Work Practices and Challenges in Continuous Integration: A Survey with Travis CI Users

Gustavo Pinto*   Fernando Castor   Rodrigo Bonifacio   Marcel Rebouças
gpinto@ufpa.br, castor@cin.ufpe.br, rbonifacio@cic.unb.br,
marcel.reboucas@inlocomedia.com

Federal University of Pará, Belém - PA, 66075-110, Brazil
Federal University of Pernambuco, Recife - PE, 50740-560, Brazil
University of Brasília, Brasília - DF, 73345-010, Brazil
In Loco Media, Recife - PE, 50020-360, Brazil

SUMMARY

Continuous Integration (CI) is a software development practice that is gaining increasing popularity along the last few years. However, we still miss a collection of experiences regarding how software developers perceive the idea of Continuous Integration, in terms of its fundamental concepts, the reasons that motivate the adoption of this practice, the reasons for build breakage, and the benefits and problems related to Continuous Integration. To shed light on this direction, we conducted a user survey with 158 Continuous Integration users. Through a mostly qualitative investigation, we produce a list of findings which are not always obvious. For instance, we observed that (1) developers are not sure whether a job failure represents a failure or not; (2) Inadequate testing is the most common technical reason related to build breakage, whereas lack of time plays a role on the social reasons; (3) Although some respondents reported that Continuous Integration systems increase the confidence that the code is in a known state, some respondents also reported that there is a false sense of confidence when blindly trusting tests. This empirical study is particularly relevant to those interested in better understanding and fostering Continuous Integration practices either in open source or industrial settings. Copyright © 2018 John Wiley & Sons, Ltd.

1. INTRODUCTION

Continuous Integration (CI) is a core agile practice [3]. It is aimed at integrating code that is under development with the mainline codebase in a shared repository at least daily, leading to multiple integrations per day [12]. Each integration is properly verified by an automated build. Throughout this process, errors can be detected quickly, their causes can be logged and, when appropriate, located with less effort. As a consequence, the overall development process can be sped up [17]. Due to its claimed benefits, software development teams are adopting Continuous Integration practices and tools at a fast pace [7, 18, 24].

Although Continuous Integration is a practice that does not require any particular tooling to deploy, tools that automate the process of compiling, building, and testing software are more and more common, since they greatly decrease the barriers for its adoption. Notable examples include

---

*Correspondence to: Gustavo Pinto, Faculty of Computing, Universidade Federal do Pará (FACOMP-UFPA), Rua Augusto Corrêa, 1 - Guamá, Belém - PA, 66075-110, Brazil.

CruiseControl[†], Hudson[‡], and Travis CI[§]. When integrated with social coding environments, such as GitHub and BitBucket, these tools become not only helpful for automating the Continuous Integration process, but they also promote a high degree of social transparency [4]. Thus, team members, project managers, and even external users can easily be aware of the current (or even historical) build status of a given project.

In spite of the increasing adoption, the existing set of tools, and the well-known benefits, there is a lack of a collection of experiences regarding whether software developers are taking advantage of Continuous Integration systems. According to Hilton and colleagues [18], and also pointed out in the practitioner arena [24], more research is needed to better understand the dynamics, process, practices, and challenges involved when integrating with such systems. Starting from this premise, this paper presents an empirical study aimed at illuminating the motivations for adoption, the reasons for build breakage, and the benefits/problems of Continuous Integration systems. Specifically, the question we are trying to answer is:

RQ. What are the perceptions from Continuous Integration users, in terms of the fundamental concepts, reasons for build breakage, and the benefits and problems of these systems?

To provide answers to this question, we conducted an online survey with 158 Continuous Integration users that proposed code changes that broke the build of systems they worked on. In this work, we rely on the Travis CI build processes, the most popular Continuous Integration tool to date [18]. According to the GitHub website [24], as of 2017, about 50% of the 50 million open source projects hosted on GitHub have Travis CI enabled. Since this corpus includes forks and toy projects, in a more systematic study, Beller and colleagues [7] found that about 30% of the 2 million studied open source projects hosted on GitHub that could leverage Travis CI, do use it actively. Through quantitative and qualitative analyzes of the answers of this survey, enriched with the findings of related literature, our study produced a set of findings, many of which were unexpected. In the following, we highlight three of them.

Developers are not sure about what constitutes a successful build. Although the build can have roughly two states (passing or failure), 33.1% of the respondents are not sure whether a job failure also represents a build failure or not, which is in sharp contrast with Martin Fowler's definition [13]. This behavior is in part due to flaky tests or misconfigured job files.

Inadequate testing and time pressure are the most common technical and social reasons related to build breakage. Still, although developers spent 25% of their development time on testing activities [5], we noted a lack of a testing culture, which might lead developers to skip or write poor/naive tests. Also, some developers mentioned that this time pressure is self-imposed, because they are eager to contribute.

The confidence in Continuous Integration systems is a double-edged sword. While developers become more confident to perform required code changes (since the Continuous Integration system will automatically test the submitted changes), blindly trusting on tests may also hide important perils to patch integrators (since tests can be flaky [21] or insufficient [6]). Interestingly, pull request research suggests that maintainers rely on contributors test to assess the quality of the proposed changes [15, 16].

This work is built upon a previous study [25]. Several factors were improved since the publishing of the previous work such as: (1) a more depth analysis of benefits and problems related to CI adoption, as well as the discussion about questions of the survey that were not explored in our previous work, and (2) a better discussion of our findings, in the light of recent related work. Indeed, the current paper is twice longer than the previous one.

[†]`http://cruisecontrol.sourceforge.net/`
[‡]`http://hudson-ci.org/`
[§]`https://travis-ci.com/`

2. BACKGROUND

Continuous Integration started to gain popularity in the 2000s, after appearing as one of the 12 original eXtreme Programming (XP) practices and after a blog entry by Martin Fowler about the subject [13]. Continuous Integration is the process of integrating the contributions of a team of developers into a shared repository, multiple times per day. With Continuous Integration, each addition of code to a repository will trigger an automated build, followed by the automatic execution of a test suite and static-analysis tools. By integrating often, integration errors can be detected and fixed as early as possible. Such a process, added to the fact that the repository can be tested against multiple platforms and in different configurations, helps to create more reliable software and reduce common risks [12].

A typical Continuous Integration process works as follows: a developer sets up the Continuous Integration system (*e.g.,* Travis CI) to work with his source control repository (*e.g.,* GitHub). Continuous Integration systems, in general, have the concept of build and Travis CI, in particular, has also the concept of job. A build is triggered when a developer commits changes to a source control management system. According to Beller *et al.*, a job is the concrete manifestation of the build in one pre-configured environment. Jobs allow the customization of the building and testing process (*e.g.,* which compilers to use, which commands to run before or after the build, or which parameters to use on the test suite). Therefore, developers can test their modifications in many environments that they do not have access to (as mentioned in the Section 5.1, under "Cross-platform testing"). A job is either performed either periodically or when a specific event occurs (such as a committed change). When a job is triggered, a life-cycle that starts. First, the job starts by initializing a virtual machine, cloning the repository, and installing relevant packages needed for building the software system. A job is said to be *errored* if any problems occur during the initializing phase. Afterwards, the job builds and tests the software system. If the test execution finds any problem, the job is said to be *failed*. Otherwise, the job is *succeeded*. A job can also be cancelled. The number of jobs per build can range from one to several hundred.

The Feedback-Driven Development (FDD) [4] workflow is illustrative here. FDD is based on quality assurance methods commonly found in software development practice, which includes the GitHub pull request system, its code review process, and tools that support these activities. When a pull request is submitted, it will be associated with a build status, which is the result of the automated process. If a build is broken, a reviewer can quickly ask for the fix without having to build or test the pull request locally, which greatly reduces her workload. Otherwise, if all tests have passed, the reviewing process can begin. At the end of the code review process, when the project integrator merges the proposed changes, another build is triggered in order to make sure that the repository is in a build-passing state. Therefore, there are at least two builds for each pull request proposed (and another build is triggered for each additional commit in the pull request).

3. SURVEY DESCRIPTION

In order to investigate the perception of Continuous Integration users, we conducted an online survey. Our target population consists of software developers that have broken at least one build in a non-trivial open source software project.

3.1. Subjects

Our subjects are software developers working on non-trivial open source projects. Therefore, we started our search by selecting representative open source projects using the same approach introduced by Ray and colleagues [27]. At first, we selected the most popular programming languages on GitHub, measured by the total number of files created in each programming language. Using this criteria, the chosen programming languages were: C, C++, C#, Clojure, CoffeeScript, Erlang, Go, Haskell, Java, JavaScript, Objective-C, Perl, PHP, Python, Ruby, Scala, and TypeScript. However, we removed CoffeScript and TypeScript since Travis CI does not offer support for these

programming languages. For each one of the remaining programming languages, we selected the 50 most popular projects with a Travis CI configuration file (`.travis.yml`), considering the number of stars, which is an explicit way for GitHub users to manifest their satisfaction with a project [9]. The `.travis.yml` file stores the configuration information required to run Travis CI's service. We ended up with an initial list of 750 Travis CI-configured projects. According to Beller and colleagues [7], the adoption of Travis CI is roughly uniform among the programming languages that Travis CI supports. Table I shows some characteristics of the build behavior of the 750 target software systems, considering each studied programming language.

Table I. Build behavior of the 750 projects selected per programming language.

Prog Language	# builds	# successful builds	# broken builds	Build duration (sec)
C	53,957	38,958	14,999	1,390
Clojure	8,645	7,477	1,168	271
C++	64,987	48,410	16,577	2,077
C#	11,189	5,771	5,418	656
Erlang	12,525	10,083	2,442	1,575
Go	90,875	73,883	16,992	233
Haskell	32,207	24,440	7,767	2,951
Java	32,436	22,695	9,741	1,250
JavaScript	85,265	70,989	14,276	529
Objective-C	12,963	9,722	3,241	481
Perl	20,217	16,681	3,536	729
Php	71,128	53,725	17,403	886
Python	102,802	69,698	33,104	1,470
Ruby	93,870	68,135	25,735	1,756
Scala	44,444	34,771	9,673	2,235
Total	**737,510**	**555,438**	**182,072**	**1,275**

When manually analyzing these projects, we observed that some of them do not properly fit the purpose of this study, because:

- **Some projects are not software projects**. We found that several popular open source projects are not software projects. Instead, these projects are either mirrors of other repositories[¶] or used to share bookmarks[‖] or textbooks[**]. We excluded these non-software projects.
- **Some projects are not active**. We found 30 projects that are not active. We believe it is important to focus on active projects because we wanted to measure ongoing development. Therefore, we selected projects that have at least one commit and at least 2 committers in the last 12 months. We excluded non active projects.

After this manual process, we ended up with a list of 682 projects. For each project, we queried the Travis CI API in order to retrieve builds' metadata. However, the Travis CI API did not answer the requests for 16 of our subject projects. Unfortunately, some of these requests are for projects such as `Angular.js` and `Rails`, which are highly active and have a long history of software builds. This fact reduced our final sample to 666 projects.

Overall, these projects performed about 737,000 builds, among which 182,072 (25%) have failed. Our subjects constitute a random sample of 1,100 software developers, with valid email addresses, that have contributed to at least one of those build failures. We chose to focus on developers with broken builds because they might have to provide additional commits to fix the build. Therefore,

---

[¶]For instance, `http://www.GitHub.com/WordPress/WordPress`

[‖]For instance, `https://GitHub.com/vinta/awesome-python`

[**]For instance, `https://GitHub.com/SamyPesse/How-to-Make-a-Computer-Operating-System`

they might have more experience with Continuous Integration than developers that have never faced problems with it, *i.e.,* they have to read the logs, understand the root cause, and perform additional commits. We contacted participants individually by email inviting them to participate in the survey.

3.2. Design

The survey used in this work was based on the recommendations of Kitchenham *et al.* [20], following the phases prescribed by the authors: planning, creating the survey, defining the target audience, evaluating, conducting the survey, and analyzing the results. We also employed a number of principles used to increase survey participation [30], such as *liking*, *e.g.,* we sent personalized invitations, *anonymity*, *e.g.,* the survey was completely anonymous, and *brevity*, *e.g.,* we asked closed and direct questions as much as possible. We set up the survey as an on-line questionnaire (it can be found online at the companion website: `http://gustavopinto.org/ survey-travis-ci/`). Before sending the link to our subjects, we created a first draft of the survey and informally presented it to several colleagues. Based on the colleagues' remarks, we refined some of the questions and explanations, mainly to make them more precise. Along with the instructions of the survey, we included some examples as an attempt to clarify our intent. Participation was voluntary and the estimated time to complete the survey was 10-15 minutes. Over a period of 30 days, we obtained 158 responses, resulting in 14.4% of response rate.

3.3. Questions

Our survey consisted of 15 questions, 7 of which were open-ended (free-response). The survey was divided in five sections: (1) technical background of the participants, (2) experience with Continuous Integration techniques, (3) Continuous Integration fundamentals, (4) reasons for build breakage, and (5) benefits and problems of Continuous Integration systems.

(1) Background. We started by asking technical background information, such as their current position (Q1), who do they work for (Q2), how often do they contribute to open source software (Q3).

(2) Experience with CI. In this group of questions, we asked how familiar are they with Continuous Integration techniques (Q4), how many projects to which they have contributed use Continuous Integration techniques (Q5), whether or not they have ever configured a project to use Continuous Integration techniques (Q6) and, finally, what reasons motivated them to use Continuous Integration techniques (Q7).

(3) Continuous Integration fundamentals. Questions in this group aim to understand how developers perceive builds and jobs—the fundamentals of Continuous Integration systems. We asked their opinions on how often a commit breaks the build in a popular open source project (Q8), if they found cases where a commit broke a particular job, but not all jobs (Q9), if they consider a commit that does not break all jobs as a faulty one (Q10) and, if so, to elaborate on why do they believe that (Q11).

(4) Reasons for build breakage. Here we asked what are the technical reasons (Q12) and social reasons (Q13) that might influence build breakage.

(5) Benefits and Challenges of Continuous Integration systems. Finally, we asked their opinion about the benefits (Q14) and problems (Q15) related to the use of Continuous Integration techniques.

To give respondents the most flexibility, no question was mandatory. To avoid unreliable responses, developers were free to skip questions that they did not feel confident to answer.

3.4. Analysis

We conducted two forms of data analysis on the survey responses. First, we examined the distributions of responses in an effort to build a broad view of each closed question. We present

these distributions in chart form. In the second analysis, we followed open-coding and axial-coding procedures [32]. We highlight the main themes that emerged along with quotes from the survey. Among similar opinions, we chose to quote only the one we considered the most representative for each case. Similarly, for the sake of simplicity, we only quoted a single answer per theme. Still, although some quotes have typos, we decide to keep them in order to make the answers more trustworthy.

As an example of an interesting answer, we quote one response suggesting that Continuous Integration systems should be considered in advance and may not be easily and readily available for any system:

> "*Continuous Integration requires that applications be written from scratch with the assumption that they be buildable, testable and runnable anywhere, not in their special snowflake developer's environment. This is often hard to achieve in a project that is old and crufty, as it requires actual restructuring of the application.*"

4. RESPONDENT BACKGROUND

Among the respondents, 67.5% are software developers, 7% are software architects, 2.5% are project managers, and the remaining 23% play other roles (Q1). Moreover, most of the respondents work for the software industry (79.5%), whereas 34% work for open source initiatives, 10.9% work for academia, 2.6% work for the government, and 5.1% work for other agencies (Q2). However, Figure 1 shows a major disparity regarding their participation in open open source projects (Q3). Although 10.1% of the respondents contribute on a daily-basis and 29.1% contribute on a weekly-basis, 15.8% contribute on a monthly-basis, 13.3% contribute once every 2 months, and 8.2% contribute once every 6 months. Finally, 3.2% contribute yearly and 20.3% have made just few contributions.
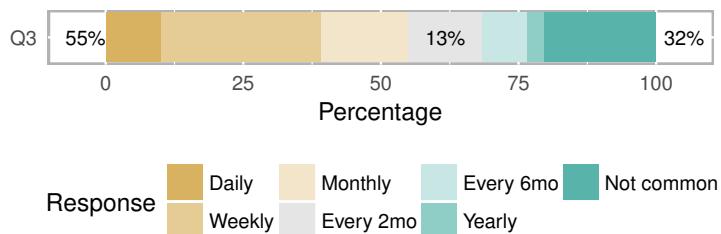

Figure 1. Q3. How often do you contribute to open source software? (%).

5. EXPERIENCE WITH CONTINUOUS INTEGRATION

As Figure 2 suggests, 29.5% of the respondents said that they are "Extremely familiar" with Continuous Integration techniques, 36.5% said that they are "Very familiar", 20.5% are moderately familiar, and 13.5% are slightly familiar. No respondent mentioned to be "Not familiar at all".

As we can see in Figure 3, we found a similar result when we asked, among the projects they contribute to, how many projects use Continuous Integration techniques (Q5). Among the respondents, 18.4% mentioned that Continuous Integration techniques are used in all projects that they contribute to, whereas 39.9% use Continuous Integration in most of the projects, 29.1% in some of the projects, 12% in just a few projects, and 0.6% in none contributed projects. These results corroborate with recent work that suggests that a large proportion of open source active projects use Continuous Integration techniques [18]. However, we found that about 25% of the respondents have never configured a project to use Continuous Integration (Q6).
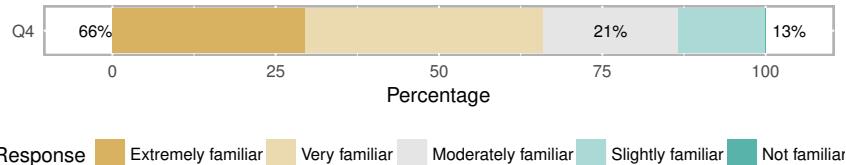
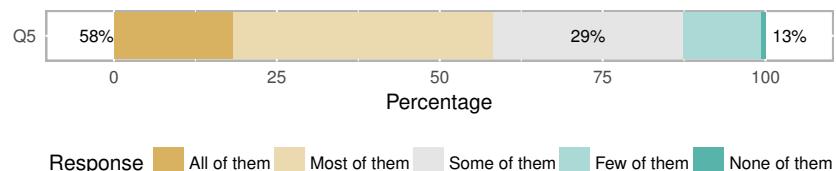Figure 2. Q4. How familiar are you with Continuous Integration techniques? (%).

Figure 3. Q5. How many of the projects you contribute with use CI? (%).

5.1. Reasons for Adoption

We found 10 different reasons that motivate (some) of our subjects to configure a project to use Continuous Integration techniques (Q7). We next quote each one of them.

- **Speed up development practice (23 occurrences)**. Saving time was the most common reason that motivates developers to configure Continuous Integration techniques on their projects. This particular reason was also commonly reported in the related literature [18]. Saving time includes fast feedback on breakage, *e.g.,* "*quick feedback on produced development for adding business value*", while time spent on running tests locally was highlighted by one respondent: "*running *all* tests locally is taking to long (10+ mins)*". This finding is particularly relevant in the light of recent related work that suggests that only a small part of the tests ($\leq 5\%$) take more than two minutes to execute [5].
- **Improve software quality (18 occurrences)**. Some developers believe that Continuous Integration techniques can improve software quality as a whole, either by "*ensuring quality of releases*" or "*ensuring both, mine and other people changes, do not break anything*". Some recent papers go along these lines and suggest that CI helps maintain code quality [7, 18, 36]; for instance, the work of Hilton provided evidence that projects that use CI release more frequently than those that do not [18].
- **Catch regressions/bugs (16 occurrences)**. Some developers agree that Continuous Integration techniques can be useful to avoid regressions, as mentioned by one respondent: "*much easier regression testing in complex projects*". As a result, Continuous Integration practice might "*prevent to release a bug software into production environment*". This finding goes in line with Fowler's statement that "Continuous Integration does not get rid of bugs, but it does make them dramatically easier to find and remove" [14]. Hilton and colleagues also reported that catching bugs earlier is a top reason for adopting CI [17, 18].
- **Enforce automated software testing (15 occurrences)**. The guarantee that tests are run for each and every commit was also a driving force towards Continuous Integration techniques, as explained by one respondent: "*To make sure tests are always run, since humans can be inconsistent on that*". Interestingly, as pointed out in another study [7], only about 20% of open source projects that use Travis CI do not include a testing phase in their CI.
- **Personal needs (9 reasons)**. We found that some developers start using Continuous Integration techniques due to personal needs, as one developer said: "*I wanted to get better at test driven development*". As pointed out in another study [38], when properly integrated in the software development environment, CI can foster other software development practices, such as (1) commit often, (2) TDD, or even (3) DevOps.

- **Cross-platform testing (8 occurrences)**. The ability to test on different platforms also drew attention to some developers, as experienced by one respondent: "*mostly to test software on Linux, to which I rarely have access*". As reported elsewhere [7], cross-platform testing could be helpful to uncover failing tests that would not be easily found by running tests locally.
- **Credibility (7 occurrences)**. Some developers believe that Continuous Integration techniques add credibility to their projects, as one respondent indicated: "*it makes the project look more legit*". This finding was corroborated in a recent study that indicated that repository badges (*e.g.,* the build status) embedded into a project yield higher credibility [34].
- **Best Practices (7 occurrences)**. Some developers advocate in favor of software development best practices, as experienced by one respondent "*It's a proven best practice (from my own experience)*", and evidenced by another one: "*Common practice, I would never set up a project without Continuous Integration*". Notwithstanding, Continuous Integration is considered an established best practice of agile software development methods like eXtreme Programming [3].
- **Improve transparency (3 occurrences)**. The social transparency features provided by the Continuous Integration systems made it easier to assess the quality of a change, as one respondent cited: "*See if other's pull requests doesn't break the code base before merging*".
- **Improve communication (2 occurrences)**. Finally, we found that Continuous Integration techniques can be useful to improve communication between team members, as one respondent pointed out: "*[improving] cooperation with other engineers*".

The reasons found corroborate and expand on already-discussed reasons in recent literature [7, 24, 35]. For instance, in a GitHub blog post [24], it was discussed that developers should consider trade-offs that include complexity or simplicity (if the application is small, there is no need to use a CI system that operates on dozens of platforms, programming languages, and frameworks). In addition, the work of Beller and colleagues [7] suggest that dynamically-typed programming languages, such as Ruby, are more prone to embrace Travis CI. This finding might be in part due to the lack of a strong type system, and therefore might corroborate with our "Catch regressions/bugs" adoption reason. In contrast, the work of Vasilescu *et al.* [35] suggests that the possibility of enhancing automation is a key factor for adoption Travis CI. Another paper suggests that CI is a key enabler of other software development practices. For instance, DevOps, in particular, can be more easily achieved if the build pipeline is automated [22, 38]. Finally, generally speaking, the use of badges in repositories, such as the one that displays the build status, is not only considered a best practice but also improves transparency and credibility [34].

6. CONTINUOUS INTEGRATION FUNDAMENTALS

As aforementioned in Section 2, Travis CI has the concepts of build and job. Since a job exist within a build, the relationship between build and jobs is often 1 to N, respectively.

One scenario that may arise in this landscape is that, due to rather specific environment needs, one or more jobs may fail (*e.g.,* on Linux) while the remaining ones may pass (*e.g.,* on OSX). Indeed, 83.2% of our respondents have already faced this scenario in practice (Q9). When asked if they consider a commit that does not break all jobs as a faulty one (Q10), 60.4% of them said "Yes, I think the build has failed" and 6.5% said "No, I don't think the build has failed". Interestingly, 33.1% of the respondents are not sure of ***what makes a successful build***. In contrast, in his introductory blog post, Martin Fowler stated that "If all [required] steps execute without error or human intervention and every test passes, then we have a successful build" [13]. When we asked them to elaborate on why they are not sure about this, they provided a great variety of reasons.

The most common reason is that ***tests may be faulty***, as one respondent mentioned: "*A commit may be fine, but the test may be faulty*" (7 occurrences). Another respondent reported that "*It depends if the tests are written in a reliable way*". Aligned with the quality of the tests, we also observed that ***misconfigured job files*** play a role (6 occurrences), as stated by one respondent: "*If the failure is due to a misconfigured (or unsupported) build environment, then 'it depends'*". Although there are

existing tools for verifying the syntax of the job files (*e.g.,* `linter`), such tools are not helpful in verifying their semantics.

Moreover, seven respondents reported that **flaky tests** that may affect only few jobs. A flaky test is a test that could fail or pass for the same configuration, therefore, can create several problems since it can be hard to reproduce [21]. One respondent mentioned that "*[flaky tests] are non-deterministics and sometimes restart of particular job helped*". Other respondents said that flaky tests may happens due to "*network errors that never happened locally*" or because "*a deployment script may fail*". Also, it is known that Continuous Integration techniques can take advantage of environments that are beyond the team's control or too complex to configure. This advantage can highlight the lack of **backward/forward compatibilities**, as one respondent reported: "*Sometimes commits break builds with older environments (e.g. Java 1.6). [A given job failure] may not matter if the intention is to drop support for these [environments].*" The same premise holds for tools that are still on beta, "*Sometimes there would be a beta version of a compiler that would expose an issue with the compiler itself*".

In addition, five respondents do not consider a build to be a failure if the problem is in a **small yet known part of the code**, as described by one respondent: "*sometimes the test is broken or doesn't apply to the release under test*", that is, the respondent acknowledges that they have a problem, but this problem is not a priority at the moment. Similarly, two respondents found that some jobs are of **less importance**, as one of them highlighted "*In certain cases, some jobs are less important, and not critical to the application*". However, the respondent also raised the fact that "*these cases should be fully documented*". Finally, two respondents highlighted the fact that a **previous failing could forward its status to the next build**, if the next build did not fixed the problem. Therefore, even if the new build did not introduce any problems, it would be labeled as a failure (*e.g.,* "*Sometimes some part of the build system is flakey and not my commit.*").

7. REASONS FOR BUILD BREAKAGE

In this group of questions we consciously asked respondents what are the (1) technical and (2) social reasons for build breakage.

7.1. Technical Reasons for Build Breakage

We found several technical reasons that might explain build breakage. Among the most common ones, there is **inadequate testing** (33 occurrences). Respondents mentioned that some tests are "*Badly written that fail with minor bugfixes*" or even "*not enough tests*". However, one respondent mentioned that they might be tempted to skip test execution, since "*running a test suite may be too slow and skipped*". These respondents use Continuous Integration systems as a way to speed up the development process, while still relying on the test suite. This finding corroborates to recent related work that suggests that developers *offload* running tests to the CI [7]. Interestingly, running tests locally is about three times faster than offloading to the CI [5].

Moreover, some respondents claimed that **version changes** might play a role (12 occurrences). For instance, "*[the] version of a language component is different, and the change made to the language cause breakage*". Another reported drawback is related to **dependency management** (8 occurrences). As one example, one respondent mentioned that "*people sometimes don't update dependencies, so the Continuous Integration server detects errors that do not happen locally*". It is important to note that **dependency management** and **version changes** are two key activities of build systems. Therefore, developers that may face technical problems with the usage of build systems may also face the with Continuous Integration systems. According to related work, most build errors are due to missing dependencies [29]. In contrast, the work of Beller *et al.* [7] suggests that 59% of the build breakages in Java open source projects are due to test failures.

Still, we noticed that some developers are facing barriers with the **intricacy of the code base** (14 occurrences). These developers claimed that "*unfamiliarity with the architecture of the code and overall module interactions*" is an obstacle, which might be due to a "*lack of experience with the*

project". Such developers might be first-time contributors [31], or even casual contributors [26]. As opposed to the ***intricacy of the code base***, some developers ***missed edge cases*** (5 occurrences), such as "*syntax errors, formatting errors if using a linter*" [33], which, according to one respondent, are due to an "*underestimation of impact of small changes*".

Finally, other not so common technical reasons include: ***git usage*** (4 occurrences), e.g., "*git rebase and left the work on hold for too long*" and ***timezones*** (2 occurrences), e.g., "*we had to be careful to leave the build broken at the end of the work day; other developers might work in other timezones*".

Ultimately, five developers either perceived no technical reason related to build breakage or cannot recall.

7.2. Social Reasons for Build Breakage

When analyzing the answers, we found that the most common social reason associated with build breakage is ***time pressure*** (36 occurrences). One particular respondent mentioned that he self-imposed this time pressure, because of "eagerness to help", as he mentioned: "*I'll just make that change right now!*". Moreover ***lack of domain knowledge***, mentioned by 18 respondents, might hinder contributions from external contributors (*e.g.,* "*lack of familiarity with a specific project*"). This is particularly relevant for open source projects, since newcomers often do not know how the project is organized or how to start contributing [31]. In addition, the social facilities brought by social coding websites such as GitHub and Bitbucket lowered the barriers for one placing a contribution to an open source project. Therefore, casual contributors that happen to enjoy one particular open source project might be tempted to contribute [26]. However, without proper guidance or documentation, such contributors might not be aware of the workflow (*e.g.,* "*I didn't know about the linter (or that Continuous Integration was being used) until after I submitted my patch.*").

Another social reason is the ***lack of testing culture***, as evidenced by 17 respondents. In this particular finding, the majority of the respondents acknowledged that this is due to "*people not running the tests and build on their machines before pushing the changes*", which corroborates with the findings of Beller *et al.* [5] that suggest that developers *offload* the test execution to the CI service. We also found cases of "*poor testing and reviewing of the commit beyond the 'immediate problem' the developer is attempting to rectify*". Similarly, we found 8 respondents that believe that ***carelessness*** on the part of developers can cause build breakage (*e.g.,* "*just be happy I'm commiting to the project, somebody else can test if what I did works*"). Confluent with ***carelessness***, there is ***overconfidence*** on the part of 11 respondents. Before breaking the build, they thought that "*the change is alright*", "*this is such a trivial change*", or "*this is only a small fix, it should not break anything*".

Still, we found six respondents that claimed that the ***lack of communication*** hindered the solution of a task, leading to build breakage. This finding puts some related work in challenge, since it was reported that one of the benefits of CI is to improve communication [34]. Indeed, most of the respondents mentioned a ***lack of communication*** skills (*e.g.,* "*disincentive to ask folks for help*" or "*not knowing who to ask for help*"). Yet, two respondents believe in ***moving fast and breaking things***, as one of them clarified "*I'll merge a commit that isn't quite ready as a clear signal of intent to move in that direction.*". Aligned with the desire of ***moving fast and breaking things***, seven respondents are convinced that ***it is fine to break the build***, as one respondent exemplified: "*CI is there for you not to be afraid for broken builds in a branch*". Notwithstanding, one respondent warned that it is fine "*as long as it's not merged into trunk/master until it's green*". A similar finding was reported in the work of Hilton *et al* [17], when the authors observed that, when using CI, developers are less worried about breaking the build. Finally, five respondents mentioned that they are not aware of any social reason.

8. BENEFITS AND CHALLENGES OF CONTINUOUS INTEGRATION SYSTEMS

In this final set of questions, we elucidate some benefits while highlighting some hidden challenges.

8.1. The Benefits of Continuous Integration Systems

The most often-cited benefit of using a continuous integration system is to **catch problems as early as possible**, reported by 32 respondents. It is important to note that problems can be described in terms of new bugs and regressions. One respondent summarized this benefit as "*Being aware of when/where breakage occurs greatly accelerates solution*". Moreover, we perceived **automation** as one of the main benefits, as pointed out by 19 respondents. Automation, however, is not only related to automated testing, as one respondent highlighted, but can be described in terms of "*automated code quality enforcement, automated release cycles, automated deployment*". Generally speaking, automation is aimed at "*performing a wide range of manual steps that a human would not normally be bothered to check*".

Another benefit is related to improving **software quality**, as described by 18 respondents. Although most of the respondents have mentioned "software quality" with no additional details, some respondents have cite specific quality attributes, such as (1) increased stability, (2) quality of code and test, and (3) quality of documentation. One of the anonymous respondents has detailed how Continuous Integration can improve software quality: "*When you use CI, you have a good health check in your code base, and from time to time you can keep looking if any other dependencies didn't break your package. It is a warrant of quality of your code*". Another interesting benefit is the **fast development cycle**. According to one respondent, this happens because "*[its] extremely fast development cycle [made it] easy to try out stuff & easy to add new testing (you don't need to coordinate with other people as much, which is time-expensive)*".

Furthermore, 16 respondents mentioned **cross-platform testing** as an effective method for "*compiling for multiple different targets (x86, arm, windows, linux etc)*". Another respondent reported that "*cross-platform testing acts as a sort of documentation*", and "*[it can be configured] with moderate ease and in a fraction of the time*". Some respondents also mentioned that "*this [task] is not feasible or cost-effective to do manually*". Interestingly, as pointed out by Beller *et al.* [7], on average, five different cross-platform environments are tested in open source Travis CI. Still, 10 respondents mentioned that Continuous Integration systems give more **confidence** to perform required code changes, as one respondent described: "*Depends on the coverage, but some sort of confidence that introduced changes don't break the current behavior*". The same respondent went further and mentioned that "*I assume the most critical parts of the system have been covered by test cases*". It is important to note that test cases can only ensure that *known* bugs stay fixed. As Dijkstra claimed in the 1970s [11], and extensively evaluated in the following decades (*e.g.,* [37, 39]), "*program testing can be used to show the presence of bugs, but never their absence*" [11].

8.2. The Challenges of Continuous Integration Systems

As regarding the hidden challenges associated with Continuous Integration usage, we found that 31 respondents are having a hard time **configuring the build environment**. This finding corroborates with a recent study that suggests that maintaining and setting up the CI environment is among the most common barriers developers face [17]. Although some respondents agreed that the burden placed by the configuration process is often doable (*e.g.,* "*Some extra overhead and complexity for setting them up / maintaining the configuration. Not usually a big problem*"), we found cases where the configuration process is far from trivial (*e.g.,* "*I found it hard to setup some distributed/multicomponent tests. This partially can be resolved by the containers.*"). It is important to observe that, according to a recent study [18], a number of open source projects that use Continuous Integration performed 5 or less changes to their Continuous Integration configurations. Also, many changes were due to the version of dependencies that had been changed. Therefore, this additional overhead might not place a constant burden on the software development team. Overcoming such technical problems is particularly challenging for onboarding newcomers (9 occurrences) since they "*do not understand what Continuous Integration does and what it doesn't*".

Another recurring problem is the ***false sense of confidence*** (25 occurrences). As opposed to the ***confidence*** benefit, respondents described the ***false sense of confidence*** as a situation where developers blindly trust the tests. One respondent synthesized this problem as:

> "*Over-reliance on a passing build result can encourage a reviewer to merge code without a thorough review. The Continuous Integration pass is only as meaningful as the test coverage.*"

This over-reliance on software testing, in general, and unit testing, in particular, as a measure of quality has been thoroughly discussed in the software testing literature [10, 37]. For instance, tests can be insufficient, have poor quality, or even be incorrect. Furthermore, some respondents have associated this false sense of confidence to insufficient testing (*e.g.,* "*If test coverage is not sufficient, passing all the time doesn't mean much*"). However, high coverage percentages alone cannot ensure code quality, since intermittent, non-deterministic, or unknown bugs may not be detected by the periodic maintenance tests [1, 21]. In fact, there is a complex relationship between test-coverage and defect-coverage [10]. Interestingly, recent pull request research suggests that project maintainers rely on contributors tests to assess the quality of the proposed changes [15, 16].

Moreover, we observed that the use of Continuous Integration techniques require developers a certain level of ***discipline*** (12 occurrences), as one respondent mentioned "*it takes a lot of self-organization to work under pressure of 'master must build'*". In addition, one respondent claimed that such ***discipline*** might introduce a lack of focus, since "*instead of only focusing on the code of your application, you also focus on the code of your build system. Basically, a software engineer becomes an infrastructure engineer, which is much less interesting*". Likewise, ***additional effort*** was also evidenced by 3 other respondents (*e.g.,* "*Need more people sharing / Documenting and helping resolve problems.*").

Interestingly, one respondent drew attention to the fact that the benefit ***multiple environments*** can also be a problem, for instance "*If build fails it may be hard to debug it, especially if the problem doesn't occur in the local environment*". Additional problems that were highlighted in our research are:

1. ***monetary costs*** (*e.g.,* "*Hosted services like Travis CI are not free if you want to use them at a bigger scale or you need to maintain by yourself*");
2. ***flaky tests*** (*e.g.,* "*Tests can sometimes be flaky, which is frustrating*"), and;
3. ***additional effort*** required to, for instance, "*making sure builds pass*".

Finally, 10 respondents mentioned that they have never experienced any problems with Continuous Integration systems.

9. IMPLICATIONS

This work has implications to different kinds of stakeholders. Here we discuss some of them.

Researchers can propose visualization techniques that could improve the understanding of the state of the builds. Currently, one can only see if the build has passed or not. However, when several jobs are performed in the same build, software developers have a hard time to understand the outcome of the build and, as a consequence, in case of failure, how to fix it. Also, some respondents said that some jobs are more important than others, e.g., because of rarely used production environments. Such visualizations that can highlight jobs that are more important to developers. Moreover, since flaky tests were commonly observed by our respondents, researchers can also propose techniques to identify and mitigate flaky tests. Some respondents also mentioned a lack of quality in test code. Researchers can then propose novel approaches to generate test cases based on partial code (*e.g.,* the diff file).

Despite the well-known benefits (*e.g.,* CI catches regressions and save software development time), we showed additional reasons that might motivate software developers that still do not leverage CI practices. For instance, some respondents mentioned that CI systems improved the

credibility to their projects. This is particularly relevant to open source projects that depend on volunteers to sustain the long term evolution of a project. CI might also increase the likelihood of newcomers stopping and giving a chance to the project. Software developers also mentioned that they have a better understanding of the changes, which in turn improves their communication. Although Beller and colleagues [7] provide an in-depth discussion of build breakage reasons, in this paper we enrich the discussion regarding why build breakage happens, and that we should not always consider this a signal of a poor contribution. Actually, many reasons might cause a build failure and developers should not be afraid of contributing a patch because it might eventually lead to a breakage. This is particularly relevant to newcomers who are starting to contribute to a software project.

In our study, we perceived that there is a lack of testing culture, which might motivate educators to place additional care in providing testing courses. Moreover, we perceived that Continuous Integration is a very important skill, since all of the respondents have some degree of familiarity with CI systems. However, one-fourth of the respondents have never configured a single project to use Continuous Integration. Educators can take advantage of these findings and introduce Continuous Integration practices to the classroom. As a running example, educators can set up class projects and assignments to be automatically tested and graded using a CI system. Due to the disagreements we found in our responses, Educators can also stress the difference between jobs and builds. Some respondents also highlighted the importance of writing good tests. Educators can take advantage of this fact and not only introduce software testing in introductory programming courses, but also require students to write their test cases in different courses.

We found that developers face difficulties when setting up and maintaining the configuration files. Tool builders can also propose new tools to check the syntax and semantics of CI configuration files.

10. LIMITATIONS

As any empirical study, our work has many limitations. First, our conclusions can only be drawn from our limited list of participants. Although our participants have diverse backgrounds (Section 2), they also share some particularities: they have caused at least one Continuous Integration build to break in one software project hosted on GitHub. Moreover, since our subjects are open source contributors, our findings might not be easily translated to developers used to working at companies and other organizations producing non-open source software. Similarly, this work did not address the perceptions of other participants in other social coding websites. Therefore, it is still unclear whether the social features introduced by GitHub (e.g., the transparency of the build status) has any implication on the perception of a broken build.

Second, most of the studied systems are open source software; we did not study the build behavior in other kinds of projects (e.g., proprietary software projects). The findings of our work are also limited to the questions asked. When analyzing the answers of the questions, we realized that additional questions could have been asked (e.g., how do you fix broken jobs in a platform you do not have access to?). However, after the initial deployment of the survey, we were unable to change our survey design. Still, it is unclear how the findings can be useful in other CI platforms (such as Hudson or CruiseControl). With out methodology, we expect that other researchers can conduct similar work when relevant.

Finally, regarding the ethical implications of email invitations, we believe that the risks or discomforts are minimal. We did not share the email address of any developer. We took care to not invite the same developer more than once. The developers also have the freedom to decide whether to participate or not in the survey, either by ignoring our email or not answering the questionnaire. For those that answered, the responses will remain anonymous. No one will be able to identify a developer or their answers, and no one will know whether or not a developer participated in the study. Moreover, since the invited developers are not aware of how many developers were invited, and since a developer was invited just once, they are not aware of any potential problems related to mass-sending of emails.

11. RELATED WORK

Although there is a plethora of studies focusing on software testing techniques (*e.g.,* [5, 6, 37, 39]) and build systems (*e.g.,* [2, 23, 29]), Continuous Integration systems are far less studied, and the perception of CI users is still a topic that deserves further investigation.

Vasilescu and colleagues [35] performed a quantitative study of over 200 active GitHub projects. They restricted their search to Java, Ruby, and Python projects. Among the findings, they found that 92% of the selected projects have configured to use Travis CI, although 45% of them have no associated builds recorded in the Travis CI database. They also found that direct contributions (pushed commits) are more frequent than pull requests. In contrast, our work is qualitative by nature. We also search for builds involved in several other programming languages. The work of Vasilescu *et al.* [36] analyzed historical data of GitHub projects to see the effects of Continuous Integration usage. They found that Continuous Integration helped to increase the number of accepted pull requests from core developers, and to reduce the quantity of rejected from non-core developers, without affecting code quality. While Continuous Integration can help a reviewer to make decisions regarding pull requests faster, our study provide evidence that it is important not to over-rely on its results.

Beller and colleagues [7] performed a large-scale analysis of Continuous Integration builds on open source projects that used Travis CI, in order to determine "how central testing really is in Continuous Integration". They showed that although failing tests are the main reason of why builds fail in Continuous Integration systems, open source projects that consistently ignore the tests results are rather rare. In our work, we also observed the importance of writing good test cases (inadequate testing is the most common technical problem that lead to build failures). Moreover, different to their work, which is artifact-focused (*i.e.,* based on data and meta-data acquired from the repositories), our work is developer-focused (*i.e.,* based on the perception of software developers that have experience with Continuous Integration tools). In another up study, Beller and colleagues provided a dataset comprehending 2,640,825 Travis CI builds from more than 1,000 open source projects [8]. This data set is useful for encouraging and empowering software engineering researchers to conduct Continuous Integration studies. As an example, Rebouças and colleagues leverage this dataset to better understand whether contributors that are not very active in an open source project are more likely to break the build [28]. Preliminary results suggest that the rate of build success of barely active contributors is similar to active contributors in 85% of the analyzed projects. The work of Kim *et al.* [19] is aimed at making Continuous Integration more cost-effective, which is done by selecting only test cases that are potentially affected by changes. Although orthogonal to our work, the authors did not evaluate the perception of software developers using the proposed technique, neither whether the technique lead to fewer broken builds.

Moreover, the work of Hilton *et al.* [18] is aimed at understanding how software developers use Continuous Integration tools, for instance, why do some projects do not use CI or how CI configuration evolve. By analyzing builds and performing a survey, they collected evidence showing that Continuous Integration reduces the time between releases and that it is widely adopted in popular projects. Our work differs to their work in particular due to our focus on broken builds; although their work provides some initial discussion about build breakage, they did not provide an in-depth investigation in this regard. To the best of our knowledge, the work of Hilton and colleagues [17] is the closed to our work. Through semi-structured interviews and two surveys, the authors studied why do developers use CI, the barriers they face, and the benefits they experienced. The authors have identified similar findings that we reported in this study, such as the possibility to catch bugs earlier (reported as a reason for adoption in Section 5.1) and the barrier to setting up a CI service (reported as a problem in Section 8.2). However, we believe that our work can be seen as complementary to Hilton and colleagues [17] work, since we enlightened unique characteristics that were so far unknown (such as the overconfidence on CI). Moreover, while they have a broader population (i.e., software developers that use social networks), our work is focused on open source developers that have already broken a CI build.

In a previous effort, we reported part of the findings of this study [25]. Nevertheless, this previous study mostly focused on the reasons for build breakage and on the benefits and problems of these systems. In the current submission, we greatly expand the results, not only covering the respondents background and details about how developers perceive builds and jobs, but also improving the overall discussion.

12. CONCLUSIONS

Continuous Integration systems are gaining adoption among software developers. However, there is missing a collection of experience that developers have with CI systems. In this work we tried to better understand the perceptions of open source developers about the fundamental concepts related to Continuous Integration systems, the reasons for build breakage, and the benefits and problems associated with these systems. In this study we performed an user survey with 158 Continuous Integration users to shed the light on these questions. Through a qualitative research analysis, we produce a list of findings about the Continuous Integration systems, some which are not always obvious, such as the (over) confidence in Continuous Integration systems, a lack of testing culture, and the promoted fast development cycle.

For *future work*, we plan to conduct semi-structured interviews with a selected group of Continuous Integration users, in order to refute or gain further confidence in our findings. Still, we plan to triangulate our findings with additional data sources. For instance, we plan to leverage databases that record build breakage and try to cross-validate our findings, *e.g.,* comparing the technical reasons that lead to a build break.

ACKNOWLEDGEMENTS

REFERENCES

1. Abramovici, M., Parikh, P.S.: WARNING: 100% fault coverage may be misleading!! In: Proceedings International Test Conference 1992, pp. 662– (1992). DOI 10.1109/TEST.1992.527887
2. Adams, B., Tromp, H., de Schutter, K., de Meuter, W.: Design recovery and maintenance of build systems. In: 2007 IEEE International Conference on Software Maintenance, pp. 114–123 (2007). DOI 10.1109/ICSM.2007.4362624
3. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change (2Nd Edition). Addison-Wesley Professional (2004)
4. Beller, M.: Toward an empirical theory of feedback-driven development. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE '18, pp. 503–505 (2018)
5. Beller, M., Gousios, G., Panichella, A., Zaidman, A.: When, how, and why developers (do not) test in their ides. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 179–190 (2015)
6. Beller, M., Gousios, G., Zaidman, A.: How (much) do developers test? In: Proceedings of the 37th International Conference on Software Engineering - Volume 2, ICSE '15, pp. 559–562 (2015)
7. Beller, M., Gousios, G., Zaidman, A.: 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 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 356–367 (2017)
8. Beller, M., Gousios, G., Zaidman, A.: Travistorrent: synthesizing travis CI and github for full-stack research on continuous integration. In: Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 447–450 (2017)
9. Borges, H., Hora, A., Valente, M.T.: Understanding the factors that impact the popularity of GitHub repositories. In: 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 334–344 (2016)
10. Briand, L., Pfahl, D.: Using simulation for assessing the real impact of test coverage on defect coverage. In: Proceedings of the 10th International Symposium on Software Reliability Engineering, ISSRE '99, pp. 148– (1999)
11. Dijkstra, E.W.: Structured programming. chap. Chapter I: Notes on Structured Programming, pp. 1–82 (1972)

12. Duvall, P., Matyas, S., Glover, A.: Continuous Integration: Improving Software Quality and Reducing Risk. A Martin Fowler signature book. Addison-Wesley (2007)
13. Fowler, M.: Continuous integration (original version) (2000). URL http://web.archive.org/web/20180626090647/https://www.martinfowler.com/articles/originalContinuousIntegration.html
14. Fowler, M.: Continuous integration: Integrate at least daily (2018). URL http://web.archive.org/web/20180626090647/https://www.thoughtworks.com/continuous-integration
15. Gousios, G., Storey, M.A., Bacchelli, A.: Work practices and challenges in pull-based development: The contributor's perspective. In: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, pp. 285–296 (2016)
16. Gousios, G., Zaidman, A., Storey, M.D., van Deursen, A.: Work practices and challenges in pull-based development: The integrator's perspective. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, pp. 358–368 (2015)
17. Hilton, M., Nelson, N., Tunnell, T., Marinov, D., Dig, D.: 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, Paderborn, Germany, September 4-8, 2017, pp. 197–207 (2017)
18. Hilton, M., Tunnell, T., Huang, K., Marinov, D., Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pp. 426–437 (2016)
19. Kim, S., Park, S., Yun, J., Lee, Y.: Automated continuous integration of component-based software: An industrial experience. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE '08, pp. 423–426 (2008)
20. Kitchenham, B.A., Pfleeger, S.L., Pickard, L.M., Jones, P.W., Hoaglin, D.C., Emam, K.E., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. IEEE Trans. Softw. Eng. **28**(8), 721–734 (2002)
21. Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 643–653 (2014)
22. Luz, W., Pinto, G., Bonifácio, R.: Building a collaborative culture: A grounded theory of well succeeded devOps adoption in practice. In: 2018 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM, pp. 1–10 (2018)
23. Mcintosh, S., Adams, B., Hassan, A.E.: The evolution of java build systems. Empirical Softw. Engg. **17**(4-5), 578–608 (2012)
24. Nicolai, J.: Github welcomes all CI tools (2000). URL http://web.archive.org/web/20180626090647/https://blog.github.com/2017-11-07-github-welcomes-all-ci-tools/
25. Pinto, G., Rebouças, M., Castor, F.: Inadequate testing, time pressure, and (over) confidence: A tale of continuous integration users. In: 10th IEEE/ACM International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE@ICSE 2017, Buenos Aires, Argentina, May 23, 2017, pp. 74–77 (2017)
26. Pinto, G., Steinmacher, I., Gerosa, M.A.: More common than you think: An in-depth study of casual contributors. In: IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER, pp. 112–123 (2016)
27. Ray, B., Posnett, D., Filkov, V., Devanbu, P.: A large scale study of programming languages and code quality in github. In: Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, pp. 155–165 (2014)
28. Rebouças, M., Santos, R.O., Pinto, G., Castor, F.: How does contributors' involvement influence the build status of an open-source software project? In: Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 475–478 (2017)
29. Seo, H., Sadowski, C., Elbaum, S., Aftandilian, E., Bowdidge, R.: Programmers' build errors: A case study (at google). In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 724–734 (2014)
30. Smith, E., Loftin, R., Murphy-Hill, E., Bird, C., Zimmermann, T.: Improving developer participation rates in surveys. In: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), pp. 89–92 (2013). DOI 10.1109/CHASE.2013.6614738
31. Steinmacher, I., Wiese, I.S., Conte, T., Gerosa, M.A., Redmiles, D.F.: The hard life of open source software project newcomers. In: Proceedings of the 7th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE, pp. 72–78 (2014)
32. Strauss, A., Corbin, J.M.: Basics of Qualitative Research : Techniques and Procedures for Developing Grounded Theory, 3rd edn. SAGE Publications (2007)
33. Tómasdóttir, K.F., Aniche, M.F., van Deursen, A.: Why and how javascript developers use linters. In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017, pp. 578–589 (2017)
34. Trockman, A., Zhou, S., Kästner, C., Vasilescu, B.: Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem. In: International Conference on Software Engineering, ICSE. ACM (2018). DOI https://doi.org/10.1145/3180155.3180209
35. Vasilescu, B., van Schuylenburg, S., Wulms, J., Serebrenik, A., van den Brand, M.G.J.: Continuous integration in a social-coding world: Empirical evidence from github. In: 30th IEEE International Conference on Software Maintenance and Evolution, pp. 401–405 (2014)
36. Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., Filkov, V.: Quality and productivity outcomes relating to continuous integration in github. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, pp. 805–816 (2015)

37. Williams, T.W., Mercer, M.R., Mucha, J.P., Kapur, R.: Code coverage, what does it mean in terms of quality? In: Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity, pp. 420–424 (2001). DOI 10.1109/RAMS.2001.902502

38. Zhao, Y., Serebrenik, A., Zhou, Y., Filkov, V., Vasilescu, B.: The impact of continuous integration on other software development practices: A large-scale empirical study. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pp. 60–71 (2017)

39. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Comput. Surv. **29**(4), 366–427 (1997)