make necessary changes to their code, and repeats the CI process again.

The above-mentioned description implies the practice of making frequent commits playing a pivotal role on how programmers are using CI for feedback. But do programmers change their commit patterns upon adoption of CI? We answer this question by mining 273 OSS and proprietary projects.

2.3 Related Work

Our paper is closely related to prior research that have investigated the adoption and usage of CI tools. We briefly describe this prior work as following:

2.3.1 ***Adoption.*** Hilton et al. [19] mined OSS projects hosted on Github. They observed that most popular projects use CI, and reported that the median time of CI adoption is one year. They also advocated for wide-spread adoption of CI, as CI correlates with several positive outcomes. However, adoption of CI is non-trivial as suggested by other prior work; for example, Olsson et al. [29] who identified lack of automated testing frameworks as a key barrier to transition from a traditional software process to a CI-based software process. Also, Hilton et al. [18] surveyed industrial practitioners and identified three trade-offs to adopt CI: assurance, flexibility, and security. Rahman et al. [32] observed that adoption of CI is not wide-spread amongst practitioners. They investigated which diffusion of innovation (DOI) factors influence adoption of CI tools, and reported four factors: relative advantages, compatibility, complexity, and education.

The above-mentioned studies indicate that adoption of CI involves overcoming certain barriers, and adopting certain practices. Findings from these papers also highlight the importance of assessing if after adoption of CI, do software teams enjoy benefits such as increased collaboration, better bug resolution, and better issue resolution?

2.3.2 ***Usage.*** Beller et al. [4] mined Java and Ruby-based projects from Github, and synthesized the nature of build and testing attributes exhibited amongst OSS projects that use CI. Zolfagharinia et al. [49] mined 30 million builds of the CPAN ecosystem's CI environment, and characterized the evolution of build systems. Similarly, Rausch et al. [34] investigated build failures from 14 Java-based systems and characterized 14 error categories. They also observed that process metrics have relatively more influence on CI build failures. Zampetti et al. [44] investigated the use of static analysis tools in a CI-based workflow, and reported that the main reason builds fail is due to not maintaining coding standards. Vasilescu et al. [43] mined GitHub projects that use Travis CI, and reported that adoption of CI increases productivity for OSS projects. Zhao et al. [47] mined OSS GitHub projects, and investigated if software development practices such as commit frequency, commit size, and pull request handling, changes after adoption of CI.

The above-mentioned findings highlight the community's interest in how CI is being used in software projects. From the above-mentioned prior work, we can also list the following as exemplars of the expected benefits of adopting CI:

- Zhao et al. [47] reported that for OSS GitHub projects, after the adoption of CI tools:
  - The number of closed issues increases.
  - The frequency and size of commits increases.
- Vasilescu et al. [43] reported that for OSS GitHub projects, number of bugs do not increase after adoption of CI.

Note that all of these findings are derived from OSS projects. With respect to the development process, structure, and complexity, proprietary projects are different from OSS projects [30] [37], which motivates us to pursue our research study. Hence, for the rest of this paper, we will compare the influence of adopting CI within OSS and proprietary projects. We consider four attributes of software development: bug resolution, collaboration amongst team members, commit patterns, and issue resolution.

3 METHODOLOGY

As stated before, we take motivation from prior work [19] [47] [43] that have studied CI usage on OSS projects, and answer four research questions. We first describe how we filter the collected projects, then we answer the four research questions. We follow the Goal-Question-Metric (GQM) approach [9] to conduct our empirical analysis. The goal of our paper is to quantify the influences of CI on bug resolution, collaboration, and issue resolution, for both OSS and proprietary projects. To achieve this goal we answer four research questions as described in Sections 3.2, 3.3, 3.4, and 3.5. Each of the research questions are answered by metrics, as described in their respective section. We select the GQM approach as this approach has been successfully used to conduct empirical studies in software engineering [7] [48].

3.1 Filtering

To perform our experiments we use OSS projects from GitHub, and proprietary projects collected from our industrial partner. In case of OSS projects we select public GitHub projects that are included as a 'GitHub showcase project'. Of the publicly available projects hosted on GitHub, a selected set of projects are marked as 'showcases', to demonstrate how a project can be developed in certain domain such as game development, and music [16]. Our assumption is that by selecting these GitHub projects we can start with a representative set of OSS projects that enjoy popularity, and provide good examples of software development. Example of popular projects included in the GitHub showcase that we use for our analysis are: Javascript libraries such as 'AngularJS'[1], and programming languages such as 'Scala'[2].

In case of proprietary projects our industrial partner provided us a list of projects that are hosted on private GitHub. We download OSS and proprietary projects, respectively, by using the public GitHub API, and a private API maintained by our industrial partner.

Projects hosted on GitHub which gives researchers the opportunity to extract necessary project information such as commits, and issues [22] [8] [28]. Unfortunately, these projects can contain short development activity, can be used for personal use, and not be related to software development at all [22] [8]. Hence, we need to create a set of projects that can contain sufficient software development data for analysis. We apply a filtering strategy that can be described in the following manner:

- **Filter-1:General** As the first step of filtering, we identify projects that contain sufficient software development information using the following criteria. These criteria address the limitations of mining GitHub projects as stated by prior researchers [22] [8].
  - *Collaboration*: Number of pulls requests are indicative of collaboration, and the project must have at least one pull request.
  - *Commits*: The project must contain more than 20 commits.
  - *Duration:*The project must contain software development activity of at least 50 weeks.
  - *Issues*: The project must contain more than 10 issues.
  - *Contributors:*The project must not be used and maintained by one person. The project must have at least eight contributors.
  - *Releases:*The project must have at least one release.

[1]https://github.com/angular/angular.js
[2]https://github.com/scala/scala

- *Software Development*:The project must only be a placeholder for software development source code.
- **Filter-2:CI** We use the second filter to identify projects that have adopted CI tools.
  - *CI Tool Usage*: The project must use any one of the following tools: Circle CI, Jenkins, and Travis CI. We select these tools as these tools are frequently used in GitHub projects [19]. We determine if a project is using Circle CI, Jenkins, and Travis CI by inspecting the existence of 'circle.yml', 'jenkins.yml', and 'travis.yml', respectively, in the root directory of the project.
  - *Availability of Data (Before and After Adoption of CI)*: The project must have at least one month of software development activity data available before and after adoption of CI. We exclude projects that have less than one month of software development activity data before adoption of CI, or after adoption of CI, or both. Our assumption is that availability of at least one month of software development data, both before and after adoption of CI, can be sufficient to conduct analysis.
  - *Start Date*: The project must start on or after January, 2014. From our initial exploration we observe that 90% of the proprietary projects start on or after 2014. Our assumption is that by selecting the year of 2014, we can obtain software projects that are comparable in terms of count.

We use the GitHub API to extract necessary information from these projects and test each criteria stated above. Upon completion of Filter-1 we obtain a set of projects that contain sufficient software development activity for analysis. Upon completion of 'Filter-2:CI', we obtain a list of projects from which we extract metrics for both: OSS and proprietary projects. The raw data for the OSS projects is available online [3].

3.2 RQ1: How does adoption of continuous integration influence issue resolution?

We investigate if CI influences issue resolution in RQ1. Resolution of issues is important to practitioners, as resolution of issues leads to new feature development, and increased productivity [43]. Issues in GitHub correspond to features, tasks that need be completed, or enhancements of existing features. RQ1 focuses on if CI has an influence on how many issues are closed. Closed issues correspond to issues that are resolved by programmers, and indicates the team's productivity. We answer RQ1, by computing the normalized proportion of issues that are closed ('Normalized Proportion of Closed Issues' *NCI*). We compute *NCI* by normalizing 'Proportion of Closed Issues' (*CLI*) respectively, with time. We perform normalization to account for the variability in number of months before and after adoption of CI, from one project to another. We compute *CLI* using Equation 1.

$$CLI(p, m) = \frac{\text{total count of closed issues in month } m\text{, for project } p}{\text{total count of issues in month } m\text{, for project } p} \tag{1}$$

$$NCI(p) = \frac{\sum_{i=1}^{M} CLI(p, i)}{M} \tag{2}$$

[3]https://figshare.com/s/9340ebc7ffa90c4ad82b

To calculate *NCI* for both, before an after adoption of CI, we use the same Equation 2. In Equation 2, M presents the total count of months before or after adoption of CI for project p. For example, the number of months before and after adoption of CI is respectively, five and six then, we use Equation 2 with $M = 5$ to calculate the project's *NCI* before adoption of CI, and with $M = 6$, to calculate the project's *NCI* after adoption of CI. We compute *NCI*, for all projects, before and after adoption of CI. We apply statistical tests to determine if CI has an influence on issue resolution. We describe the statistical tests in Section 3.6.

3.3 RQ2: How does adoption of continuous integration influence bug resolution?

Bugs in software is always a concern for practitioners, as bugs in deployed software can lead to serious economic toll. If CI influences resolution of bugs then practitioners might consider in CI tool adoption and continuing usage. We quantify the influence of CI on bug resolution in RQ2. In Gihtub, issues can be tagged as 'bug', which indicates the issue is related to fixing a bug. However, issues may not be marked as a 'bug', yet related to bug resolution. We account for this limitation by adopting a keyword-based strategy similar to prior work [35] [46]. We selected the following keywords: 'bug', 'fix', 'issue', 'error', 'correct', 'proper', 'deprecat', 'broke', 'optimize', 'patch', 'solve', 'slow', 'obsolete', 'vulnerab', 'debug', 'perf', 'memory', 'minor', 'better', 'complex', 'break', 'investigat', 'compile', 'defect', 'inconsist', 'crash', 'problem', and 'resol'. We consider an issue to be marked as a 'bug', if any of the above-mentioned keywords appear at least once as a stemmed token in the issue comments, issue title, or issue description. Next, we compute bugs that are closed and normalized by time ('Normalized Proportion of Closed Bugs' *NCB*). We compute *NCB* by normalizing 'Proportion of Closed Bugs' (*CB*) with time. We compute *CB* and *NCB* respectively using Equation 3 and Equation 4.

$$CB(p, m) = \frac{\text{total count of closed bugs in month } m\text{, for project } p}{\text{total count of bugs in month } m\text{, for project } p} \tag{3}$$

$$NCB(p) = \frac{\sum_{i=1}^{M} CB(p, i)}{M} \tag{4}$$

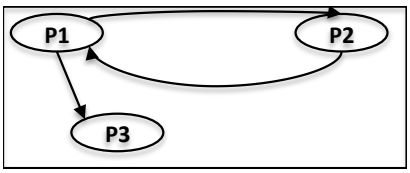
Similar to issues as stated in Section 3.2, we compute *NCB*, for all projects, before and after adoption of CI. We apply statistical tests, described in Section 3.6, to determine if CI has an influence on bug resolution.

3.4 RQ3: How does adoption of continuous integration influence collaboration amongst team members?

Practitioners perceive CI to increase visibility and transparency in the software development process [14]. Furthermore, as CI is one of the primary XP practices, teams that have adopted XP practices such as CI, are expected to be highly collaborative [41]. If CI is properly adopted and executed, visibility and transparency within the software team will increase, and we expect to observe programmers to collaborate more with each other. For RQ3, we measure collaboration by using file modification history.

File (lines)	Author	Modifier
File1 (1-5)	P1	P1
File1 (10-13)	P1	P1
File1 (7-8)	P1	P2
File1 (51-60)	P1	P3
File2 (9-13)	P2	P1
File2 (21-29)	P2	P2
File2 (4-7)	P2	P2

a



b

Nodes	P1, P2, P3
Edges	(P1, P2), (P2, P1), (P1, P3)
In-degree	P1: 1 P2: 1 P3: 1

c

Figure 2: Hypothetical example on how we construct collaboration graphs. Figure 2a presents the list of modified files $File1$ and $File2$, the lines that were modified, and the programmers who modified the files. Figure 2b presents the resulting collaboration graph constructed from the modification history. Figure 2c presents nodes, edges, and in-degree of each nodes, for the constructed graph.

In prior work, researchers have mined software artifacts to investigate the nature of collaboration amongst programmers working on the same software project. They used emails [6], bug reports [45], file change history [27] [5], to construct collaboration graphs, and observed that social structures exist between programmers. We take inspiration from these prior research, and mine software artifacts to characterize collaboration before and after adoption of CI. Similar to Meenely et al. [27] and Bhattacharya et al. [5], we use the concept of file changes to construct a graph, In our approach, we construct collaborative graphs for each available month, where programmers are nodes, and an edge exist between two programmers, if one programmer modifies the same lines of a file, authored by another programmer. We use a hypothetical example, stated in Figure 2, to illustrate our approach.

In our hypothetical example, a project consists of two files $File1$ and $File2$. In Figure 2, we observe a list of programmers who are authoring and modifying two files. We construct a graph, using the modification information, as shown in Figure 2b. The constructed graph has three nodes (P1, P2, and P3), and three edges. From the constructed graph we extract in-degree. In-degree, corresponds to the count of edges that are incoming to a node. In our hypothetical example, the project's collaboration graph has three edges, and the in-degree for nodes P1, P2, and P3 is one. Therefore, the median in-degree for the collaboration graph is one. We measure collaboration using median in-degree because in-degree corresponds to collaboration between the programmers. The higher the median in-degree, the higher connection is between the nodes [25] [5], indicating more collaboration between the programmers.

Projects can vary in node count, and needs to be accounted for [20]. Similar to Hong et al. [20] and Bird et al. [6], we normalize median in-degree by the count of nodes. We use Equation 5, we normalize median in-degree.

$$\text{Median In-Degree (MID)} = \frac{\text{median in-degree}}{\text{total count of nodes}} \tag{5}$$

Finally, we aggregate the graph metrics for each project, and normalize with the respect to time. We use Equation 6, to calculate 'Normalized Median In-Degree'.

$$\text{Normalized Median In-Degree (NMID)} = \frac{\sum_{i=1}^{M} MID(i)}{M} \tag{6}$$

Similar to issues, as stated in Section 3.2, and bugs, as stated in Section 3.3, we compute $NMID$, for all projects, before and after adoption of CI. We apply statistical tests, described in Section 3.6, to determine if CI has an influence on collaboration.

3.5 RQ4: Does adoption of continuous integration influence commit patterns?

RQ4 focuses on investigating the differences in commit patterns before and after adoption of CI. Investigation of commit patterns can give us two insights: (i) understand the nature of programmer commit practices, and (ii) explain the influence of CI on issue and bug resolution. Practitioners perceive that frequency and size of commits is related to code quality and productivity, and by investigating the nature of commit patterns we can identify clues that explain the influence of CI on issue and bug resolution. We mine two features from commits, and normalize the identified features with respect to programmer count and time. These two features are commit count and commit size. We describe our process of mining these two features as following. To answer RQ4 we do not consider commits used to merge branches.

Commit count: First, we calculate the count of commits (CC) performed each month in a project using equation 7. Next we calculate the normalized commit count (NCC) using Equation 8.

$$CC(p, m) = \frac{\text{total count of commits in month } m\text{, for project } p}{\text{total count of active committers in month } m\text{, for project } p} \tag{7}$$

$$NCC(p) = \frac{\sum_{i=1}^{M} CC(p, i)}{M} \tag{8}$$

Commit size: We calculate commit size (CS) by calculating the total lines of code added and deleted per commit, within a month, as shown in Equation 9. Next, we calculate the normalized commit size of a project (NCS) using Equation 10.

Table 1: Projects filtered for each sanity check of Filter-1.

Sanity check	OSS	Proprietary
Commits > 20	96	68
Issues > 10	89	60
# Contributors >= 8	67	47
SW development only	51	9
Duration >= 50 weeks	46	12
Releases >0	44	136
Collaboration (Pull requests > 0)	54	35
Project count after filtering	661	171

$$CS(p, m) = \frac{\text{total lines added and deleted in month } m\text{, for project } p}{\text{total count of commits in month } m\text{, for project } p} \quad (9)$$

$$NCS(p) = \frac{\sum_{i=1}^{M} CS(p, i)}{M} \quad (10)$$

Similar to prior research questions, we compute *NCC*, and *NCS*, for all projects and for each month before and after adoption of CI. We apply statistical tests, described in Section 3.6, to determine if CI influences commit patterns.

3.6 Statistical Measurements

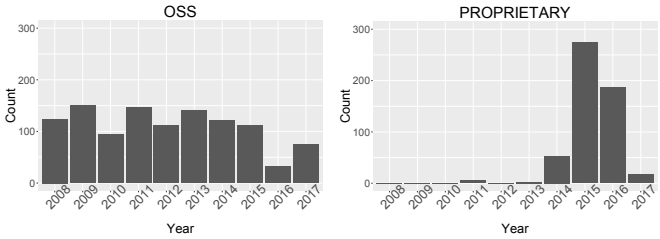
We use three statistical measures to compare the metrics of interest before and after adoption of CI: effect size using Cliffs Delta [11], the Mann Whitney U test [26], and the 'delta' measure. Both, Mann Whitney U test and Cliffs Delta are non-parametric. The Mann Whitney U test states if one distribution is significantly large/smaller than the other, whereas effect size using Cliffs Delta measures how large the difference is. Following convention, we report a distribution to be significantly larger than the other if $p - value < 0.05$. We use Romano et al.'s recommendations to interpret the observed Cliffs Delta values. According to Romano et al. [39], the difference between two groups is 'large' if Cliffs Delta is greater than 0.47. A Cliffs Delta value between 0.33 and 0.47 indicates a 'medium' difference. A Cliffs Delta value between 0.14 and 0.33 indicates a 'small' difference. Finally, a Cliffs Delta value less than 0.14 indicates a 'negligible' difference.

We also report 'delta (Δ)', which is the difference between the median values, before and after adoption of CI. The 'delta' measurement quantifies the proportion of increase or decrease, after and before adoption of CI. As a hypothetical example, for OSS projects, if median *NCI* is 10.0, and 8.5, respectively, after and before adoption of CI, then the 'delta' is +0.17 (= 10-8.5/8.5).

4 RESULTS

Before providing the answers to the research questions, we present summary statistics such as CI adoption time, release frequency, and size, of the studied projects. Initially we started with 1,108 OSS projects and 538 proprietary projects. The start year of these 1,108 OSS and 538 proprietary projects are presented in Figure 3. We report how many of the projects were filtered out by the first filter, Filter-1, in Table 1. The projects that are discarded when the steps given in Table 1 are applied sequentially, from top to bottom, we are left with 661 open-source and 171 proprietary projects.

From Table 1, we observe that 59.6% of the GitHub showcase projects pass the recommended sanity checks by researchers. The



a b

Figure 3: Count of projects that started in each year. Figure 3a presents the count of OSS projects. Figure 3b presents the count of proprietary projects.

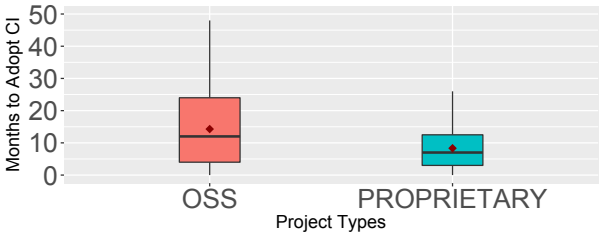


Figure 4: Time to adopt CI. The median time to adopt a CI technology is respectively 12.5, and 7 months for OSS and proprietary projects. The difference in the two distributions is significant ($p - value < 0.001$, Cliffs Delta=0.2).

Table 2: Projects filtered for each sanity check of Filter-2.

Sanity check	OSS	Proprietary
CI Tool Usage	448	46
Data Availability (at least one month)	0	0
Start Date (Must start on or after 2014)	63	2
Project count after filtering	150	123

447 projects filtered by applying Filter-1 further emphasizes the need to validate software project data mined from GitHub. From Figure 3 we observe that in 2014, majority of the proprietary projects started, and by selecting projects on or after the year 2014, we obtain a comparable set of OSS and proprietary projects.

As shown in Table 2, after applying Filter-2, we are finally left with 150 OSS and 123 proprietary projects. We use these projects to answer the four research questions. A brief summary of the filtered projects is presented in Table 3. From Table 3, we observe the differences between the OSS and the proprietary projects. The commit count per programmer is 24.2 and 46.7, respectively for OSS and proprietary projects. On average a programmer changes 141 and 345 files, respectively for OSS and proprietary projects.

The time to adopt CI, and release frequency for OSS and proprietary projects is also different as shown in Figure 4. We observe that the median time to adopt CI is 1.7 times longer for OSS projects, compared to that of proprietary projects. For OSS projects the median time to adopt CI is 12.5 months, which is consistent with prior research: Hilton et al. [19] reported that for GitHub projects, the median time to adopt CI is 12 months. From Figure 5, we observe average release per month to be significantly higher for OSS projects ($p - value < 0.001$, Cliffs Delta=0.9).

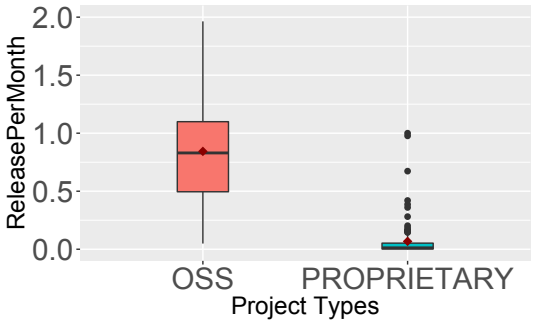


Figure 5: Average release per month for projects. The median average releases per month is respectively 1.0, and 0.01 for OSS and proprietary projects.

Table 3: Summary of Projects

Property	Project Type	
	OSS	Proprietary
Total Changed Files	1,122,352	728,733
Total Commits	191,804	98,542
Total LOC Added	48,424,888	44,003,385
Total LOC Deleted	30,225,543	26,614,230
Total Programmers	7,922	2,109
Total Projects	150	123

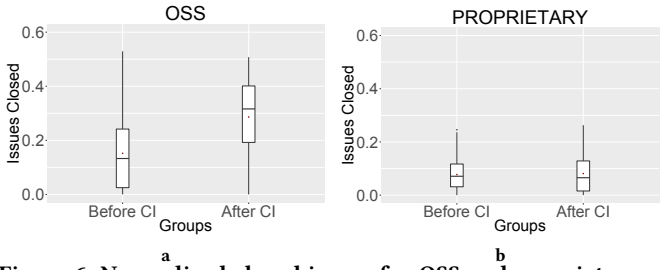


Figure 6: Normalized closed issues for OSS and proprietary projects. Figure 6a presents the normalized count of closed issues in OSS projects. Figure 6b presents the normalized count of closed issues in proprietary projects.

4.1 Answer to RQ1

Zhao et al. [47] reported that for OSS GitHub projects, number closed issues increases after adoption of CI. Our expectation is that for our set of OSS GitHub projects we will observe the same. In this section, we answer RQ1 by reporting the summary statistics of number of issues that are closed (*NCI*), before and after adoption of CI. In Figure 6, we report the *NCI* values for both OSS and proprietary projects. In Table 4 we report the results of the three statistical measures: the Mann Whitney U test, effect size, and the 'delta' measure. The 'delta' measure is presented in the Δ row. The 'delta' value for which we observe no significant difference is highlighted in grey. According to Table 4, for OSS projects, after adoption of CI, significantly more issues are closed ($p - value < 0.001$). On the contrary, for proprietary projects, the influence of CI is negligible on issue resolution. These findings are also evident form the box-plots presented in Figure 6. Furthermore, in OSS projects, considering

Table 4: Influence of CI on Closed Issues (*NCI*)

Measure	OSS	Prop.
Median	(A:0.31, B:0.13)	(A:0.06, B:0.7)
Δ	+1.38	-0.14
p-value	< 0.001	0.6
Effect size	0.5	0.0

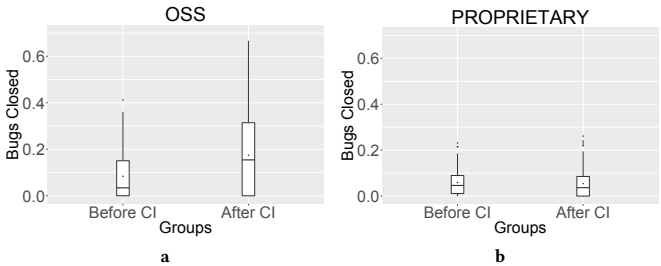


Figure 7: Normalized closed bugs for OSS and proprietary projects. Figure 7a presents the normalized count of closed bugs in OSS projects. Figure 7b presents the normalized count of closed bugs in proprietary projects.

median, the normalized count of closed issues, increases by a factor of 2.4, after adoption of CI, whereas, the normalized count of closed issues almost remains the same for proprietary projects. Our OSS-related findings are consistent with Zhao et al. [47].

> ***Lesson-1: For OSS projects, significantly more issues are resolved after adoption of CI. For Proprietary projects, adoption of CI has no influence on issue resolution.***

4.2 Answer to RQ2

Vasilescu et al. [43] reported that for OSS GitHub projects, code quality does not suffer after adoption of CI. For our set of OSS GitHub projects we expect to derive similar conclusions i.e. number of closed bugs will increase after adoption of CI. We answer RQ2 by reporting how many bugs are closed before and after adoption of CI. We report the normalized count of closed bugs (*NCB*) in Figure 7, for both OSS and proprietary projects. We report the results of the three statistical measures in Table 5. The 'delta' metric is represented in the Δ row. The 'delta' value for which we observe no significant difference is highlighted in grey.

According to Table 5, for OSS projects, after adoption of CI, significantly more bugs are closed. ($p - value < 0.001$). From Figure 7 we observe the median *NCB* to be 0.15 and 0.03, respectively for after and before adoption of CI. Hence, we can state that for OSS projects, bugs are closed five times more after adoption of CI. Similar to issue resolution, our OSS-related findings for bug resolution is somewhat consistent with prior research [43]. For proprietary projects, the influence of CI is non-significant for closed bug count. The median *NCB* values are almost similar, before and after adoption of CI, as observed from Figure 7 and Table 5.

> ***Lesson-2: Unlike proprietary projects, significantly more bugs are resolved in OSS projects after adoption of CI. For proprietary projects the influence of CI on bug resolution is non-significant.***

Table 5: Influence of CI on Closed Bug Count (*NCB*)

Measure	OSS	Prop.
Median	(A:0.15, B:0.03)	(A:0.03, B:0.04)
Δ	+4.0	-0.25
p-value	< 0.001	0.9
Effect size	0.3	0.1

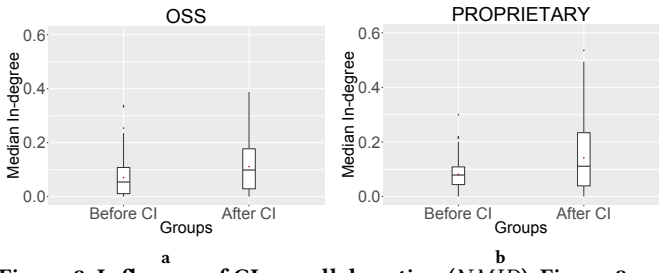


Figure 8: Influence of CI on collaboration (*NMID*). Figure 8a and Figure 8b respectively presents the median in-degree for OSS and proprietary projects.

Table 6: Influence of CI on Collaboration (*NMID*)

Measure	OSS	Prop.
Median	(A: 0.09, B:0.05)	(A: 0.11, B:0.07)
Δ	+0.8	+0.5
p-value	< 0.001	< 0.001
Effect size	0.2	0.2

4.3 Answer to RQ3

As described in Section 3.4, we report the normalized median in-degree for OSS and proprietary projects to answer RQ3. We report the summary statistics in Table 6, and the box-plots in Figure 8. For both OSS and proprietary projects, the median in-degree significantly increases after adoption of CI. The effect size for OSS and proprietary projects is 0.2, which is small according to Romano et al. [39]. Based on effect size, we observe that the influence of CI is not as large as closed issues or closed bugs for OSS projects, as reported in Section 4.1 and Section 4.2, respectively. Based on the 'delta' measure (Δ in Table 6) we observe that the increase in collaboration is not as high for proprietary projects, as it is for OSS projects. One possible explanation of this finding might be how CI is adopted in proprietary projects: the CI practices followed in the OSS domain might not be followed in the similar manner in the proprietary domain, which is leading to the differences in collaboration increase.

> ***Lesson-3*: After adoption of CI, collaboration between programmers significantly increases for OSS and proprietary projects. That said, increase in collaboration is larger for OSS projects, compared to proprietary projects.**

4.4 Answer to RQ4

In prior work, Zhao et al. [47] mined OSS GitHub projects, and reported that after adoption of CI, frequency and size of commits increases. We expect that our answers to RQ4 for OSS projects will be consistent with Zhao et al.'s [47] findings. We answer RQ4, by first reporting the frequency of commits before and after adoption of CI. Similar to RQ1, RQ2, and RQ3, we report the results of the three statistical measures in Table 7 and the box-plots in Figure 9. The 'delta' metric is represented in the Δ row. The 'delta' value

Table 7: Influence of CI on Commit Patterns.

	Commit Count (*NCC*)		**Commit Size** (*NCS*)	
Measure	OSS	Prop.	OSS	Prop.
Median	(A:2.2, B:0.9)	(A:0.7, B:1.1)	(A:25.2, B:10.5)	(A:14.6, B:23.8)
Δ	+1.44	-0.36	+1.40	-0.38
p-value	< 0.001	0.9	0.001	0.9
Effect size	0.3	0.1	0.2	0.1

for which we observe no significant difference is highlighted in grey. According to Table 7, after CI adoption programmers make significantly more commits in OSS projects, but not in proprietary projects.

The influence of CI on commit count is also visible from Figure 9. The median *NCC* is respectively, 2.2, and 0.9, after and before adoption of CI, which indicates that after adoption of CI, programmers make 2.4 times more commits than that of before adopting CI. Based on median *NCC*, the commit count for proprietary projects decreases from 1.1 to 0.7, after adoption of CI. Our findings indicate that for proprietary projects, programmers are not making frequent commits after adoption of CI. On the contrary for OSS projects programmers are making significantly more commits, confirming prior research findings [47] as well as practitioners' perceptions [14].

> ***Lesson-4*: After adoption of CI, commit frequency significantly increases for OSS projects, but not for proprietary projects. For proprietary projects we do not observe CI to have an influence on commit frequency.**

Commit size is another measure we use to answer RQ4. As shown in Table 7 we observe size of commits i.e., churned lines of code per commit to significantly increase for OSS projects, but not for proprietary projects. Even though the difference in commit size increases significantly, the effect size of the observed differences is smaller than that of commit counts. Our commit size-related findings are consistent with Zhao et al.'s [47] observations: they observed commit size to increase after adoption of CI. For proprietary projects our findings are similar to that of commit count. We do not observe significant differences in commit size before and after adoption of CI.

> ***Lesson-5*: For OSS projects, the size of commits significantly increase after adoption of CI. For proprietary projects, the difference in commit size before and after adoption of CI is non-significant.**

5 DISCUSSION

In this section, we discuss our findings with possible implications:

Commit Frequency: Standard practice in CI is to use a VCS (e.g., Git). When a programmer makes a commit, the CI tool fetches the code changes, triggers a build that includes inspection checks and/or tests [14]. If the build fails the CI tool provides rapid feedback on which code changes are not passing the inspection checks and/or test cases [14]. In this manner, the CI process provides rapid feedback about code changes to the programmer [14]. The programmer utilizes this feedback to fix the code changes by making more commits, fixing their code changes, eventually leading to more

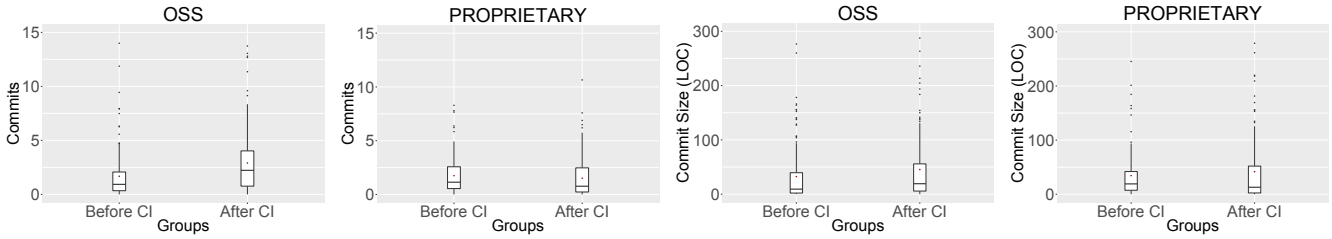


Figure 9: Normalized commit count and commit size for OSS and proprietary projects. Figure 9a presents the normalized count of commits in OSS projects. Figure 9b presents the normalized count of commits in proprietary projects. Figure 9c and Figure 9d respectively presents the normalized commit sizes for OSS and proprietary projects.

bug fixes and issue completions. Hence, by making more commits programmers might resolve more bugs and issues. Our explanation related to rapid feedback is congruent with Duvall et al. [14]; they stated "rapid feedback is at the heart of CI" and "without feedback, none of the other aspects of CI is useful".

On the contrary to OSS projects, after CI adoption, we have observed that in proprietary projects, change in commit frequency, number of closed bugs, and number of closed issues is non-significant. Based on above-mentioned explanation, we suggest that for the proprietary projects, programmers are not relying on CI for rapid feedback, and as a result, the commit frequency does not increase significantly, nor does the count of closed bugs and issues.

Based on the above-mentioned discussion, we make the following **conjecture**: ***practitioners might be benefited by seeking rapid feedback on submitted code changes from the CI process, by committing frequently***.

On the Differences Between OSS and Proprietary Projects As reported in prior work [30] [37], our empirical analysis also provides clues on the differences between OSS and proprietary projects, which we discuss below:

- On average, CI adoption rate is quicker for proprietary projects compared to OSS projects. One possible explanation can be related to co-adoption of CI and VCS i.e., when proprietary projects were transferred to Github, they also adopted CI, leading to an overall decreased adoption time. Another possible explanation can be related to CI's perceived benefits: teams might be biased to adopt CI tools because of it's perceived benefits.
- Findings from Table 1 suggest that proprietary projects are less likely to release frequently. Of the 538 projects, 25.2% of the projects do not pass the check 'Releases > 0'. On the other hand, the number is lower for OSS projects: 3.9% (44 of 1,108). The observation that OSS projects tend to release frequently is unsurprising: Raymond [36] in his 'Cathedral and the Bazaar' book mentioned the practice of frequent releases as "a critical part of the Linux development model". One possible explanation can be attributed to the nature of projects: these proprietary projects may be used internally within the organization, or the team, which are not delivered as software or services to the end-users. On the other hand, majority of the OSS projects are working software constantly changing to satisfy end-user needs.
- Release frequency for OSS projects is significantly higher than that of proprietary projects. As CI is a primary practice of XP, we expected to see frequent releases for both: OSS and proprietary projects. But as suggested by Figure 5, proprietary projects have a smaller release frequency than OSS projects.

"Doing" Agile vs "Being" Agile We have observed proprietary projects to make infrequent commits, and make infrequent releases. Both these practices are antitheses of two recommended Agile practices: making frequent code changes to get rapid feedback [14], and releasing working software frequently [3]. We refer to this observation as **"doing" Agile, but not "being" Agile**, for proprietary projects, because of the following observation: even though an Agile tool (CI) is used, the practices of Agile are not being applied. From Section 4 we observe the benefits of CI not to be applicable for proprietary projects, providing some empirical evidence to the claims of Agile enthusiasts: being Agile is more beneficial than doing Agile [15] [40]. Our findings also suggest that OSS projects are more likely to 'do' and also 'be' Agile.

Observed Benefits of CI, and 'Hero Projects' Based on the analysis of OSS GitHub projects, Hilton et al. [19] strongly advocated for CI adoption stating "developers should consider CI as a best practice and should use it widely as possible". We restrain ourselves from making similar recommendations for proprietary projects, as we do not observe CI to have influence on two attributes of software development for proprietary projects. As described in Section 4, we observe that CI to have no influence on bug and issue resolution for proprietary projects. Even for OSS projects, the influence of CI is more observable for issue resolution (larger effect size), than bug resolution. Our findings suggest that CI might not have the same influence on all software development attributes.

The fact that unlike OSS projects, the benefits of CI is not applicable for proprietary projects deserves explanation. One possible explanation can be derived from the 'hero' concept observed in proprietary projects by Agrawal et al. [1]. They identified projects where one or few programmers work in silos, and do 80% or more of the total programming effort as 'hero projects'. Agrawal et al. [1] reported the prevalence of hero projects amongst proprietary projects, which indicates that regardless of what tool/technique/methodology is being used, majority of the work will be conducted by a few programmers. In case of these projects, even if CI results in increased collaboration, the resolution of bug and issues will still be dependent on the programmers who are doing majority of the work i.e., 'hero' programmers. We conjecture, to some extent, for hero projects, the benefits of adopting Agile tools such as CI, is dependent on what

practices they are following, for e.g., making frequent commits, and obtaining feedback from peers for bug/issue resolution.

Changing Perceptions on CI Adoption: In software engineering, practitioners hold strong perceptions in certain topics, which are formed primarily from personal experiences and peer interactions [13]. Also, practitioners often follow the 'diffusion of innovation' rule, which states that practitioners prefer to learn from other practitioners who have already adopted the tool of interest [33] [38]. Our empirical study can be helpful to practitioners who hold certain perceptions about CI adoption. For example, by reading a success story of CI adoption for an OSS project, a practitioner might be convinced that CI adoption is a good choice for his/her team. The constructed perception can be checked and contrasted with empirical evidence. For CI adoption, learning from other practitioners can be a starting point, but practitioners also need to (i) consider their teams' development context, and (ii) systematically assess, to what extent other practitioners' experiences hold.

6 THREATS TO VALIDITY

We discuss the limitations of our paper as following:

Spurious Correlations: In any large scale empirical study where multiple factors are explored, some findings are susceptible to spurious correlations. To increase the odds that our findings do not suffer from such correlations we have:

- applied sanity checks to filter out irrelevant projects.
- applied normalization on our collected metrics.
- applied two tests: the effect size test and the Mann Whitney U test to perform statistically sound comparisons. Furthermore, for OSS projects, we compare and contrast our findings with prior research.
- discussed our findings with business users who are practitioners working for our industrial partner. The practitioners agreed with the general direction of findings: they stated that many teams within their company use a wide range of tools and techniques which does not work optimally for all teams. The practitioners also agreed that there are significant differences between OSS and proprietary software development, and we should not assume these tools and techniques will yield similar benefits.

Detection of CI Usage: We have adopted a heuristic-driven approach to detect use of CI in a project. We acknowledge that our heuristic is limited to the three CI tools, and we plan to improve our heuristics by exploring the possibility to add more CI tools.

Detection of Bugs: We have relied on issues for which a set of keywords appear to identify bugs. Even though keyword-based strategies have been used in software engineering research [35] [46], we acknowledge that a keyword-based strategy can be limiting. In future, we plan to apply manual analysis on the collected issues to determine which issues are actually related to defects.

Generalizability: We ask ourselves "Can the results from proprietary projects be generalized to other proprietary projects?". We acknowledge the proprietary projects come from one information technology (IT) organization. Whether or not our findings are generalizable for other IT organizations remain an open question. IT organizations can adopt different set of CI practices that are not captured by our analysis. We hope to address this limitation in future work.

Methodology Choices: Our approach is metrics-based, normalized by the time before and after adoption of CI, as well as by number of bugs and issues. Vasilescu et al. [43] used a 'count of bugs per unit time' with a negative binomial regression model, to quantify the influence of CI on bug resolution. Zhao et al. [47], used regression continuity design, to quantify the influence of CI on issue resolution. Despite our methodology differences, our findings for OSS projects are consistent with Vasilescu et al. [43] and Zhao et al. [47]'s findings. In future, we will explore other methodology choices to investigate the influence of CI on bug and issue resolution.

7 CONCLUSION

After mining 150 OSS and 123 proprietary projects, we have quantified the influences of CI on software development for OSS and proprietary projects. We have investigated the influence of CI on bug and issue resolution, collaboration, and programmers' commit patterns. We have observed that closed bugs, closed issues, and frequency of commits, significantly increase after adoption of CI for OSS projects, but not for proprietary projects. Furthermore, we observe evidence of 'doing' Agile, but not 'being' Agile in proprietary projects. Our findings suggest that if proprietary projects only 'do' Agile, without 'being' Agile, they may not fully reap the benefits of Agile tools and techniques, as warned by Agile practitioners [31] [15] [23]. We suggest practitioners not to use tools from another community, if the practices that are associated with those tools are not adopted.

While our findings can be biased by our sample of projects, to the best of our knowledge, there exists no large scale research study that reports the opposite of our conclusions. At the very least, our results raise the issue of the benefits of CI tools for proprietary projects–an issue that, we hope, will be addressed by other researchers in future research studies.

REFERENCES

[1] A. Agrawal, A. Rahman, R. Krishna, A. Sobran, and T. Menzies. 2018. We Don't Need Another Hero? The Impact of "Heroes" on Software Development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '18)*. To Appear, preprint: https://arxiv.org/abs/1710.09055.

[2] Kent Beck. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[3] Kent Beck, Mike Beedle, Arie Van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, et al. 2001. Manifesto for agile software development. (2001). http://agilemanifesto.org/.

[4] Moritz Beller, Georgios Gousios, and Andy Zaidman. 2017. Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, Piscataway, NJ, USA, 356–367. https://doi.org/10.1109/MSR.2017.62

[5] Pamela Bhattacharya, Marios Iliofotou, Iulian Neamtiu, and Michalis Faloutsos. 2012. Graph-based Analysis and Prediction for Software Evolution. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 419–429. http://dl.acm.org/citation.cfm?id=2337223.2337273

[6] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. 2006. Mining Email Social Networks. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*. ACM, New York, NY, USA, 137–143. https://doi.org/10.1145/1137983.1138016

[7] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. 2011. Don'T Touch My Code!: Examining the Effects of Ownership on Software Quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*

(ESEC/FSE '11). ACM, New York, NY, USA, 4–14. https://doi.org/10.1145/2025113.2025119

[8] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. 2009. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, 1–10.

[9] VRBG Caldiera and H Dieter Rombach. 1994. The goal question metric approach. *Encyclopedia of software engineering* 2, 1994 (1994), 528–532.

[10] Travis CI. 2017. Travis CI. https://travis-ci.org/. (2017). [Online; accessed 15-October-2017].

[11] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114, 3 (Nov. 1993), 494–509.

[12] Amit Deshpande and Dirk Riehle. 2008. *Continuous Integration in Open Source Software Development*. Springer US, Boston, MA, 273–280. https://doi.org/10.1007/978-0-387-09684-1_23

[13] Prem Devanbu, Thomas Zimmermann, and Christian Bird. 2016. Belief & Evidence in Empirical Software Engineering. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 108–119. https://doi.org/10.1145/2884781.2884812

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

[15] Jim Ewel. 2017. Doing Agile vs Being Agile. (2017). http://www.agilemarketing.net/agile-agile/ [Online; accessed 29-Jan-2018].

[16] Github. 2017. Github Showcases. https://github.com/showcases. (2017). [Online; accessed 13-October-2017].

[17] Jim Highsmith. 2011. Stop Doing Agile Start Being Agile. (2011). https://www.youtube.com/watch?v=cZsB8-7KI3o [Online; accessed 29-Jan-2018].

[18] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 197–207. https://doi.org/10.1145/3106237.3106270

[19] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 426–437.

[20] Qiaona Hong, Sunghun Kim, S. C. Cheung, and Christian Bird. 2011. Understanding a Developer Social Network and Its Evolution. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM '11)*. IEEE Computer Society, Washington, DC, USA, 323–332. https://doi.org/10.1109/ICSM.2011.6080799

[21] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional.

[22] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2014. The promises and perils of mining github. In *Proceedings of the 11th working conference on mining software repositories*. ACM, 92–101.

[23] Rash Khan. 2015. Being Agile vs Doing Agile, Why is it important to differentiate!? (2015). https://www.linkedin.com/pulse/being-agile-vs-doing-why-important-differentiate-rash-khan [Online; accessed 29-Jan-2018].

[24] M. Leppanen, S. Makinen, M. Pagels, V. P. Eloranta, J. Itkonen, M. V. Mantyla, and T. Mannisto. 2015. The highways and country roads to continuous deployment. *IEEE Software* 32, 2 (Mar 2015), 64–72. https://doi.org/10.1109/MS.2015.50

[25] Priya Mahadevan, Calvin Hubble, Dmitri Krioukov, Bradley Huffaker, and Amin Vahdat. 2007. Orbis: Rescaling Degree Correlations to Generate Annotated Internet Topologies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07)*. ACM, New York, NY, USA, 325–336. https://doi.org/10.1145/1282380.1282417

[26] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50–60. http://www.jstor.org/stable/2236101

[27] Andrew Meneely, Laurie Williams, Will Snipes, and Jason Osborne. 2008. Predicting Failures with Developer Networks and Social Network Analysis. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '08/FSE-16)*. ACM, New York, NY, USA, 13–23. https://doi.org/10.1145/1453101.1453106

[28] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. 2017. Curating GitHub for engineered software projects. *Empirical Software Engineering* (2017), 1–35. https://doi.org/10.1007/s10664-017-9512-6

[29] Helena Holmstrom Olsson, Hiva Alahyari, and Jan Bosch. 2012. Climbing the "Stairway to Heaven" – A Mulitiple-Case Study Exploring Barriers in the Transition from Agile Development Towards Continuous Deployment of Software. In *Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '12)*. IEEE Computer Society, Washington, DC, USA, 392–399. https://doi.org/10.1109/SEAA.2012.54

[30] James W. Paulson, Giancarlo Succi, and Armin Eberlein. 2004. An Empirical Study of Open-Source and Closed-Source Software Products. *IEEE Trans. Softw. Eng.* 30, 4 (April 2004), 246–256. https://doi.org/10.1109/TSE.2004.1274044

[31] Monica Quintana. 2017. Being Agile vs Doing Agile. (2017). https://atendesigngroup.com/blog/being-agile-vs-doing-agile [Online; accessed 29-Jan-2018].

[32] Akond Rahman, Asif Partho, David Meder, and Laurie Williams. 2017. Which Factors Influence Practitioners' Usage of Build Automation Tools?. In *Proceedings of the 3rd International Workshop on Rapid Continuous Software Engineering (RCoSE '17)*. IEEE Press, Piscataway, NJ, USA, 20–26. https://doi.org/10.1109/RCoSE.2017..8

[33] Akond Ashfaque Ur Rahman, Eric Helms, Laurie Williams, and Chris Parnin. 2015. Synthesizing Continuous Deployment Practices Used in Software Development. In *Proceedings of the 2015 Agile Conference (AGILE '15)*. IEEE Computer Society, Washington, DC, USA, 1–10. https://doi.org/10.1109/Agile.2015.12

[34] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 345–355. https://doi.org/10.1109/MSR.2017.54

[35] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 428–439. https://doi.org/10.1145/2884781.2884848

[36] Eric Raymond. 1999. The cathedral and the bazaar. *Philosophy & Technology* 12, 3 (1999), 23.

[37] Brian Robinson and Patrick Francis. 2010. Improving Industrial Adoption of Software Engineering Research: A Comparison of Open and Closed Source Software. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*. ACM, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/1852786.1852814

[38] Everett M Rogers. 2010. *Diffusion of innovations*. Simon and Schuster.

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

[40] Michael Sahota. 2016. Doing Agile vs. Being Agile. (2016). https://dzone.com/articles/doing-agile-vs-being-agile [Online; accessed 29-Jan-2018].

[41] Helen Sharp and Hugh Robinson. 2008. Collaboration and Co-ordination in Mature eXtreme Programming Teams. *Int. J. Hum.-Comput. Stud.* 66, 7 (July 2008), 506–518. https://doi.org/10.1016/j.ijhcs.2007.10.004

[42] Daniel Stahl, Torvald Martensson, and Jan Bosch. 2017. The continuity of continuous integration: Correlations and consequences. *Journal of Systems and Software* 127, Supplement C (2017), 150 – 167. https://doi.org/10.1016/j.jss.2017.02.003

[43] Bogdan Vasilescu, Yue Yu, Huaimin Wang, Premkumar Devanbu, and Vladimir Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 805–816.

[44] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. 2017. How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR '17)*. IEEE Press, Piscataway, NJ, USA, 334–344. https://doi.org/10.1109/MSR.2017.2

[45] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. 2013. Categorizing Bugs with Social Networks: A Case Study on Four Open Source Software Communities. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 1032–1041. http://dl.acm.org/citation.cfm?id=2486788.2486930

[46] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. 2016. Towards building a universal defect prediction model with rank transformed predictors. *Empirical Software Engineering* 21, 5 (01 Oct 2016), 2107–2145. https://doi.org/10.1007/s10664-015-9396-2

[47] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The Impact of Continuous Integration on Other Software Development Practices: A Large-Scale Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. ACM, New York, NY, USA.

[48] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering* 32, 4 (April 2006), 240–253. https://doi.org/10.1109/TSE.2006.38

[49] M. Zolfagharinia, B. Adams, and Y. G. Guehenuc. 2017. Do Not Trust Build Results at Face Value - An Empirical Study of 30 Million CPAN Builds. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 312–322. https://doi.org/10.1109/MSR.2017.7