Oops, My Tests Broke the Build: An Explorative Analysis of Travis CI with GitHub

Moritz Beller
Delft University of Technology,
The Netherlands
m.m.beller@tudelft.nl

Georgios Gousios
Delft University of Technology,
The Netherlands
g.gousios@tudelft.nl

Andy Zaidman
Delft University of Technology,
The Netherlands
a.e.zaidman@tudelft.nl

ABSTRACT

Continuous Integration (CI) has become a best practice of modern software development. Yet, at present, we have a shortfall of insight into the testing practices that are common in CI-based software development. In particular, we seek quantifiable evidence on how central testing is to the CI process, how strongly the project language influences testing, whether different integration environments are valuable and if testing on the CI can serve as a surrogate to local testing in the IDE. In an analysis of 2,640,825 Java and Ruby builds on TRAVIS CI, we find that testing is the single most important reason why builds fail. Moreover, the programming language has a strong influence on both the number of executed tests, their run time, and proneness to fail. The use of multiple integration environments leads to 10% more failures being caught at build time. However, testing on TRAVIS CI does not seem an adequate surrogate for running tests locally in the IDE. To further research on TRAVIS CI with GITHUB, we introduce TRAVISTORRENT.

Categories and Subject Descriptors

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

General Terms

Experimentation, Measurement, Theory, Verification

Keywords

Continuous Integration, Travis CI, GitHub, TRAVISTORRENT, GHTORRENT

1. INTRODUCTION

Continuous Integration (CI) is the software engineering practice in which developers not only integrate their work into a shared mainline frequently, but also verify the quality of their contributions continuously. CI facilitates this through an automated build process that typically includes (developer) tests [1] and various static analysis tools that can be run in different integration environments [2]. Originally described by Microsoft [3] and proposed as one of the twelve Extreme Programming (XP) practices in 1997 [4], CI has become a universal industry and Open-Source Software (OSS) best practice, often used outside the context of XP [5, 6].

A full CI build comprises 1) a traditional build and compile phase, 2) a phase in which automated static analysis tools (ASATs) such as FINDBUGS and JSHINT are run [7, 8], and 3) a testing phase, in which unit, integration, and system tests are run. If any of these three phases fails, the whole CI build is typically aborted and regarded as broken [9]. Researchers have explored the compile and ASAT phase of CI [7, 10]; yet, we still lack a quantitative empirical investigation of the testing phase to gain a holistic understanding of the CI process. This is surprising, as testing stands central in CI [2] and a better understanding is the first step to further improve both the CI process and the build tools involved.

In this paper, we study CI-based testing in the context of TRAVIS CI, an OSS CI as-a-service platform that tightly integrates with GITHUB. While there has been research on aspects of TRAVIS CI [11, 12], we lack an overarching explorative study to quantitatively explore the CI domain for testing from the ground up. Moreover, as accessing data from TRAVIS CI and overlaying it with GITHUB data involves difficult technicalities, researchers would profit from making this promising data source more accessible.

Our explorative research into CI is steered by five concrete propositions inspired from and raised by previous research:

P1. The use of CI is a widespread best practice. CI has become an integral quality assurance practice [13]. But just how widespread is its use in OSS projects? One study on TRAVIS CI found an adoption rate of 45 to 90% [11]. This seems surprisingly high given it was measured in 2013, when TRAVIS CI was still very new, and also based on only a small subset of projects.

P2. Testing is central to CI. Two studies on the impact of compilation problems and ASATs at Google found that missing dependencies are the most important reason for builds to break [7, 10]. However, these studies have not considered the testing phase of CI. To gain a complete picture of CI, we need to measure the importance and impact of the testing phase in a CI build process.

P3. Testing on the CI is language-dependent. While CI is a general purpose practice for software development projects, the programming languages used in CI have been shown to differ, e.g. in terms of programming effort [14]. As such, CI observations for one language might not generalize to other languages. A cross-language comparison might unveil which testing practices of a certain language community and culture might benefit more from CI, in terms of shorter run time or fewer broken builds.

P4. Test Integration in different environments is valuable [13, Chapter 4]. Building and integrating in different environments multiple times is time- and resource-intensive. Consequently, it should deliver additional value over a regular one-environment in-

tegration strategy. We currently lack data to support this claim.

P5. Testing on the CI is a surrogate for testing in the IDE for getting quick feedback. One of the core ideas of developer testing is to provide quick feedback to developers [15, 16]. Yet, a recent study on how 416 software developers test in their Integrated Development Environments (IDEs) [17] could not explain the "testing paradox:" developers spent a substantial 25% of their time working on tests, but rarely executed them in their IDE. We received anecdotal evidence that, instead, developers might *offload* running tests to the CI. However, it is unclear whether the CI environment is indeed a suitable replacement for running tests locally. In particular, while Fowler claims that CI provides quick feedback [2], it typically does not allow developers to execute specific tests. It also introduces other scheduling-related latencies, the magnitude of which is not known yet.

To guide our investigation of propositions P1-P5, we derived a set of research questions, presented below along with the propositions they address:

RQ1 How common is the use of TRAVIS CI on GitHub? (*P1*)

RQ2 How central is testing to CI? (*P2, P5*)
RQ2.1 How many tests are executed per build?
RQ2.2 How long does it take to execute tests on the CI?
RQ2.3 How much latency does CI introduce in test feedback?

RQ3 How do tests influence the build result? (*P3, P4, P5*)
RQ3.1 How often do tests fail?
RQ3.2 How often do tests break the build?
RQ3.3 Are tests a decisive part of CI?
RQ3.4 Does integration in different environments lead to different test results?

Developers need to be aware of the answers to these questions to know and assess the *state of the art of how CI is done in the OSS community.* Especially maintainers of and newcomers to CI, whether they join an established project or plan to introduce CI, will benefit from knowing what they can expect from it ("How many builds are going to fail and require additional work?", "Can I use CI to execute my tests instead of locally executing them?") and how most other projects are using it ("How common is CI?," "Can my project use it?"). It is therefore important to make our results known to the CI community.[1]

2. BACKGROUND

In this section, we outline related CI work and build tools. We also give an overview and technical description of TRAVIS CI.

2.1 Related Work

Introduced as one of the twelve best practices of extreme programming in 2000 [4], CI is a relatively new trend in software engineering. In their 2014 systematic review, Ståhl and Bosch provided the most recent overview over CI practices and how they differ in various settings of industrial software development [18]. Of particular interest to us is their analysis of what is considered a failure in a CI build. The most commonly observed stance is that if any test fails during the build, then the build as a whole is considered failed (e.g., [9, 19]). Ståhl and Bosch found that build failures due to test failures are sometimes accepted, however: "[I]t is fine to permit acceptance tests to break over the course of the iteration as long as the team ensures that the tests are all passing prior to the end of the iteration" [20].

A case study at Google investigated a large corpus of builds in the statically typed languages C and Java [10], uncovering several patterns of build errors. While the study is similar in nature, it focused on static compilation problems and spared out the dynamic execution part of CI, namely testing. Moreover, it is unknown whether their findings generalize to a larger set of OSS.

Vasilescu et al. examined whether a sample of 223 GITHUB projects in Java, Python, and Ruby used TRAVIS CI [11]. While more than 90% had a TRAVIS CI configuration, only half of the projects actually used it. In follow-up research, Vasilescu et al. found that CI, such as provided through TRAVIS CI, significantly improves their definition of project teams' productivity, without adversely affecting code quality [12].

Pinto et al. researched how test suites evolve [21]. This work is different in that we observe real test executions as they were run in-vivo on the CI server here, while Pinto et al. performed their own post-mortem, in-vitro analysis. Their approach offers a finer control over the produced log data, yet it bears the risk of skewing the original execution results, for example because a build dependency is not available anymore [21].

Pham et al. investigated the testing culture on social coding sites. In particular, they note that to nurture a project's testing culture, the testing infrastructure should be easy to set up. Their interviews furthermore lead to the observation that TRAVIS CI "arranges for low barriers and easy communication of testing culture" [6]. By analyzing build logs, we hope to be able to see how many projects make use of this infrastructure.

With TRAVIS CI, a public and free CI service that integrates tightly with GITHUB, we have the chance to observe how CI happens in the wild on a large basis of influential OSS projects.

Similar to TRAVIS CI, but typically setup and maintained by the project themselves, are a number of other CI servers like CruiseControl, TeamCity, Jenkins, Hudson, and Bamboo [22].

2.2 Travis CI

In this section, we provide an overview over TRAVIS CI.

Overview. TRAVIS CI is an open-source, distributed build service that, through a tight integration with GitHub, allows projects to build and run their CI procedures without having to maintain their own infrastructure [23]. TRAVIS CI started in 2010 as an open-source project and turned into a company in 2012. In August 2015, it supports 26 different programming languages including Java, C(++), Scala, Python, R, and Visual Basic.[2] Apart from the community edition, free to use for OSS, TRAVIS CI also hosts a paid service that provides non-public builds for private GITHUB repositories. This edition features a faster build environment.[3]

User View. Figure 1 showcases TRAVIS CI's main User Interface for build number 518 in the OSS project TESTROOTS/WATCHDOG [24, 25].[4] At marker ①, we see the project name and build status of its master branch. On the right hand side ②, TRAVIS CI shows which GIT commit triggered the build, its build status ("passed") along with information such as the overall build time (7 minutes, 51 seconds). The description at ③ indicates that the build was triggered by a pull request. Through link ④, we can retrieve the full history of all builds performed for this project. Under ⑤, we can see a lightly parsed and colorized dump of the log file created when executing the build. By clicking ⑥, developers can trigger a re-execution of a build.

Build Setup. Whenever a commit to any branch on a TRAVIS

[1] A summary of the paper appeared in the TRAVIS CI blog. https://blog.travis-ci.com/2016-07-28-what-we-learned-from-analyzing-2-million-travis-builds/

[2] http://docs.travis-ci.com/user/getting-started/

[3] https://travis-ci.com/plans

[4] https://github.com/TestRoots/watchdog

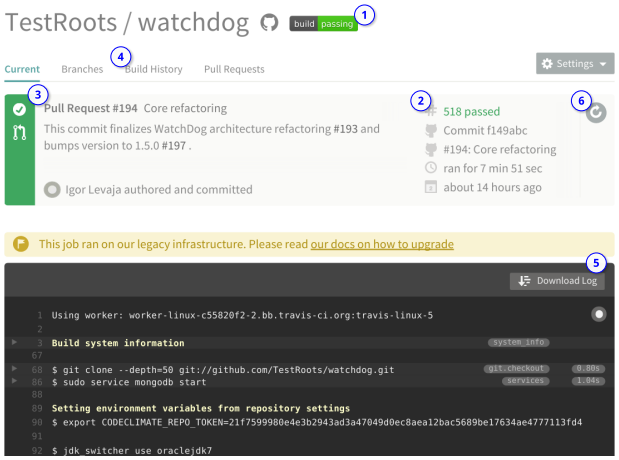

Figure 1: TRAVIS CI's UI for an OSS project (WATCHDOG, [24]).

Figure 2: Integration of TRAVIS CI on GitHub, displaying build results in a pull request for multiple commits.

CI-enabled GITHUB repository is pushed, the latest commit of said branch or pull request is automatically received by TRAVIS CI through a GITHUB web hook and subsequently built. The result of this build is then displayed on GitHub, like the excerpt of the pull request in Figure 2: The developer pushed the first two commits separately, thereby triggering two passing TRAVIS CI builds ①. He then transferred the third and fourth commit in one single push, leading to TRAVIS CI only building the latest ②. If a build is running while a newer commit is pushed to the same branch, TRAVIS CI immediately cancels the execution of the current build and starts building the latest relevant commit. This seamless integration into projects' workflow caters for the popular pull request model [26] and is supposedly key to TRAVIS CI's popularity among GITHUB projects.

TRAVIS CI users configure their build preferences through a top-level file in their repository. This defines the language, the default environments and possible deviations from the default build steps that TRAVIS CI provisions for building the project. TRAVIS CI currently only provides single-language builds, but it does support building in multiple environments, e.g., different versions of Java. For each defined build environment, TRAVIS CI launches one job that performs the actual build work in this environment. If one of these jobs breaks, i.e. the build execution in one build environment exits with a non-successful status, TRAVIS CI marks the whole build as broken. TRAVIS CI instills a maximum job runtime of 50 minutes for OSS, after which it cancels the active job.

Build Life-cycle. Each job on TRAVIS CI goes through the build steps depicted in the state machine in Figure 3: A CI build always starts with the three infrastructure provisioning phases BEFORE_ INSTALL, INSTALL and BEFORE_SCRIPT. In CI phase ①, the TRAVIS CI job is initialized, either as a legacy virtual machine like in step ⑤ in Figure 1 or as a new DOCKER container [27], the GIT repository is cloned, additional software packages are installed, and a system update is performed. If a problem occurs during these phases, the job is marked as *errored* and immediately aborted. The

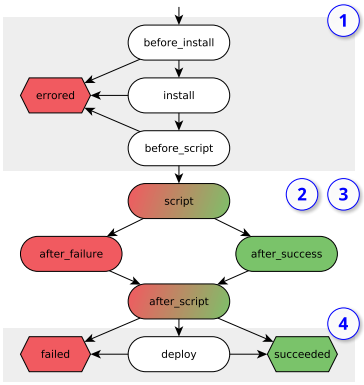

Figure 3: TRAVIS CI build phases as a state machine.

SCRIPT phase actualizes the build, for example for a Java MAVEN project, TRAVIS CI calls `mvn -B` to build and test the application, and for Ruby, it calls `rake` per default, CI phases ASAT and test runs, ② and ③. The build phase can either *succeed* or *fail*, denoted in a Unix-fashion non-zero return value from the SCRIPT phase. The DEPLOY phase ④ is optional and does not influence the build result.

TRAVIS CI pipes the console output from the different build phases into the build log of the job, separated as so-called folds. For example, the `git.checkout` is part of the BEFORE_INSTALL phase in Figure 1. The output generated in the SCRIPT phase contains the typical console output of build tools: build status, execution time, possible compilation problems, test executions, and failures. Naturally, the format of this output depends on the actual build and test framework used.

Build Status. TRAVIS CI features a *canceled* build status that can occur in any phase and is triggered from the outside. We call an *errored* or *failed* build more generally *broken*, opposed to a *succeeded* build.

REST API. Next to its normal user interface in Figure 1, TRAVIS CI provides an unrestricted RESTful Web-API,[5] using which data from all publicly built OSS repositories can be queried. The API allows us to conveniently access build-related information to perform our deeper build analysis.

3. RESEARCH SETUP

In this section, we give a high-level overview of our research design and describe our research methodology in detail.

3.1 Study Design

The main focus of this study is to evaluate how testing works in the context of CI. We performed a purely quantitative study to address our propositions, combining multiple data sources and RQs. We use the GHTORRENT database [28] as a source of projects to examine and apply filtering to select the most appropriate ones. The results for RQ1 lead us to the projects we would analyze in RQs 2 and 3.

These remaining research questions require a deep analysis of the project source code, process and dependency status at the job level. Moreover, as we needed to examine test tool outputs, we restricted our project search space to Ruby and Java. Both languages enjoy wide popularity among developer and have a strong testing tradition, evidenced by the plethora of available automated testing tools. Using the projects selected in the previous step as

[5] `https://api.travis-ci.org/`

a starting point, we filtered out those that are not written in Ruby or Java and are not integrated with TRAVIS CI. Then, we extract and analyze build information from TRAVIS CI build logs and the GHTORRENT database, combining both data sources in the newly implemented TRAVISTORRENT.

3.2 Tools

In this section, we detail the tools we used to carry out our study. Our data extraction and analysis pipeline is written in Ruby and R. For replication purposes and to stimulate further research, we created TRAVISTORRENT [29], which disseminates our tools and data set publicly.[6]

TravisPoker. To find out which and how many projects on GitHub use TRAVIS CI, we implemented TRAVISPOKER. This fast and lightweight application takes a GITHUB project name as input (for example, RAILS/RAILS), and finds out if and how many TRAVIS CI builds were executed for this project.

TravisHarvester. We implemented TRAVISHARVESTER to aggregate detailed information about a project's TRAVIS CI build history. It takes as input a GITHUB project name and gathers general statistics on each build in the project's history in a CSV file. Associated with each build entry in the CSV are the SHA1 hash of the GIT commit, the branch and (if applicable) pull request on which the build was executed, the overall build status, the duration and starting time and the sub jobs that TRAVIS CI executed for the different specified environments (at least one job, possibly many for each build). TRAVISHARVESTER downloads the build logs for each build for all jobs and stores them alongside the CSV file.

While both TRAVISPOKER and TRAVISHARVESTER utilize TRAVIS CI's Ruby client for querying the API,[7] we cannot use its job log retrieval function (`Job:log`) due to a memory leak[8] and because it does not retrieve all build logs. We circumvented these problems by also querying the Amazon AWS server that archives build logs.[9]

To speed up the process of retrieving thousands of log files for each project, we parallelize our starter scripts for TRAVIS HARVESTER with GNU PARALLEL [30].

Build Linearization And GITHUB Synthesis. To assess the status of the project at the moment each build was triggered, we extract information from two sources: the project's GIT repository and its corresponding entry in the GHTORRENT database. During this step we also perform the build linearization described in Section 3.3.

BUILDLOG ANALYZER. BUILDLOG ANALYZER is a framework that supports the general-purpose analysis of TRAVIS CI build logs and provides dedicated Java and Ruby build analyzers that parse build logs in both languages and search for output traces of common testing frameworks.

The language-agnostic BUILDLOG ANALYZER reads-in a build log, splits it into the different build phases (folds, see Section 2.2), and analyzes the build status and runtime of each phase. The fold for the SCRIPT phase contains the actual build and continuous testing results. The BUILDLOG ANALYZER dispatches the automatically determined sub-BUILDLOG ANALYZER for further examination of the build phase.

For Java, we support the three popular build tools MAVEN, GRADLE, and ANT [31]. In Java, it is standard procedure to use JUNIT as the test runner, even if the tests themselves employ other testing frameworks, such as POWERMOCK or MOCKITO. Moreover, we also support TESTNG, the second most popular testing framework for Java. Running the tests of an otherwise unchanged project through MAVEN, GRADLE and ANT leads to different, incompatible build logs, with MAVEN being the most verbose and GRADLE the least. Hence, we need three different parsers to support the large ecosystem of popular Java build tools. As a consequence, the amount of information we can extract from a build log varies per build technology used. Moreover, some build tools give users the option to modify their console output, albeit rarely used in practice.

Example 1: Standard output from MAVEN regarding tests

```
1
2  T E S T S
3
4  Running org.testroots.watchdog.ClientVersionCheckerTest
5  Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time
     elapsed: 0.04 sec
6
7  Results :
8
9  Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
10
11 [INFO] All tests passed!
```

Example 1 shows an excerpt of one test execution from the TESTROOTS/WATCHDOG project. In the output, we can see the executed test classes (line 4), and how many tests passed, failed, errored and were skipped. We also get the test execution time (line 5). Moreover, MAVEN prints an overall result summary (line 9) that the BUILDLOG ANALYZER uses to triage its prior findings. Line 11 shows the overall test execution result. Our BUILDLOG ANALYZER gathers all this information and creates, for each invoked project, a CSV table with all build and test results for each job built. We then aggregate this information with information from the build status analyzer step by joining their output. TRAVISTORRENT provides easy access to this data.

Example 2 shows the equivalent GRADLE output. The silent GRADLE becomes more verbose when a test fails, providing us with similar information to Example 1.

Example 2: Standard output from GRADLE regarding tests

```
1 :test
```

By contrast, in Ruby, the test framework is responsible for the console output: it is no different to invoke RSPEC through RAKE than through BUNDLER, the two predominant Ruby build tools [31]. For Ruby, we support the prevalent TEST::UNIT and all its off springs, like MINITEST. Moreover, we capture behavior driven tests via RSPEC and CUCUMBER support [32].

3.3 Build Linearization and Mapping to Git

If we want to answer questions such as "how much latency does CI introduce" (RQ2.3), we need to make a connection between the builds performed on TRAVIS CI and the repository which contains the commits that triggered the build. We call this build linearization and commit mapping, as we need to interpret the builds on TRAVIS CI as a directed graph and establish a child-parent relationship based on the GIT commits that triggered their execution.

Figure 4 exemplifies how a typical GITHUB project uses TRAVIS CI in a UML-inspired syntax [33], where vertices go from child to parent, dotted lines mean ambiguity and ◆ denotes association. In the upper part ①, we see the TRAVIS CI builds (§1-§9), where §1 is older than §2 and so on, which are either passed (§1-§6, §9), canceled (§7), or broken (§8), see Section 2.2. In the lower part ②, we see the corresponding GIT repository hosted on GITHUB with its individual commits (#A-#H). Commits #D1-#D3 live in a

[6] http://travistorrent.testroots.org

[7] https://github.com/travis-ci/travis.rb

[8] https://github.com/travis-ci/travis.rb/issues/310

[9] http://s3.amazonaws.com/archive.travis-ci.org

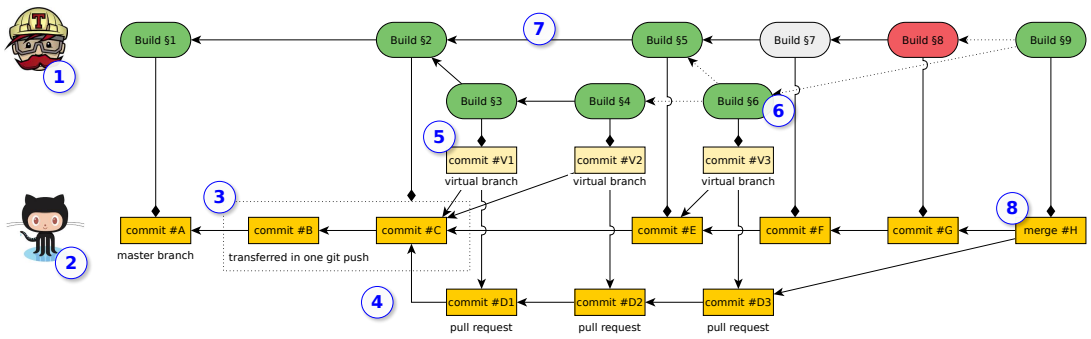

Figure 4: Exemplary scenario of how to match commits from a GITHUB repository to their corresponding TRAVIS CI builds.

pull request, and not on the master branch, traditionally the main development line in GIT.

a) Build §1 showcases a standard situation, in which the build passed and the commit id stored with the build leads to the correct commit #A that triggered build §1. However, there are a number of more complex situations.

b) If multiple commits are transferred in one `git push` ③, only the latest of those commits is built (§2). In order to get a precise representation of the changes that lead to this build result, we have to aggregate commits #B and #C.

c) It is a central function of TRAVIS CI to support branches or pull requests ④, such as commit #D1. When resolving builds to commits, we know from the API that §3 is a pull request build (see Section 2.2). Its associated commit points us to a virtual integration commit #V1 that is not part of the normal repository, but automatically created as a remote on GITHUB ⑤. This commit #V1 has two parents: 1) the latest commit in the pull request (#D1), and 2) the current head of the branch the pull request is filed against, the latest commit on the master branch, #C. Similarly, when resolving the parent of §4, we encounter a #V2, resolve it to #D2 and the already known #C. We also know that its direct parent, #D1, is branched-off from #C. Hence, we know that any changes from build result §4 to §3 were induced by commit #D2.

d) In the case of build §6 on the same pull request ⑥, its direct predecessor is unclear: we traverse from #V3 to both 1) commit #D2 in the pull request, which is known, and to 2) #E on the master branch, which is unknown and cannot be reached from any of our previous commits #D2, #D1, or #C. This is because there was an intermediate commit #E on the master branch in-between, and pull requests are always to be integrated onto the head commit of the branch they are filed against. In such a case, one build can have multiple parents, and it is undecidable whether the changes in #D3, #E or a combination of both lead to the build result §6.

e) Build §5 shows why a simple linearization of the build graph by its build number would fail: It would return §4 as its predecessor, when in reality, it is §2 ⑦. However, even on a single branch, there are limits to how far GIT'S complex commit relationship graph can be linearized and mapped to TRAVIS CI builds. For example, if a build is canceled (§7), we do not know about its real build status – it might have passed or broken. As such, for build §8, we cannot say whether the build failure resulted from changes in commit #F or #G.

f) Finally, when merging branches or pull requests ⑧, a similar situation as in c) occurs, in which one merge commit #H naturally has two predecessors.

For builds with situations *d)-f)*, we have not denoted a previous build in this paper, effectively excluding them from analyses where the previous build is needed. Other, more complicated solutions would be to use both, or randomly choose one.

3.4 Statistical Evaluation

When applying statistical tests in the remainder of this paper, we follow established principles [34]: we regard results as significant at a 95% confidence interval ($\alpha = 0.05$), i.e. if $p \leqslant \alpha$. All results of tests t_i in the remainder of this paper are statistically significant at this level, i.e. $\forall i : p(t_i) \leqslant \alpha$.

For each test t_i, we first perform a *Shapiro-Wilk Normality test* s_i [35]. Since all our distributions significantly deviate from a normal distribution according to Shapiro-Wilk ($\forall i : p(s_i) < 0.01 \leqslant \alpha$), we use non-parametric tests: for testing whether there is a significant statistical difference between two distributions, we use the non-parametric *Wilcoxon Rank Sum test.*

Regarding effect sizes, we report Vargha-Delaney's ($\hat{A}_{12}$) [36], a non-parametric effect size for ordinal values [37]. The $\hat{A}_{12}(A,B)$ measure has an intuitive interpretation: its ratio denotes how likely distribution A outperforms B.

4. RESULTS

In this section we report the results to our research questions.

4.1 RQ1: How common is the use of Travis CI on GitHub?

Before investigating the testing patterns on TRAVIS CI, we must first know 1) how many projects on GITHUB use TRAVIS CI, and 2) what characterizes them and their use of TRAVIS CI. In August 2015, we were in a good position to measure the TRAVIS CI adoption rate on a broad scale, as projects interested in using free CI had two years of adoption time to start to use TRAVIS CI (see Section 2.2). We conjecture that, if projects have a primary interest in CI, this was enough time to hear about and set up TRAVIS CI.

According to GHTORRENT, GITHUB hosted 17,313,330 active OSS repositories (including forks) in August, 2015. However, many of these 17 million projects are toy projects or duplicated (forks with no or tiny modifications). In our analysis, we are interested in state-of-the-art software systems that have a larger real-world user base. To retrieve a meaningful sample of projects from GITHUB, we follow established project selection instructions [38]: we selected all projects that are not forks themselves, and received more than 50 stars.

This filtering resulted in 58,032 projects. For each project, we extracted five GITHUB features from GHTORRENT: main project language $\in \{$C, C++, Java, Ruby, ...$\}$, number of watchers $\in [51;41,663]$, number of external contributors $\in [0;2,986]$, number of pull requests $\in [0;27,750]$, number of issues $\in [0;127,930]$ and

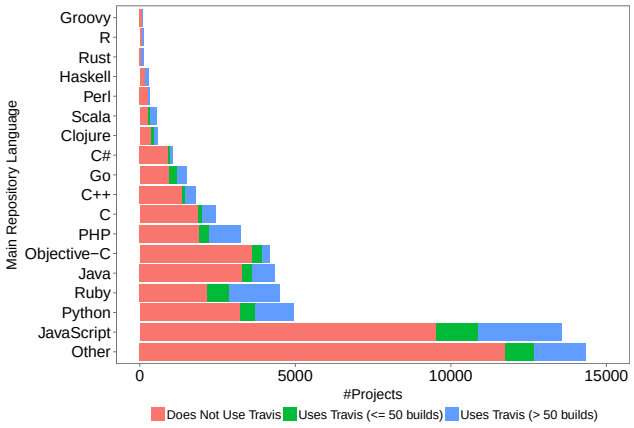

Figure 5: TRAVIS CI adoption per language.

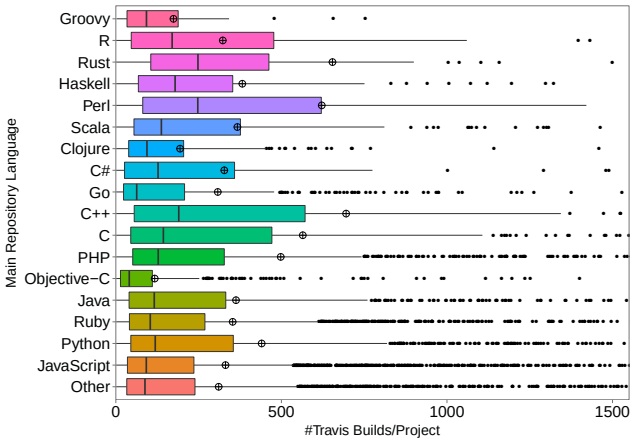

Figure 6: Horizontal boxplot of the #builds per project and language.

active years of the project $\in [0;45]$); using TRAVIS POKER we collected how many TRAVIS CI builds were executed. In total, we found that our 58,032 projects were written in 123 unique main repository languages. 16,159 projects used TRAVIS CI for at least one build, resulting in an overall TRAVIS CI usage rate of 27.8%. The majority of the 123 primary languages of our projects are not supported by TRAVIS CI (see Section 2.2). When reducing the main project language to the 26 languages supported by TRAVIS CI, we are left with 43,695 projects (75.3% of all projects). Out of these, 13,590 (31.1%) used TRAVIS CI for at least one build.

Figure 5 details these findings, showing the number of projects using TRAVIS CI aggregated per programming language. Inspired by Vasilescu et al., who found that many projects were configured for TRAVIS CI but did not really use it, we group projects into categories with 1) no, 2) a shorter ($\leqslant$ 50 builds), and 3) a longer ($>$ 50) TRAVIS CI history. If there is a smaller number of TRAVIS CI builds, this means that the project either recently started using TRAVIS CI, or that TRAVIS CI was quickly abandoned, or that the project was not active since introducing TRAVIS CI. Due to their short build history, we have to exclude such projects from our onward analyses: it is questionable whether these projects ever managed to get CI to work properly, and if so, there is no observable history of the projects using CI. We, however, are interested in how projects work and evolve with an active use of CI.

While 31.1% is a closer approximation of the real TRAVIS CI usage rate, Figure 5 hints at the fact that also projects whose main language is not supported, use TRAVIS CI, expressed as "Other".

In total, TRAVIS CI executed 5,996,820 builds on all 58,032 sampled projects. Figure 6 gives a per-language overview of the number of builds executed per each project, based on all 16,159 projects that had at least one build. Next to the standard boxplot features, the $\oplus$-sign marks the mean number of builds.

From our 13,590 projects in a TRAVIS CI-supported main language, we selected the ones which were best fit for an analysis of their testing patterns. We therefore ranked the languages according to their adoption of TRAVIS CI as depicted in Figure 5. We also requested that they be popular, modern languages [39], have a strong testing background, one language be dynamically and the other statically typed, and the number of available projects be similar. Furthermore, there should be a fairly standard process for building and testing, widely followed within the community. This is crucial, as we must support all variants and possible log output of such frameworks. Moreover, it would be desirable if both languages show a similar build frequency in Figure 6. The first two languages that fulfilled these criteria were Ruby and Java. From

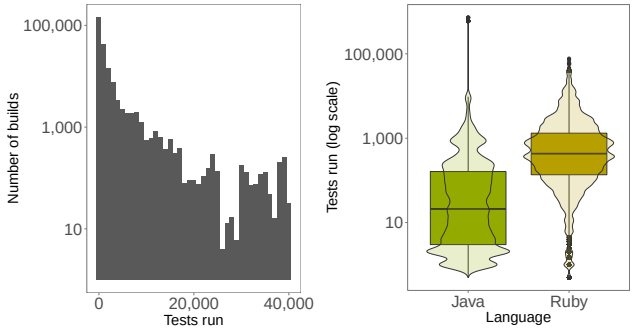

Figure 7: Number of tests run per build (left, log-scale) and number of tests per build per language (right, log-scale).

these two languages, we sampled the 1,359 projects (Ruby: 936, Java: 423) that showed considerable TRAVIS CI use (> 50 builds in Figure 5).

All further analyses are based on an in-depth investigation of 1,359 Java and Ruby projects, for which we downloaded and analyzed 2,640,825 build logs from TRAVIS CI (1.5 TB). This is the TRAVISTORRENT data set `travistorrent_5_3_2016`.

4.2 RQ2: How central is testing to CI?

The purpose of CI is usually twofold: first, to ensure that the project build process can construct a final product out of the project's constituents and second, to execute the project's automated tests. The testing phase is not a mandatory step: in our data set, 31% of Java projects and 12.5% of the Ruby projects do not feature test runs. Overall, 81% of the projects we examined feature test runs as part of their CI process. On a per build level, 96% of the builds in our sample feature at least one test execution.

RQ2.1 How many tests are executed per build? Figure 7 presents a histogram of the number of tests run per build on the left-hand side, and an analysis of the tests run per language on its right hand side in the beanplot.

As expected, the histogram follows a near-power law distribution often observed in empirical studies [17]: most projects execute a smaller amount of tests, while a few run a lot of tests (mean: 1,433; median: 248; 95%: 6,779). A particular outlier in our data set was a project that consistently executed more that 700,000 tests per build. Upon manual inspection of the project (GOOGLE/GUAVA),

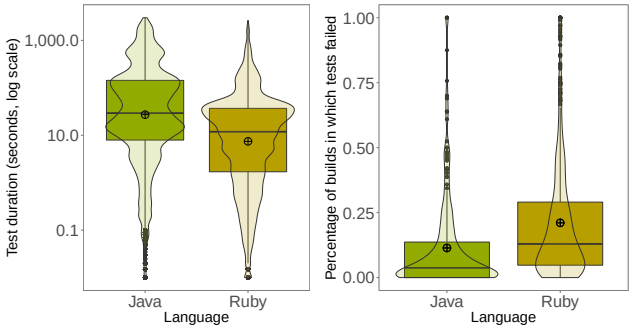

Figure 8: Beanplots of test duration (left) and percentage of test failures broken down per project and language (right).

Table 1: Descriptive statistics for CI latencies (minutes)

Latency type	5%	median	mean	80%	95%
Commit push latency	0.1	0.4	182	17.1	1,201
Job scheduling latency	0.3	1.2	33.9	8.7	85.9
Build environment setup latency	0	0	0.05	0.05	0.35
Build execution latency	0.8	8.3	19.64	30.2	77

we found that it automatically generates test cases at test runtime.

The right hand side of Figure 7 shows the distribution of test runs across the two programming languages in our sample; on average, Ruby builds run significantly more tests (median: 440) than Java builds (median: 15), revealed by a pairwise Wilcoxon test with a very large ($\hat{A}_{12} = 0.82$) effect size.

RQ2.2: How long does it take to execute tests on the CI? Figure 8 depicts a log-scale bean- and boxplot of the test duration, split by language. We observe that the median test duration is relatively short, at $\sim$1 minute for Java and 10 seconds for Ruby projects. Despite the significant differences in the test duration, both languages feature a similar set of large outliers, reaching maximum test execution times of over 30 minutes.

RQ2.3: How much latency does CI introduce in test feedback? CI introduces a level of indirection between developers and the feedback that they receive from testing, especially when compared to testing in the IDE. In the TRAVIS CI and GITHUB setting, latency can be broken down into: the time 1) between locally committing and pushing to GITHUB (commit-push latency), 2) to schedule a build after a push (job scheduling latency), 3) to provision the infrastructure (build environment setup latency), and 4) to execute the build (build execution latency). To calculate latencies, we exploit the fact that TRAVIS CI builds are always triggered by GITHUB push events. The process to connect builds to commits is as follows:

1) We identify all commits that were built on TRAVIS CI and map commits and builds to each other (Section 3.3).
2) For each commit, we search for the GITHUB push event that transferred it. As multiple push events might exist that contain a specific commit (e.g. a push to a repository that is cherry-picked by a fork and pushed to a different branch), we always select the earliest.
3) We list all commits in the push event and select the first one as our reference. We chose to keep the information about the first commit (and not e.g. the commit actually built) as this allows us to calculate the total latency induced by the developer (not pushing a commit creates latency to the potential feedback received by the CI) and compare it to the latency introduced by the CI.

Table 1 presents an overview of the latencies involved in receiving feedback from testing on the CI environment. The results reveal two interesting findings: firstly, developers tend to push their commits to the central repository shortly after they record them locally; secondly, the total time the code remains within the TRAVIS CI environment dominates the latency time.

Developers typically push their commits quickly after their creation to the remote repository. The commit push latency distribution is very skewed; 80% of the commits only stay on the developer's local repositories for less than 17 minutes, and 50% are pushed even within one minute. The distribution skewness is a result of using distributed version control; developers have the option to commit without internet access or to delay showing their changes until perfected. Our data shows that this only happens in few cases.

On TRAVIS CI, a build is scheduled immediately (average latency is less than a second) after commits are pushed to a repository. The time between scheduling and actual build execution depends upon resource availability for free OSS projects. The added latency is about one minute in the median case, but can reach up to nine minutes for the 80% case. While this latency is significant, it represents the price to pay for the free service offered by TRAVIS CI; builds on commercial versions are scheduled immediately.

Moreover, before executing each build, TRAVIS CI needs to provision a virtual machine or Docker container with the required programming language and runtime combination. This operation is usually fast: on average, across all builds, it takes 3.1 seconds (median: 0; 80%: 3; 90%: 22). However, as build job execution is mostly serial on TRAVIS CI, the time cost to be paid is linear in the number of executed jobs or build environments. As the average project in our data set spawns 5 jobs (median: 3; 80%: 7; 90%: 10), running the build in multiple environments induces an average time overhead of 25s just for provisioning operations on the CI server.

The build process itself adds another 8.5 minutes of median latency to the test run. As there is a strict 50 minute cap on the length of build jobs, 80% of the builds last 30 minutes or less.

To sum up the findings, the use of CI adds a median of 10 minutes to the time required to get feedback from testing, while the 80% case is significantly worse. The build time, which is entirely in each project's domain, dominates the feedback latency.

4.3 RQ3: How do tests influence the build result?

With this research question, we aim to unveil how often tests fail when executed on the CI server, how often they break the build, whether they are a decisive part of CI, and if multiple build environments are useful in terms of causing different test results.

RQ3.1: How often do tests fail? In RQ3.1, we are interested in how often tests fail, when they are executed as part of the `script` phase of a TRAVIS CI build.

For all 1,108 projects with test executions, Figure 8 shows a beanplot of the ratio of builds with at least one failed test, broken down per language. With a median of 2.9% for Java (mean: 10.3%) and a median of 12.7% (mean: 19.8%) for Ruby, the ratio of test failures among all builds is significantly higher in Ruby than in Java projects, confirmed by a Wilcoxon rank sum test with a large effect size ($\hat{A}_{12} = 0.70$).

RQ3.2: How often do tests break the build? Beyond merely knowing how often tests fail, we want to research which impact this has in the bigger picture of the CI process.

Figure 9 shows an aggregated-per-project break-down of the build outcomes of all 1,108 projects which executed tests, separated by Java and Ruby. Next to each stacked bar, we report its participation in the overall build result. The graphs of Java and Ruby are largely comparable, with a similar build result distribution, and small differences within the groups. In total, cancels are very rare

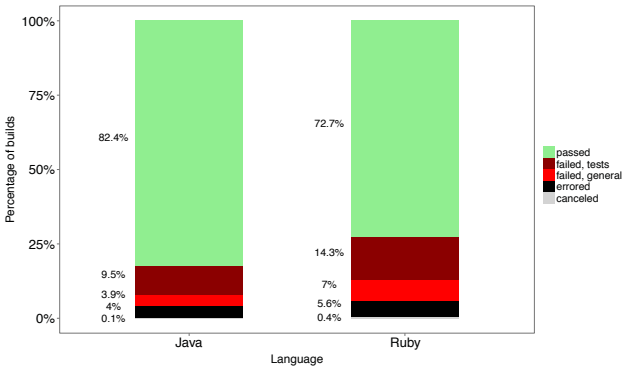

Figure 9: Distribution of build status per language.

and infrastructural problems cause builds to break in around 5% of cases. Failures during the build process are responsible for most broken builds, and they are more frequent in Ruby (21.3 percentage points) than Java (14.4 % p.). In both cases, the single largest build-breaking phase is testing, with failed tests responsible for 59.0% of broken builds in Java and 52.3% in Ruby projects.

RQ3.3: Are tests a decisive part of CI? Table 2 shows the number of builds with test failures, which have an overall failed result, and aggregates this on a per-project level. For this aggregation, a project has to consistently ignore the result of the test execution for all its history. This means that if the tests failed, this never led to a failing build. The table shows that, in general, the test execution result is decisive for the overall build result, at a per-project aggregated level of 98.3%.

Consistently ignoring the test execution result is very rare (1.7%). However, it is quite common that the failed test execution result of individual builds has no influence on the whole result (35.2%). Ignoring such individual test failures in a build is more common in Ruby (39.8%) than Java (13.6%).

RQ3.4: Does integration in different environments lead to different test results? Each build comprises $n \geqslant 1$ job(s), which perform the same build steps in altered environments on the same GIT checkout (see Section 2.2). However, integration in different environments is also expensive, as the required build computation time becomes $n\times$ the individual build time. One might argue that it therefore only makes sense to do continuous integration in several environments when their execution leads to different results, capturing errors that would not have been caught with one single environment. In RQ3.4, we set out to answer this question.

Table 3 gives an overview of how many times the execution of tests in different environments leads to a different build outcome. We observe that in total, 11.4% of builds have a different integration result, meaning that there were at least two jobs in which the test execution resulted in a different status. This effect is much more pronounced for Ruby (15.6%) than for Java (2.3%) systems. In total, over 60% of projects have at least one build in which there was a different test execution result among jobs.

5. DISCUSSION

In this section, we first discuss our propositions by combining the results of RQs 1-3 and then show how we mitigated possible threats to their validity. We also point out promising future research topics.

5.1 Results

Before we started this study, we had one data point that indicated TRAVIS CI use might be as high as 90% [11]. By contrast,

Table 2: How often are test results ignored in a build result?

		Tests Fail → Build Fail	Tests Fail ↯ Build Pass	Total
Build level	Java	7,286 (86.4%)	1,146 (13.6%)	8,432
	Ruby	23,852 (60.2%)	15,773 (39.8%)	39,625
	Both	31,138 (64.8%)	16,919 (35.2%)	48,057
Project level	Java	197 (98.0%)	4 (2.0%)	201
	Ruby	727 (98.4%)	12 (1.6%)	739
	Both	924 (98.3%)	16 (1.7%)	940

Table 3: Same build in different integr. environments.

Build Result		Identical	Different	Total
Jobs	Java	74,666 (97.7%)	1,725 (2.3%)	76,391
	Ruby	140,833 (84.4%)	25,979 (15.6%)	166,812
	Both	215,499 (88.6%)	27,704 (11.4%)	243,203
Projects	Java	196 (66.0%)	101 (34.0%)	297
	Ruby	240 (29.3%)	579 (70.7%)	819
	Both	436 (39.1%)	680 (60.9%)	1,116

regarding *P1*, we found in RQ1 that around a third of all investigated projects used TRAVIS CI in 2015. While this is significantly lower, it still shows a relatively widespread adoption. We attribute this to the facts that TRAVIS CI provides easy-to-use default build configurations for a wide array of languages and that it is gratis.

> Around 30% of GITHUB OSS projects that could potentially use TRAVIS CI for free, also make active use of it (*P1*).

Compared to about 60% of state-of-the-art GITHUB projects using ASATs [8] and some 50% of projects in general doing testing [17], a number of reasons might hinder an even more-widespread use of TRAVIS CI: Famous GITHUB projects such as SCALA/SCALA[10] often run their own CI server (see Section 2.1).[11] This exemplifies that from the 30% adoption rate, it does not follow that 70% of projects do not use CI. For high-profile projects with a complex CI process, the migration to TRAVIS CI would involve high initial risks and costs. One interesting line of future research therefore is to find out the overall CI adoption among top-GITHUB projects. It might be that word about the benefits of CI, or TRAVIS CI itself has not spread to every project maintainer yet.

A paradoxical finding was that a few projects written in languages that are not supported by TRAVIS CI, still used it (Figure 5 "Other"). A manual investigation into a sample of such projects revealed that they also contained a smaller portion of code written in a language supported by TRAVIS CI, for which they did enable CI. TRAVIS CI has traditionally had deep roots in the Ruby community [40]. We found that this bias towards the Ruby community has largely vanished nowadays.

> TRAVIS CI adoption is uniform among most languages TRAVIS CI supports (*P1*).

In RQ1, we also found that the number of builds for projects varies per language, but this variation is contained. Figure 6 is a measure of how active projects in specific languages are and how frequently they push, thereby triggering an automated CI build and thus leveraging the full potential of CI: for each push with new commits on one branch, a new TRAVIS CI build is triggered on the head of the branch (see Section 2.2). This stresses the fact that CI is a software engineering concept orthogonal to the chosen programming language used, as it applies to many languages in a similar

[10] https://github.com/scala/scala
[11] https://scala-ci.typesafe.com/

way. Thanks to homogeneous CI use and adoption of TRAVIS CI, researchers find a large corpus of comparable projects with similar CI patterns. This eases the interpretation of research results and decreases the need for an extensive control of external variables that would be necessary if the projects came from heterogeneous build systems. Specifically, factors such as build duration and setup process are more reliable in the homogeneous TRAVIS CI case.

With *P2*, we were interested in the importance of testing in the CI process. Overall, we found that testing stands central in CI.

> Testing happens in the majority of builds. Only ∼20% of the studied projects never included a testing phase in their CI (*P2*).

Failed tests have a higher impact, both relative and absolute and irrespective of the programming language, than compile errors, dependency resolution problems and other static checks combined. This puts the finding by Seo et al. [10] that most build errors stem from missing dependencies in perspective. Our investigation shows that issues in the compilation phase represent only a minority of the causes for broken builds (3.9% in Java, 7% in Ruby) in the bigger picture of a CI environment.

> Failing tests are the single dominant reason for unsuccessful builds (*P2*).

Having established that the median number of builds in which tests fail is modest (RQ3.1), but that test failures are responsible for over half of all broken builds (RQ3.2), the question stands in how many instances tests fail, but the developers configured their CI in such a way that the negative test execution result does not influence the overall build status. In such cases, the test execution would not be a crucial factor to the build success. As described in Section 2.2, TRAVIS CI runs tests per default and it would be a deliberate choice by developers to ignore the test result.

> Projects which consistently ignore the test result are very rare. However, ignoring individual builds is quite common (*P2*).

One possible reason for the difference between projects in Java and Ruby might stem from the fact that projects which do not run tests on the CI only make use of a sub-set of CI features, and therefore also have fewer builds. It might make more sense to just compile Java applications than have a CI setup for a Ruby application (that does not need to be compiled) without tests.

> The typical Ruby project has an ten times more tests than the typical Java project (*P2, P3*).

Given the size of our samples (423 Java and 936 Ruby projects), we believe that this difference might be attributable to the fundamental differences in the Java and Ruby programming languages. Specifically, the lack of a type system in Ruby might force developers to write more tests for what the compiler can check automatically in the case of Java [41]. We need a broader study with a larger sample of dynamic and static languages to verify whether this holds generally. With more tests, we naturally expect more test failures as a result.

> Ruby projects have a four-times higher likelihood for their tests to fail in the CI environment than Java projects (*P3*).

While CI testing also happens in Java, these findings raise the question whether Java in general and JUNIT test cases in particular are the best possible study objects when researching testing.

Having established that the large majority of projects execute tests as part of their CI, it remains to find out which indirection in the feedback cycle their execution causes (*P4*) and compare it to local test executions in the IDE (*P5*).

Multiple test environments are only useful when they also lead to different tests results in practice (*P4*). Otherwise, they just consume unnecessary resources and time. Our analysis in RQ3.4 showed that test execution results vary per environment for ∼10% of test executions. Some differing test results of these 10%, stem from a sheer re-execution of tests, uncovering flickering tests. One way to uncover these would be to re-execute failed builds on TRAVIS CI and observe execution result changes. We refrained from doing so, as it would involve large costs on TRAVIS CI.

> The average project on TRAVIS CI is tested in five integration environments (*P4*).

Our results suggest that by exposing more test failures, integration in multiple environments 1) is helpful in uncovering a substantial part of test failures and thus likely bugs that would otherwise be missed and 2) does lead to uncovering failing tests that would not be captured by running the tests locally, at the cost of an increased feedback latency. It might be more helpful for languages like Ruby than Java.

Having established that CI adoption is relatively widespread (*P1*), that testing is integral to CI (*P2*), that it depends very much on the project language (*P3*), and that multiple integration environments are helpful in practice, it remains to discuss whether testing on the CI could replace local testing in the IDE in terms of providing quick feedback (*P5*). For this, we consider the latency induced by CI.

In RQ2.3, we observed that commits only live shortly (typically, less than 20 minutes) in the developer's private repositories before developers push them upstream to the remote mainline. Popular belief about distributed version control indicates that developers should perfect their changes locally before sharing them publicly, which one would normally assume to take longer than 20 minutes. Why do developers apparently go against this norm? Previous work on developer usage of pull requests [26, 42] might provide an indication about potential reasons. In a "fail early, fail often" approach, integrators and contributors overwhelmingly rely on their tests to assess the quality of contributed code. Instead of perfecting code in a dark corner, this has the advantage of building global awareness through communication in the pull request discussion. While collaborating developers crave for fast feedback, with 8.3 minutes build time in the median, the CI introduces measurable delays into this feedback process.

> The main factor for delayed feedback from test runs is the time required to execute the build (*P5*).

By contrast, an empirical investigation of the testing habits of 416 developers found that the median test duration ("latency") in the IDE is 0.54 seconds [17]. This is three orders of magnitude faster than running tests on the CI: testing in the IDE is fast-paced, most test executions fail (65% in comparison to 15% on the CI) and developers usually only execute one test (in contrast to all tests on the CI). Hence, the aim, the length of the feedback cycle and their observed different use in practice, suggest that testing on the CI may not be a suitable surrogate for local testing, if fast feedback is required. These findings contradict empirical evidence of projects like the "Eclipse Platform UI," which reported to increasingly offload their test executions to the CI [17]. This calls for future research: developers spend a quarter of their time working on tests [17], yet rarely execute them in the IDE. If they do, testing in the IDE is immediate and most tests fail. On the CI, most test executions pass and there is a notable delay between creating code and receiving feedback (usually, 10 minutes for pushing plus 10 minutes for building). This leaves us with the question where else developers run their tests. One such place could be building on the command line.

5.2 Threats to Validity

In this section, we discuss the limitations and threats that affect the validity of our study, and show how we mitigated them.

Construct validity concerns errors caused by the way we collect data. For capturing build data, we relied on the custom-created tools BUILDLOG ANALYZER, TRAVIS POKER and TRAVIS HARVESTER. To gain confidence that these tools provide us with the correct data and analysis results, we tested them with several exemplary build logs, which we also packaged into their repository and made them publicly available for replication purposes.

Threats to **internal validity** are inherent to how we performed our study. We identify two major technical threats: 1) Through the use of `git push -force`, developers can voluntarily rewrite their repository's history, leading to a wrong build linearization. By counting the number of commits that we could not resolve with the strategies described in Section 3.3, we received an upper bound for the severity of this threat. Since less than 10% of all commits are affected, we consider its impact small. 2) Developers can re-execute builds (marker ⑥ in Figure 1), for example to re-run a build that failed due to a TRAVIS CI infrastructural problem. In such rare cases, TRAVIS CI overrides the previous build result. This can possibly lead to a large time difference between having pushed commits to GITHUB and TRAVIS CI starting the build, which is why we excluded the top 1% quantile of builds. As our study does not retroactively re-build projects, we are sure to observe the real build problems developers ran into.

Threats to **external validity** concern the generalizability of our results. While we examined 58,032 projects for answering RQ1 and 1,359 projects for RQs 2-3, all projects stem from a single CI environment, namely TRAVIS CI. We explicitly opted for TRAVIS CI because it is frequently used along by projects hosted on GITHUB, which in turn allowed us to combine two data sources and triage data such as time stamps. Nevertheless, we do not know how strongly the idiosyncrasies of both services affect the generalizability of our results on the CI-based software development model. For example, the build duration and latency measurements in RQ2 depend strongly on the load and resources provided by TRAVIS CI. When compared to other build servers, they might only be proportionally correct.

Due to the fact that we could only study the publicly visible part of TRAVIS CI and GITHUB, our study gives no indications of the practices in private projects, which might deviate significantly.

Similarly, for RQ2 and RQ3 we only considered Java and Ruby projects to reduce the variety of build technologies and test runners. We only compared Ruby as one instance of a dynamically-typed language to Java as one instance of a statically-typed language. Some of the differences we found between them might be more attributable to the specific differences between Ruby and Java, rather than the general nature of their type system. As programming language communities have different testing cultures, an important avenue of future work is to increase the number of programming languages that are part of this investigation.

6. FUTURE WORK

Our work opens an array of opportunities for future work in the CI domain, both for researchers and CI tool builders.

Researchers can build on top of our results, the curated TRAVISTORRENT data set, and open research questions. For example, we have shown that there is such significant use of TRAVIS CI among GITHUB projects that it might often be a valid shortcut to study only the streamlined TRAVIS CI project instead of several more diverse CI sources. In particular, our investigation raises the question whether projects in dynamically-typed languages like Ruby generally have a higher CI failure rate, and whether and which effects this has, particularly when they switch between languages. While often considered annoying and a bad smell, we do not know whether prolonged periods of broken builds or following the "break early, break often" strategy translates into worse project quality and decreased productivity. In fact, it is unclear whether breaking the build has adverse effects at all, depending on the development model. The approach proposed in Section 3.3 enables such studies that require previous build states. Deeper studies into integration environments could unveil how much of their uncovering 11.4% more build failures benefit is due to simply the repeated re-execution of the build ("flaky tests").

CI Builders can use our tooling to improve their user experience. By directly indicating the nature of a build breakage, they remove the need for developers to manually inspect potentially very large build logs. We strongly believe this could improve developers' efficiency when dealing with failing builds.

7. CONCLUSION

In conclusion, we found that a third of popular GITHUB projects make use of TRAVIS CI, and their adoption is mostly uniform (*P1*). This finding contrasts prior research that found a far higher adoption rate of 70%. Our investigation shows that testing is an established and integral part in CI in OSS. It is the single most important reason for integration to break, more prevalent than compile errors, missing dependencies, build cancellations and provisioning problems together (*P2*). Testing is configured as a crucial factor to the success of the build, but exceptions are made on an individual level. We found that testing on the CI is highly dependent on the language, and that a dynamically typed language like Ruby has up to ten times more tests and leads to a higher build breakage rate due to tests than a statically typed language like Java (*P3*). CI introduces a feature that local test execution cannot provide: integration in multiple environments. This is commonly used on TRAVIS CI, and tests show different behavior when executed in multiple environments in about 10% of builds (*P4*), showcasing the value of multiple integration environments. Contrary to prior anecdotal evidence, testing on the CI does not seem a good replacement for local test executions and also does not appear to be used as such in practice (*P5*): with a latency of more than 20 minutes between writing code and receiving test feedback, the way developers use CI induces a latency that stands in contrast to the fast-paced nature of testing in the IDE and the idea that developer tests should provide quick feedback. The low test failure rates hint at the fact that developers send their contributions pre-tested to the CI server.

Apart from research on our five propositions *P1-P5*, this paper makes the following key contributions:

1) A novel method of analyzing build logs to learn about past test executions in the context of CI.
2) A comparative study of CI testing patterns between a large corpus of projects written in a statically and a dynamically typed language.
3) The implementation and introduction of TRAVISTORRENT, an OSS open-access database for analyzed TRAVIS CI build logs combined with GITHUB data from GHTORRENT.

Acknowledgments

We thank Mathias Meyer (CEO of TRAVIS CI), Arie van Deursen, Felienne Hermans, Alexey Zagalsky, Maurício Aniche, and previous anonymous reviewers for their feedback.

8. REFERENCES

[1] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison Wesley, 2007.

[2] M. Fowler and M. Foemmel, "Continuous integration," 2006. https://web.archive.org/web/20170327121714/http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf.

[3] M. A. Cusumano and R. W. Selby, "Microsoft secrets," 1997.

[4] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[5] M. Brandtner, E. Giger, and H. C. Gall, "Sqa-mashup: A mashup framework for continuous integration," *Information & Software Technology*, vol. 65, pp. 97–113, 2015.

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

[7] C. Sadowski, J. Van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, pp. 598–608, IEEE, 2015.

[8] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, pp. 470–481, IEEE, 2016.

[9] J. Rasmusson, "Long build trouble shooting guide," in *Extreme Programming and Agile Methods - XP/Agile Universe 2004* (C. Zannier, H. Erdogmus, and L. Lindstrom, eds.), vol. 3134 of *LNCS*, pp. 13–21, Springer, 2004.

[10] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at Google)," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 724–734, ACM, 2014.

[11] B. Vasilescu, S. Van Schuylenburg, J. Wulms, A. Serebrenik, and M. G. van den Brand, "Continuous integration in a social-coding world: Empirical evidence from GitHub," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pp. 401–405, IEEE, 2014.

[12] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 805–816, ACM, 2015.

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

[14] L. Prechelt, "An empirical comparison of seven programming languages," *Computer*, vol. 33, no. 10, pp. 23–29, 2000.

[15] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 23, no. 4, pp. 22–29, 2006.

[16] D. Janzen and H. Saiedian, "Test-driven development: Concepts, taxonomy, and future direction," *IEEE Computer*, vol. 38, no. 9, pp. 43–50, 2005.

[17] M. Beller, G. Gousios, A. Panichella, and A. Zaidman, "When, how, and why developers (do not) test in their IDEs," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 179–190, ACM, 2015.

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

[19] R. Ablett, E. Sharlin, F. Maurer, J. Denzinger, and C. Schock, "Buildbot: Robotic monitoring of agile software development teams," in *Proceedings of the International Symposium on Robot and Human interactive Communication (RO-MAN)*, pp. 931–936, IEEE, 2007.

[20] R. Rogers, "Scaling continuous integration," in *Extreme programming and agile processes in software engineering*, no. 3092 in LNCS, pp. 68–76, 2004.

[21] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pp. 33:1–33:11, ACM, 2012.

[22] C. Watters and P. Johnson, "Version numbering in single development and test environment," Dec. 29 2011. US Patent App. 13/339,906.

[23] "What is Travis CI." https://web.archive.org/web/20170327121755/https://github.com/travis-ci/travis-ci/blob/2ea7620f4be51a345632e355260b22511198ea64/README.textile#goals.

[24] M. Beller, I. Levaja, A. Panichella, G. Gousios, and A. Zaidman, "How to catch 'em all: Watchdog, a family of IDE plug-ins to assess testing," in *3rd International Workshop on Software Engineering Research and Industrial Practice (SER&IP 2016)*, pp. 53–56, IEEE, 2016.

[25] M. Beller, G. Gousios, and A. Zaidman, "How (much) do developers test?," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 559–562, IEEE, 2015.

[26] G. Gousios, A. Zaidman, M.-A. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 358–368, IEEE, 2015.

[27] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, 2014.

[28] G. Gousios, "The GHTorrent dataset and tool suite," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*, pp. 233–236, IEEE, 2013.

[29] M. Beller, G. Gousios, and A. Zaidman, "Travistorrent: Synthesizing travis ci and github for full-stack research on continuous integration," in *Proceedings of the Proceedings of the 14th International Conference on Mining Software Repositories (MSR)*, 2017.

[30] O. Tange, "GNU Parallel - the command-line power tool," *;login: The USENIX Magazine*, vol. 36, pp. 42–47, Feb 2011.

[31] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan, "A large-scale empirical study of the relationship between build technology and build maintenance," *Empirical Software Engineering*, vol. 20, no. 6, pp. 1587–1633, 2015.

[32] D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, and A. Hellesoy, *The RSpec Book: Behaviour Driven*

Development with Rspec, Cucumber, and Friends. Pragmatic Bookshelf, 1st ed., 2010.

[33] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified modeling language reference manual, the*. Pearson Higher Education, 2004.

[34] W. G. Hopkins, *A new view of statistics*. 1997. https://web.archive.org/web/20170327121841/http://newstats.org.

[35] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3-4, pp. 591–611, 1965.

[36] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[37] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Software Technology*, vol. 49, no. 11, pp. 1073–1086, 2007.

[38] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. German, and D. Damian, "An in-depth study of the promises and perils of mining GitHub," *Empirical Software Engineering*, pp. 1–37, 2015.

[39] Github, "Language trends on GitHub." https://web.archive.org/web/20170327122123/https://github.com/blog/2047-language-trends-on-github.

[40] "The Travis CI blog: Supporting the ruby ecosystem, together." https://blog.travis-ci.com/2016-02-03-supporting-the-ruby-ecosystem-together.

[41] L. Tratt and R. Wuyts, "Guest editors' introduction: Dynamically typed languages," *IEEE Software*, vol. 24, no. 5, pp. 28–30, 2007.

[42] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *Proceedings of the 38th International Conference on Software Engineering*, ICSE, May 2016.