ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/321025247

Continuous Integration: The Silver Bullet?

Article · November 2017

CITATIONS
0

READS
188

5 authors, including:

Akond Rahman
North Carolina State University
12 PUBLICATIONS **52** CITATIONS

SEE PROFILE

Amritanshu Agrawal
North Carolina State University
21 PUBLICATIONS **87** CITATIONS

SEE PROFILE

Rahul Krishna
North Carolina State University
28 PUBLICATIONS **90** CITATIONS

SEE PROFILE

Tim Menzies
North Carolina State University
562 PUBLICATIONS **8,464** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Planning in software engineering View project

Project Turning lead into gold. Hunting the Snark. Re-animating corpses. Sort of View project

All content following this page was uploaded by Amritanshu Agrawal on 14 November 2017.

The user has requested enhancement of the downloaded file.

Continuous Integration: The Silver Bullet?

On The Merits of Continuous Integration for Proprietary 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

arXiv:1711.03933v1 [cs.SE] 10 Nov 2017

ABSTRACT

Continuous integration (CI) tools integrate code changes by automatically compiling, building, and executing test cases upon submission of code changes. Use of CI tools is getting increasingly popular, yet how proprietary projects reap the benefits of CI remains unknown. To investigate the influence of CI on software development, we mine 661 open source software (OSS) projects, and 171 proprietary projects. 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. Therefore, we cannot claim that CI is the 'silver bullet' for software development.

Why is this so? Our findings indicate that only adoption of CI might not be enough to improve software development. CI can be effective for software development if practitioners use CI's feedback mechanism efficiently, by applying the practice of making frequent commits. For proprietary projects we observe practitioners to commit less frequently, and hence not use CI effectively, for obtaining feedback on the submitted code changes. We recommend practitioners to (i) apply the CI best practices along with adoption of CI tools, (ii) consider their team's development context before adopting CI tools, and (iii) after adoption of CI, investigate if CI satisfies their needs by applying software analytics.

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. 2017. Continuous Integration: The Silver Bullet?: On The Merits of Continuous Integration for Proprietary Projects. In *Proceedings of May,2018 (ICSE'18).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Is continuous integration (CI) useful? Adoption of CI has become increasingly common both for open source software (OSS) projects [13], as well as for proprietary projects [32]. Prior research [33] [35] [13] that have explored the influence of CI, have focused on OSS projects, and not for proprietary projects. Accordingly, in this paper, we ask "is CI beneficial for proprietary projects?".

ICSE'18, 2018
2016. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To answer this question, we mine 661 OSS projects with 72,847 programmers from GitHub, and 171 proprietary projects with 6,620 programmers from our industrial partner at Research Triangle Park, North Carolina, USA. From that data, we suggest:

> ***to reap the benefits of CI, do not just adopt CI tools; it is also vital to adopt CI best practices.***

To support this conclusion, we apply statistical analysis on the collected data, and 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, from our findings we observe that CI is not the 'silver bullet' for proprietary software development, as CI does not significantly improve software development for proprietary projects. The benefits that OSS projects enjoy through CI adoption, are associated with the practice of making frequent commits. Commits provide a means for practitioners to get rapid feedback from CI tools on the submitted code changes. For proprietary projects, practitioners commit less frequently, implying they do not use CI effectively to get rapid feedback, which eventually may lead to lesser bug resolution, and issue resolution. The observed phenomenon in proprietary projects can be termed as the 'cargo cult phenomenon', where a team adopts a CI tool, but not the best practices associated with CI.

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, we take care to stress the following point. There may well be benefits to CI that are not captured by the metrics we collect. For example, there may be cultural benefits in introducing CI tools to an existing infrastructure. That said, we caution advocates of CI not to hype their preferred approach. If users of CI technology were 'sold' on that technology based on a promise that it will (e.g.) instantly increase bug resolution then, practitioners from proprietary projects may be dissatisfied with CI tools.

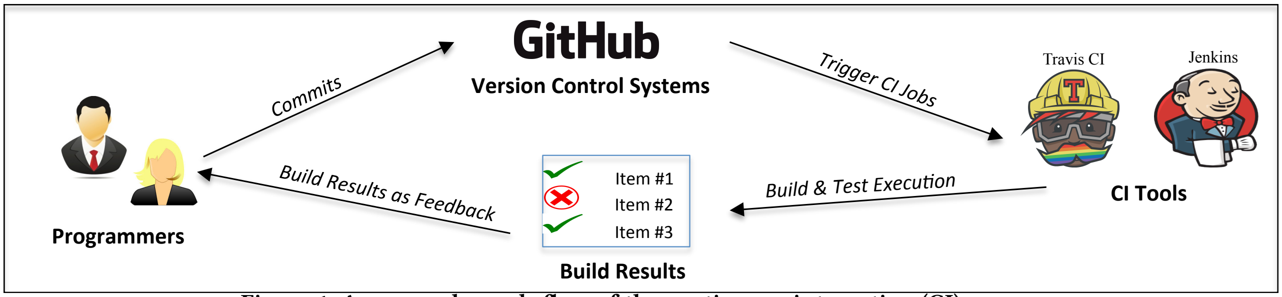


Figure 1: An example work-flow of the continuous integration (CI) process.

2 BACKGROUND

We first briefly describe the concept of 'silver bullet' and 'cargo cult'. Next, we provide a brief background on CI, and academic work related to CI.

2.1 Silver Bullets and Cargo Cults

In software engineering, a 'silver bullet' refers to as a methodology or technology that improves an existing software development process, by fundamentally changing the software process of an organization [5]. Researchers [30] [5] in prior work warned against 'silver bullets' in software engineering. Brooks [5] warned against hyping a technology as the 'silver bullet' that will automatically kill the werewolves which bedevil software engineering. Similarly, we warn that adopting continuous integration is not a silver bullet that dramatically improves proprietary projects.

In anthropology, a 'cargo cult' is characterized by the adoption of a specific culture, while missing essential practices of that culture [18]. The name 'cargo cult' derives from the belief which began among Melanesians since the 1900s where ritualistic acts such as, the building of an airplane runway will result in the appearance of Western goods (i.e., 'cargo'), via Western airplanes. We observe proprietary projects to adopt CI tools, but not implement a recommended best practice i.e. the practice of making frequent commits. We draw parallels between our observations for proprietary projects and the Melanesians: only adoption of CI tools is not enough to achieve the benefits of CI tools. The concern of this paper is that proprietary projects suffer from a 'cargo cult' mentality with respect to CI tools.

2.2 About Continuous Integration (CI)

In Extreme Programming (XP), CI is identified as one of the core practices to implement agility for software projects [1]. Humble and Farley [15] attributes the introduction of CI to Beck [1]. According to Duvall et al. [10], 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, initially usage of CI was not popular amongst practitioners [8]. Deshpande and Riehle [8] in their 2008 paper reported that OSS projects have not adopted the practice of CI. However, since 2011, with the advent of CI tools such as Jenkins [16] and Travis-CI [6], usage of CI has increased in recent years [13]. CI is also considered to be one of the fundamental pillars to implement DevOps, a relatively new software development methodology that advocates for the collaboration between the software development team and operations team [11] [15].

When a software team adopts CI, the team has to follow a set of practices and adopt a new set of tools [10]. According to the CI methodology all programmers has to check-in their code daily, which are integrated daily [10]. Unlike, traditional methodologies such as waterfall, in CI, programmers get instant feedback on their code via build results. In CI, the release cycle is short, compared to traditional methodologies. According to Hilton et al. [13], OSS projects that use CI release twice as often than those who do not use CI. To implement CI, the team must maintain its source code in a version control system (VCS), integrate the VCS with the CI tool so that builds are triggered upon submission of each commit [10]. 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. The commit triggers CI jobs on a CI tool such as Jenkins or Travis CI. The CI tool executes builds, code quality checks, and tests, and produces build results. These build results are provided to the programmers as a feedback on their submitted code changes. The build results can be communicated to programmers using bots, e-mails, or phone alerts [10]. Based on the build results, programmers make necessary changes to their code, and repeats the CI process again.

Much has been published about the influence and adoption of CI tools. Hilton et al. [13] observed that most popular GitHub projects use CI, and reported that the median time of CI adoption is one year. Olsson et al. [23] identified lack of automated testing frameworks as a key barrier to transition from a traditional software process to a CI-based software process. Stahl and Bosch [31] identified CI practice differences by performing a systematic literature review, and proposed a descriptive model for the identified differences in practices. Vasilescu et al. [33] mined GitHub projects that use Travis CI, and reported that adoption of CI increases code quality for OSS projects. In recent work, Zhao et al. [35] 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.

Recently, as more of the OSS tool suite is adopted by proprietary projects, we have become aware of large data sets where the influence of CI can be explored within proprietary projects. This raises an interesting research challenge: if we investigate proprietary software projects, will we find the same observations *outside* of the OSS software development process?

From the literature, we can list the following as exemplars of the expected benefits of adopting CI:

- Zhao et al. [35] reported that for OSS GitHub projects, after the adoption of CI tools:
  - The number closed issues increases.
  - The frequency and size of commits increases.
- Vasilescu et al. [33] reported that for OSS GitHub projects, number of closed bugs increases after adoption of CI.

Note that all of these findings are derived from OSS projects. With respect to the development process, proprietary projects have fundamental differences with OSS projects [24] [26]. 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

We first describe how we filter the collected projects, then we answer the four research questions.

3.1 Filtering

To perform our experiments we use OSS projects from GitHub, and proprietary projects collected from industrial partner at Research Triangle Park, North Carolina, USA. 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 [12]. 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 'npm'[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 the collaborating company.

Projects hosted on GitHub which gives researchers the opportunity to extract necessary project information such as commits, and issues [17] [4] [22]. Unfortunately, these projects can contain short development activity, can be used for personal use, and not be related to software development at all [17] [4]. 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 [17] [4].
  - *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.
  - *Personal Purpose*: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.
  - *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 [13]. 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 to answer our research questions. We repeat the procedure for both: OSS and proprietary projects.

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 [33]. 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

[1] https://github.com/npm/npm
[2] https://github.com/scala/scala

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.

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

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

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. We answer RQ2 by filtering out issues tagged as a 'bug'. To answer RQ2, 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.

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

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

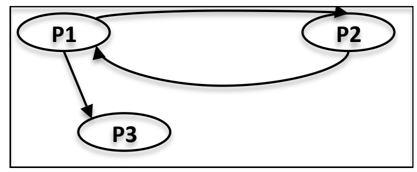
Similar to issues as stated in Section 3.2, we compute *NCB*, for all projects, before and after adoption of CI. We apply statistical tests, described in Section 3.6, to determine if CI has an influence on bug resolution.

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 [10]. 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 [29]. 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. In RQ3, we measure if adoption of CI increases 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 *File*1 and *File*2, 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 [3], bug reports [34], file change history [21] [2], to construct collaboration graphs, and observed that social structures exist between programmers. We take inspiration from these prior research, as mine software artifacts to characterize collaboration before and after adoption of CI. Similar to Meenely et al. [21] and Bhattacharya et al. [2], 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 *File*1 and *File*2. 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 [19] [2], indicating more collaboration between the programmers.

Projects can vary in node count, and needs to be accounted for [14]. Similar to Hong et al. [14] and Bird et al. [3], 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: Count of projects filtered for each sanity check of Filter-1.

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

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

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

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 [7], the Mann Whitney U test [20], 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. [28], 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', 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

Initially we start 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. After applying Filter-1 we obtain 661 OSS and 171 proprietary projects. We report how many of the projects passed each sanity check that are part of Filter-1, in Table 1. The projects 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.

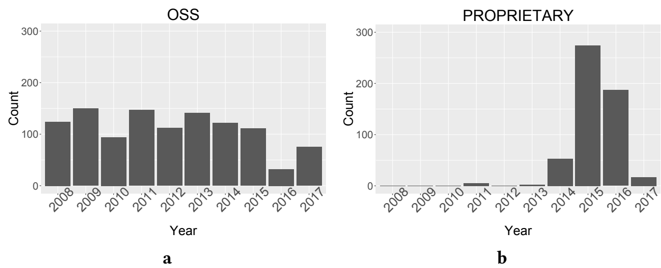


Figure 3: Count of projects that started in each year. Figure 3a presents the count of OSS projects. Figure 3b presents the count of proprietary projects.

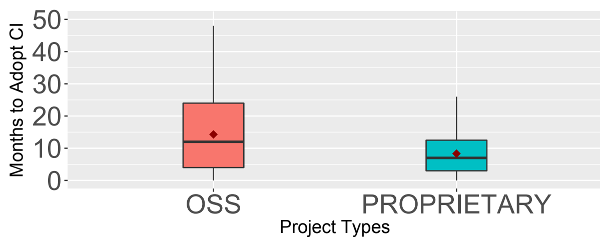


Figure 4: Time to adopt CI. The median time to adopt a CI technology is respectively 12.5, and 7 months for OSS and proprietary projects.

Table 2: Count of 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
Projects after filtering	150	123

From Table 1 we observe that 59.6% of the GitHub showcase projects pass the recommended sanity checks by researchers. The 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 obtain 150 and 123 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 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 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. [13] reported that for GitHub projects, the median time to adopt CI is 12 months.

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

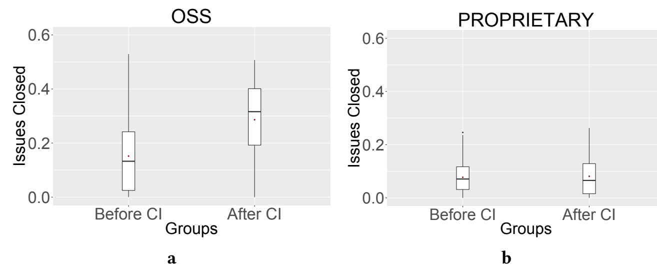


Figure 5: Normalized closed issues for OSS and proprietary projects. Figure 5a presents the normalized count of closed issues in OSS projects. Figure 5b presents the normalized count of closed issues in proprietary projects.

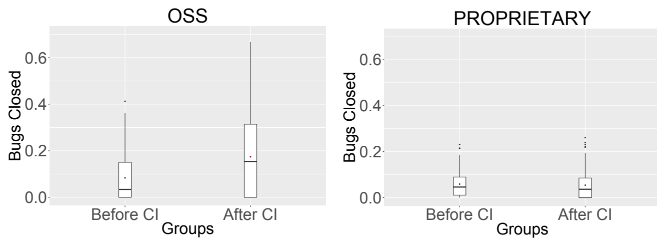
Table 4: Influence of CI on Closed Issues (*NCI*)

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

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

Zhao et al. [35] 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 5, 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 5. Furthermore, in OSS projects, considering 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. As expected, our OSS-related findings are consistent with Zhao et al. [35].

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.



a b

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

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

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

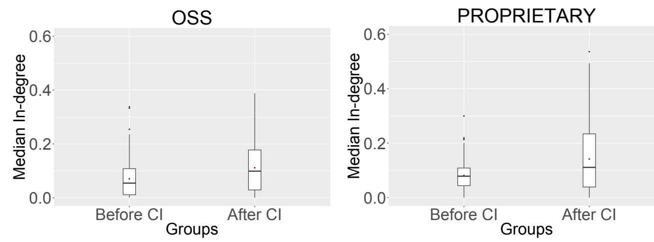
Vasilescu et al. [33] reported that for OSS GitHub projects, number of closed bugs increases after adoption of CI. For our set of OSS GitHub projects we expect to derive similar conclusions. We answer RQ2 by reporting how many bugs are closed before and after adoption of CI. We report the normalized count of closed bugs (*NCB*) in in Figure 6, 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 6 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 consistent with prior research [33]. 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 6 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.

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

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 present the box-plots in Figure 7. For both OSS and proprietary projects, the median in-degree significantly increases after adoption of CI. The effect size



a b

Figure 7: Influence of CI on collaboration (*NMID*). Figure 7a and Figure 7b 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

for OSS and proprietary projects is 0.2, which is small according to Romano et al. [28]. 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 RQ4: Does adoption of continuous integration influence commit patterns?

In prior work, Zhao et al. [35] 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 [35] 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 8. The 'delta' metric is represented in the Δ row. The 'delta' value for which we observe no significant difference is highlighted in grey. From Table 7 we observe 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 8. 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

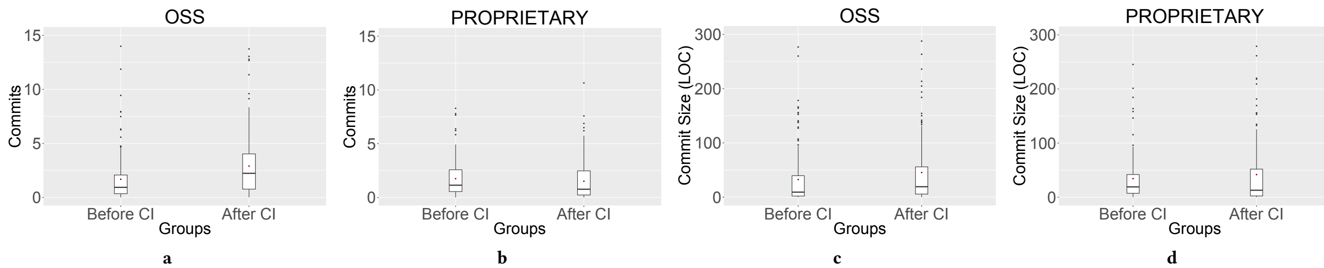


Figure 8: Normalized commit count and commit size for OSS and proprietary projects. Figure 8a presents the normalized count of commits in OSS projects. Figure 8b presents the normalized count of commits in proprietary projects. Figure 8c and Figure 8d respectively presents the normalized commit sizes for OSS and proprietary projects.

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

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 [35] as well as practitioners' perceptions [10].

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 [35] 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:

Is CI the 'Silver Bullet'?: Based on their analysis of OSS GitHub projects, Hilton et al. [13] 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, 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 is not the 'silver bullet'**, i.e., CI might not have the same influence on all software development attributes. Researchers [30] [5] in prior work warned against 'silver bullets' in software engineering. In Brooks' 1986 paper 'No Silver Bullet– Essence and Accident in Software Engineering' [5], he warns against hyping a technology as the 'silver bullet' that will automatically kill the werewolves which bedevil software engineering:

> *"There is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity." [5]*

We recommend teams to adopt CI if CI satisfies their needs. Based on our findings, if a software team is developing an OSS project and emphasizes on issue resolution then CI might be a good fit for the team. As another example, if a software team emphasizes on collaboration, then adoption of CI can be a good decision because empirical evidence suggests that CI influences collaboration for both: OSS and proprietary projects. In summary, we recommend practitioners to prioritize their needs before adoption of CI. We also recommend practitioners to continuously quantify the influence of CI on their needs, using software analytics.

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 [10]. If the build fails the CI tool provides rapid feedback on which code changes are not passing the inspection checks and/or test cases [10]. In this manner, the CI process provides rapid feedback about code changes to the programmer [10]. The programmer utilizes this feedback to fix the code changes. By making more commits, programmers are asking for rapid feedback about their code changes from the CI process. Programmers then use this feedback to fix their code changes, eventually leading to more 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. [10]; 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. Future research can investigate this explanation in details.

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***.

The 'Cargo Cult' Phenomenon: Above-mentioned discussion, and our findings presented in Section 4.4 hint an anti-pattern for proprietary projects: programmers of these projects are not implementing the practice of making frequent commits. One possible explanation of this anti-pattern can be attributed to how practitioners adopt a tool. Practitioners might be adopting CI tools, but not the best practices associated with CI adoption such as making frequent commits. Such behavior from practitioners can be attributed as the 'cargo cult' phenomenon [18]; i.e. when a culture is adopted, but the essential practices of the culture is ignored. For proprietary projects, we see practitioners treating CI as a cargo cult; i.e. they adopt CI tools, but not the best practices of CI. Keeping this phenomenon into account, we recommend practitioners to apply the best practices of CI, along with CI tool adoption.

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 [9]. 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 [25] [27]. 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.

Data-driven Decision Making: Based on our findings, we recommend software practitioners to mine features such as bug and commit metrics form version control meta-data, and use these features to make informed decisions. We also recommend practitioners to conduct software analytics in a continuous manner, instead of considering software analytics as a 'one-time-thing'. Software development process, and requirements evolve over time, and by applying software analytics continuously, practitioners can make decisions based on the recent data. We have observed from Section 4.4 that programmers' commit patterns change after CI tools are adopted. Through continuous analytics, practitioners may understand which practices are changing within the team, at what time, and which tools are influencing that change.

6 THREATS TO VALIDITY

In any large scale empirical study where multiple factors are explored, it is possible that some observations are due to spurious correlations. To increase the odds that our findings do not suffer from such conclusions 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 have observed that our findings related to bug resolution, CI adoption time, commit patterns, and issue resolution are consistent 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 help the practitioners of interest, in the same manner.

We also acknowledge other limitations of our paper. 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. We have relied on issues marked as a 'bug' to count bugs and bug resolution time, which can be limiting.

7 CONCLUSION

After mining 661 OSS and 171 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. CI is not the 'silver bullet' for software development, and practitioners need to (i) consider team's development context before adopting CI tools, and (ii) after adoption of CI, investigate if CI satisfies their needs by applying software analytics. We warn practitioners from the proprietary domain not to expect CI to improve their software development process, only through CI tool adoption. We also recommend that CI can improve the software development process, if CI best practices are applied along with adoption of CI tools.

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.

ACKNOWLEDGMENTS

The first and second authors conducted this research study as part of their internship with our industrial partner in Summer, 2017. We express our gratitude to our industrial partner for providing us the opportunity to mine the 538 proprietary projects.

REFERENCES

[1] Kent Beck. 2000. *Extreme Programming Explained: Embrace Change.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[2] 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
[3] 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
[4] 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.
[5] F. P. J. Brooks. 1987. No Silver Bullet Essence and Accidents of Software Engineering. *Computer* 20, 4 (April 1987), 10–19. https://doi.org/10.1109/MC.1987.1663532
[6] Travis CI. 2017. Travis CI. https://travis-ci.org/. (2017). [Online; accessed 15-October-2017].
[7] Norman Cliff. 1993. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological Bulletin* 114, 3 (Nov. 1993), 494–509.
[8] 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
[9] 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
[10] 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.
[11] C. Ebert, G. Gallardo, J. Hernantes, and N. Serrano. 2016. DevOps. *IEEE Software* 33, 3 (May 2016), 94–100. https://doi.org/10.1109/MS.2016.68
[12] Github. 2017. Github Showcases. https://github.com/showcases. (2017). [Online; accessed 13-October-2017].
[13] 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.
[14] 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
[15] Jez Humble and David Farley. 2010. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* (1st ed.). Addison-Wesley Professional.
[16] Jenkins. 2017. Jenkins. https://jenkins.io/. (2017). [Online; accessed 11-October-2017].
[17] 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.
[18] Lamont Lindstrom. 1993. *Cargo cult: strange stories of desire from Melanesia and beyond.* University of Hawaii Press.
[19] 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
[20] 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
[21] 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
[22] 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
[23] 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
[24] 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
[25] 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
[26] 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
[27] Everett M Rogers. 2010. *Diffusion of innovations.* Simon and Schuster.
[28] 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.
[29] 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
[30] D. Spinellis. 2007. Silver Bullets and Other Mysteries. *IEEE Software* 24, 3 (May 2007), 22–23. https://doi.org/10.1109/MS.2007.88
[31] Daniel Stahl and Jan Bosch. 2014. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software* 87, Supplement C (2014), 48 – 59. https://doi.org/10.1016/j.jss.2013.08.032
[32] 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
[33] 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.
[34] 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
[35] 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.

View publication stats