# Empirical Study of Restarted and Flaky Builds on Travis CI

Thomas Durieux
INESC-ID and IST, University of Lisbon, Portugal
thomas@durieux.me

Claire Le Goues
Carnegie Mellon University
clegoues@cs.cmu.edu

Michael Hilton
Carnegie Mellon University
mhilton@cmu.edu

Rui Abreu
INESC-ID and IST, University of Lisbon, Portugal
rui@computer.org



## ABSTRACT

Continuous Integration (CI) is a development practice where developers frequently integrate code into a common codebase. After the code is integrated, the CI server runs a test suite and other tools to produce a set of reports (e.g., the output of linters and tests). If the result of a CI test run is unexpected, developers have the option to manually restart the build, re-running the same test suite on the same code; this can reveal build flakiness, if the restarted build outcome differs from the original build.

In this study, we analyze restarted builds, flaky builds, and their impact on the development workflow. We observe that developers restart at least 1.72% of builds, amounting to 56,522 restarted builds in our Travis CI dataset. We observe that more mature and more complex projects are more likely to include restarted builds. The restarted builds are mostly builds that are initially failing due to a test, network problem, or a Travis CI limitations such as execution timeout. Finally, we observe that restarted builds have an impact on development workflow. Indeed, in 54.42% of the restarted builds, the developers analyze and restart a build within an hour of the initial build execution. This suggests that developers wait for CI results, interrupting their workflow to address the issue. Restarted builds also slow down the merging of pull requests by a factor of three, bringing median merging time from 16h to 48h.

**ACM Reference Format:**
Thomas Durieux, Claire Le Goues, Michael Hilton, and Rui Abreu. 2020. Empirical Study of Restarted and Flaky Builds on Travis CI. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387460

## 1 INTRODUCTION

Software engineers use Continuous Integration (CI) not only to integrate their work into a common branch, but also to constantly ensure the quality of their contributions. Continuous integration was originally introduced in the twelve Extreme Programming practices [1], and it is now considered to a key software development best practice in industry and Open-Source Software alike.

Continuous integration is an automatic process that typically executes the test suite, may further analyze code quality using static analyzers, and can automatically deploy new software versions. It is generally triggered for each new software change, or at a regular interval, such as daily. When a CI build fails, the developer is notified, and they typically debug the failing build. Once the reason for failure is identified, the developer can address the problem, generally by modifying or revoking the code changes that triggered the CI process in the first place.

However, there are situations in which the CI outcome is unexpected, e.g., the build may fail unexpectedly, or for unanticipated reasons. In this case, developers may *restart* the build (an option generally provided by CI services) to check whether the unexpected outcome is a result of system flakiness or some problem in the CI environment (like network latency). A build whose output changes after a restart is referred to as a *flaky build*, because by definition, the software for which a build has been restarted has not changed (but the CI result has). Although there exists prior work studying flaky tests (one contributor to flaky builds) [11], to our knowledge, there is no work that studies the practice of build restarts in a CI context, and its implication for development workflow.

Free software and services, such as Jenkins[1] and Travis CI[2], simplify the adoption of Continuous Integration by the whole community. In this work we study Travis CI, one of the largest and most popular Continuous Integration services in use today. It is free, and its build execution data is publicly available. Based on Travis CI data, we design a study that aims to gain a better understanding of the build restart process and its implication on development workflow.

We have multiple goals in this study. First, we identify how frequently developers restart their builds, and how often restarts surface build flakiness. Second, we identify the characteristics of projects that restart builds, to understand whether restarting is a common, general practice (or not). Third, we analyze the reasons that motivate developers to restart their builds. Finally, we study the impact of restarted builds on developer workflow. Based on this study, we discuss potential improvements for continuous integration, and ideas for a fully automatized build restart process that could save significant developer time.

Our main observations are that developers restart at least 1.72% of Travis CI builds, which represents 56,522 restarted builds in our dataset. We observe that the projects that have a restarted build differ from those that do not: more mature and more complex


*MSR '20, October 5–6, 2020, Seoul, Republic of Korea*

ACM ISBN 978-1-4503-7517-7/20/05.
https://doi.org/10.1145/3379597.3387460

[1]Jenkins website: https://jenkins.io/, visited March 27, 2020
[2]Travis website: https://travis-ci.org/, visited March 27, 2020

projects are more likely to restart builds. We observe that build restart rates differ by project language. Builds are most commonly restarted in response to test failures, e.g., flaky tests, as well as Travis CI limitations such as execution timeout. Finally, we observe that restarted builds appear to have a major impact on developer workflow: In 53.42% of the restarted builds, developers initiate a restart within one hour of failure. This suggests that developers choose to wait for CI results, interrupting (or pausing) their work to address the issue. Restarted builds also slow down the merging of pull requests by a factor of three, bringing the median merging time from 16h to 48h.

To summarize, the contributions of this paper are:

- A comprehensive study on restarted builds, with implications for CI system design;
- A framework, coined TravisListener, to collect real-time data from Travis CI;
- A dataset of 3,286,773 builds and 56,522 restarted builds collected in live;
- A dataset of 75,139,295 builds that cover the complete build history of 171,057 repositories.

The remainder of this paper is organized as follows. Section 2 presents background on Travis CI and the idea of restarted builds. Section 3 presents study design, including research questions and data collection and analysis. Section 4 presents results, followed by discussion (Section 5). Section 6 presents threats to the validity. Section 7 discusses related work, and Section 8 concludes.

## 2 BACKGROUND

In this section, we briefly describe Travis CI and introduce the notion of restarted builds as well as the technical implications of a restarted build on Travis CI.

### 2.1 CI and Travis CI

Travis CI is a company that offers an open-source continuous integration service tightly integrated with GitHub. It allows developers to build their projects without maintaining their own infrastructure. Travis CI provides a simple interface to configure build tasks that are executed for a set of given events: pull requests, commits, crons, and API calls. Currently, Travis CI supports 34 programming languages, including Python, Node.js, Java, C, C++, in three operating systems: Linux, Windows, and Mac OSX. It also supports technologies like Docker, Android apps, iOS apps, and various databases. The Travis CI service is free for open-source projects; a paid version is available for private projects. It is currently used by more than 932,977 open-source projects and 600,000 users.[3]

Travis CI interacts with GitHub via a set of webhooks that are triggered by GitHub events. For each event, Travis CI sets up a new build by reading the configuration that developers wrote in the repository (.travis.yml file). Each build is composed of one or several jobs. A job is the execution of the build in a specific environment, for example, one job runs with Java 8 and one with Java 9, or a job can also be used for specific tasks such as deploying Docker images.

[3]Metrics provided at https://travis-ci.org, visited March 27, 2020

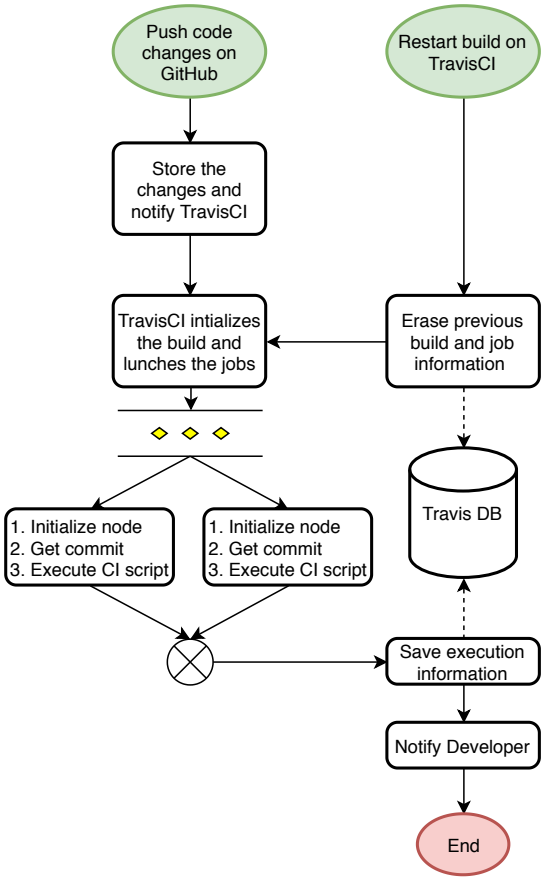


**Figure 1: Flowchart of the build restarting process.**

### 2.2 Restarted Builds

A *restarted* build is one that has been relaunched manually, without a corresponding underlying change in the project. In general, a developer might restart a build when they observe inconsistent or unexpected behavior that they suspect may be due to build or environment flakiness that could be avoided with a new run. In this study, we focus on restarted builds from Travis CI, because Travis CI is one of the largest continuous integration services, and its build information is also publicly available.

In the context of Travis CI, a restarted build is triggered when a developer clicks on the restart button that the Travis CI interface provides. Figure 1 presents a flowchart of a restarted build on Travis CI. The first phase of a restarted build is identical to any build execution on Travis CI. First, a developer pushes a code change (commits) to GitHub. Second, GitHub processes the commits, and notifies Travis CI of new code. Third, Travis CI analyzes the Travis CI configuration (`.travis.yml` file), initializes the different jobs that are configured, and saves the build and job information into its database. Fourth, Travis CI executes the configured jobs. Fifth, Travis CI saves the execution information in its database (execution time, execution logs). Finally, Travis CI notifies the developers that the build execution has ended, and provides build status.

A *restarted* build is triggered directly by the developer, typically by clicking the restart button provided by the Travis CI web interface (note that Travis CI also provides an API to restart the build). Importantly, once a build is restarted, all information saved during the original execution is erased and overwritten. It is therefore not naively possible to use the Travis CI API to learn the original status or output of a build that has been restarted. Subsequently, execution behavior is identical to a traditional build: Travis CI configures and executes the jobs, and notifies developers.

The implications of this workflow are first, that developers have to restart a build manually, typically in response to unexpected build behavior. Second, the Travis CI API does not support comparison between a restarted build and its original status. We discuss our methodology for performing such comparisons in Section 3.2.

## 3 STUDY DESIGN

In this section, we present our study on the restarted builds on Travis CI and their impact on the developer's workflow.

### 3.1 Research Questions

During this study, we study four different aspects of restarted builds to answer the following research questions.

**RQ1**. How frequently do developers restart builds? How often do restarts identify flaky behavior? We answer this question by collecting live data from Travis CI, estimating build restart frequency and identifying how many of those restarted builds have a different status outcome.

**RQ2**. Do the projects that restart builds have the same characteristics as other projects that use Travis CI? We answer this question by studying the characteristics of projects that restarted builds over the period of our live study. We aim to identify potential common characteristics of those projects.

**RQ3**. Why do developers restart builds? In the third research question, we analyze build logs to the extract reasons for the failures of builds that developers choose to restart.

**RQ4**. What is the impact of flaky builds on development workflow? In this final research question, we study when (temporally) developers restart build, as well as the relationship between build restarts and pull requests.

### 3.2 TravisListener Infrastructure

Figure 2 depicts our infrastructure for collecting the data required to answer our four research questions. The infrastructure consists of seven modules: three services, two plugins, a dashboard, and a database. The infrastructure is built on top of Docker compose v2.4.[4] Docker Compose is straightforward to install, scalable, and resilient (given, e.g., its auto-restart capabilities). Each module of our infrastructure is thus a docker image integrated into Docker compose. The services, plugins, and the dashboard are implemented using JavaScript and Node.js v.10.

The **Dashboard** provides a web interface to configure and monitor the state of the different modules of the system.

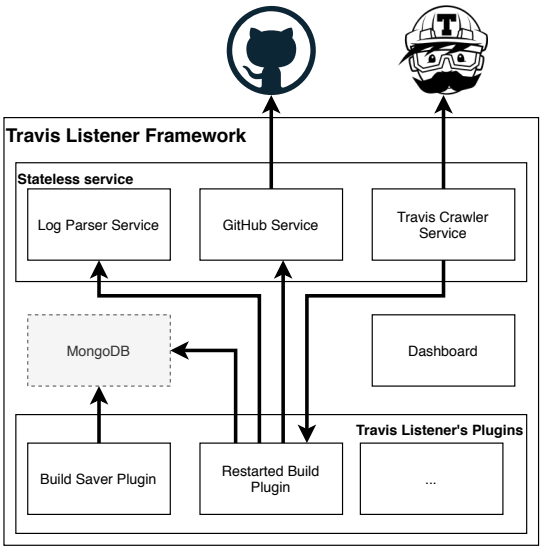


**Figure 2: Architecture of our TravisListener framework.**

For the **Database**, we use MongoDB[5] which integrates well with Node.js and provides data compression by default. Data compression is a useful feature, since we collect millions of highly compressible log files.

The **Log Parser Service** is a service that is used to manipulate logs. The current version of the service provides the following features: 1) Log minimization: removes the meaningless content such as progress bar status and log formatting. 2) Log Diff: produced minimized diffs between two logs by removing all random or time-based content, such as ids or dates. 3) Data extraction: parses the log to extract failures reasons such as test failures, checkstyle warnings, compilation errors, or timeouts. We are currently using 93 regular expressions to extract failure reasons from logs.

The **GitHub Service** is a simple middleware component that handles GitHub API's tokens. It serves to simplify the usage of the GitHub API within TravisListener by centralizing identification and rate limiting.

The **Travis Crawler Service** extracts the information from Travis CI. Its main purpose is to crawl Travis CI to detect any new jobs and builds triggered by Travis CI, live. **Travis Crawler Service** provides a WebSocket service that can be listened to by all TravisListener modules. The WebSocket provides live notifications for any new Travis CI jobs or builds.

The **Build Saver Plugin** listens to the **Travis Crawler Service** and saves all information to the database. We save the following information: Travis CI's job, Travis CI's build, commit information (not including the diff), repository information, and user information. The goal of this plugin is to track all changes, and provide statistics on who is using Travis CI.

The **Restarted Build Plugin** collects the information relevant to the present study. Its goal is to detect restarted builds on Travis CI. As explained in Section 2.2, when a build is restarted by a developer, all the original information is overwritten. Tracking restarted builds

[4]Docker Compose documentation https://docs.docker.com/compose/ visited March 27, 2020

[5]MongoDB website: https://mongodb.github.io/ visited March 27, 2020

thus requires live collection of build data (in our case, using the **Build Saver Plugin**). To detect restarted builds, the **Restarted Build Plugin** crawls periodically (once a day) the collected builds from the 30 previous days, comparing the build start timestamp provided by the Travis CI' API to the start time saved by the **Build Saver Plugin**. If the two times differ, the build was restarted. For each restarted build, we collect the new Travis CI job information and execution logs.

The modularity and the use of Docker images make the framework highly flexible. We develop this live infrastructure with the vision that it will be reused by other developers for other purposes. It could be extended to monitor other continuous integration services, other Git services, or even to provide different services. For example, we envision that TravisListener could easily be extended to create a production service that monitors Travis CI and automatically provides feedback to developers. The source code of TravisListener is available on GitHub: https://github.com/tdurieux/Travis-Listener (double-blinded for the submission).

### 3.3 Data collection overview

Data collection is composed of two main steps. The first step uses TravisListener (see Section 3.2) to collect restarted builds from Travis CI. The second step collects additional information from Travis CI and GitHub to characterize the projects.

To recap, the first step is the collection of the builds and jobs in real-time from Travis CI between Tue, 08 Oct 2019 and Thu, 19 Dec 2019. We save each new build and job from Travis CI in our MongoDB database. When a failing job is detected, we download its execution log. The log is cleaned to reduce its size, and then stored. Every hour, we query Travis CI to request the build information from the last 30 days. We compare the starting date from the freshly collected data with the data that we previously stored in our database. If the build information is different between the two collections (primarily the starting timestamp), it means the build has been restarted. For each restarted build, we collect the new job information and execution logs. We then extract the failure reason using regular expressions, and save everything in the database.

In the second step, we collected data from Travis CI and GitHub to study the different aspects of the repositories. We collected the complete Travis CI build history of all repositories that have triggered a build during our study period. The complete history contains 74,981,298 builds, amounting to 32G. We also collect GitHub repository information, which contains the programming language, the number of stars, the creation date, and the repository size. We collect pull requests comments for all pull requests that have at least one build collected during our study. Finally, we collect the commit history for all the repositories that have at least one restarted build.

Table 1 presents primary statistics of the collected data. In total, we collected 3,286,773 builds, 9,215,866 jobs, 3 jobs per build on average, from 171,057 repositories. During the collection procedure, we observed that 70% of the builds pass, 18% fail, 11% error, and 0.9% are canceled.

Table 2 presents the main statistics of the restarted builds that we identify during our study period. In total, we identify 56,522 restarted builds, 125,461 restarted jobs on 22,345 repositories. Among the restarted builds, 17% of the builds were passing, 48% were failing, 31% errored, and 3.9% were canceled.

Table 3 presents the same metrics, but from TravisTorrent dataset, which we provide for context and comparison. Our dataset differs meaningfully from TravisTorrent. First, because of the live study period, we focus on a short, recent time period; TravisTorrent's dataset spans a wider, older timeframe. Second, TravisTorrent contains the builds from 1,272 repositories, from three different programming languages; we have builds from 171,057 repositories and all 34 supported languages. The two datasets do not have the same goal: TravisTorrent focuses on a few projects but covers a large portion of the build history of those projects, while our dataset focuses on the diversity of projects and languages.

We create this new dataset for two reasons. First, and most important, TravisTorrent does not contain information about restarted builds, and the information cannot be collected post-facto (as described above). Second, our dataset provides a larger diversity of projects and languages than other previous datasets.

To summarize, we live-collected 3,286,773 builds, 9,215,866 jobs and 3,007,857 logs. We detected 56,522 restarted builds. We download the complete Travis CI history of 171,057 projects, representing 75,139,295 builds. The complete uncompressed size of the dataset is approximately 500Gb. The collected data is available on Zenodo with the following DOI: 10.5281/zenodo.3601137.

## 4 STUDY ON RESTARTED BUILDS

In this section, we answer our four research questions.

### 4.1 RQ1. Restarted Builds

In this first research question, we investigate how many builds are restarted by developers. The goal of this first research question is to determine if developers are actually restarting builds; if so, how often; and, whether the restarted builds change in outcome post-restart.

Table 2 presents the main pieces of evidence to answer this first research question. We identified 56,522 restarted builds, representing 1.72% of all the builds that we observed during our study. This indicates that developers are indeed restarting builds. Note that our observation underestimates reality, because we miss some of the restarted builds. Indeed, we are not able to detect that builds that are restarted too quickly because we need to collect the original state of the build and the restarting state of the build. Our Travis CI crawler is blind to builds that are restarted a few seconds after the original. We therefore do not consider such builds. We are also missing builds that are restarted after 30 days, primarily to to reduce the number of requests made on Travis CI. We are surprised to observe that so many builds are restarted, primarily because doing so requires manual labor from a developer with sufficient project privileges; intuitively, we expected that a privileged, manual procedure would appear only marginally throughout a fully automatic process like CI.

Our overall hypothesis is that developers restart a build because they expect the build behavior to change. Figure 3 corroborates our hypothesis. This figure shows how the states of the builds change between the original build and the restarted build. The horizontal axis presents the original state; the vertical axis presents

**Table 1: Overview of the collected data.**

| Metric Name | Value |
|---|---|
| Start Study | Tue, 08 Oct 2019 |
| End Study | Thu, 19 Dec 2019 |
| # Builds | 3,286,773 |
| # Jobs | 9,215,866 |
| # Logs | 3,007,857 |
| # Repositories | 171,057 |
| # Passed builds | 2,306,130 (70%) |
| # Failed builds | 583,415 (18%) |
| # Errored builds | 367,963 (11%) |
| # Canceled builds | 29,265 (0.9%) |
| Avg. # builds per repo | 19 |
| Avg. # jobs per repo | 52 |

**Table 2: Overview of the restarted builds.**

| Metric Name | Value |
|---|---|
| # Restarted Builds | 56,522 (1.7%) |
| # Restarted Jobs | 125,461 (1.4%) |
| # Restarted Logs | 61,783 (2.1%) |
| # Restarted Repositories | 22,345 (13%) |
| # Restart Passed builds | 9,707 (17%) |
| # Restart Failed builds | 27,006 (48%) |
| # Restart Errored builds | 17,621 (31%) |
| # Restart Canceled builds | 2,186 (3.9%) |

**Table 3: Overview of TravisTorrent metrics.**

| Metric Name | Value |
|---|---|
| Start Study | 2011-08-29 |
| End Study | 2016-08-31 |
| # Builds | 702,922 |
| # Jobs | 3,734,303 |
| # Repositories | 1,272 |
| # Languages | 3 |
| # Passed builds | 524,017 (75%) |
| # Failed builds | 124,821 (18%) |
| # Errored builds | 55,907 (8.0%) |
| # Canceled builds | 2,394 (0.3%) |
| Avg. # builds per repo | 553 |
| Avg. # jobs per repo | 2,936 |

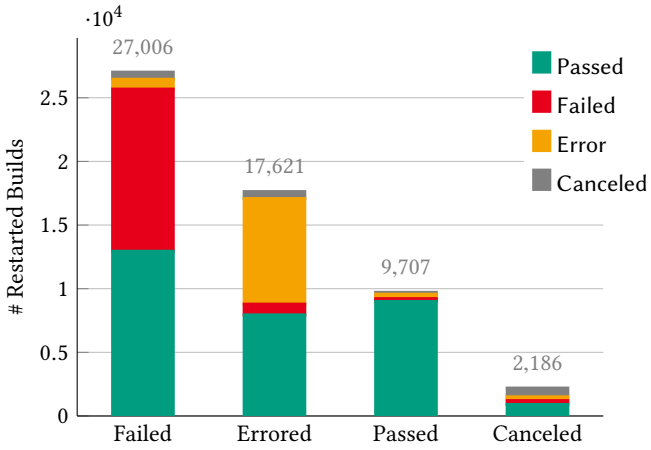


**Figure 3: Evolution of the build status between original builds and restarted builds.**

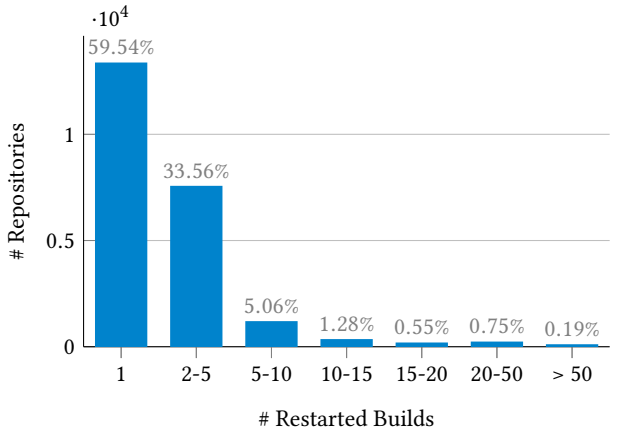


**Figure 4: Distribution of the number of repositories versus the number of time the repositories restart a build.**

the numbers of restarted builds; and the stacked colors represent the state of the restarted builds. We observe that 46.77% of the failing/errored builds pass post-restart. This high percentage of state changes between the original and the restarted builds surprised us, because it indicates that builds are flaky; it also indicates that developers are reasonably adept at identifying such flaky builds. In our study, we observed that 0.80% (46.77% of 1.72% of the restarted builds) of all Travis CI builds in our dataset are flaky (and recall that this is a lower bound). We study the cause of flakiness in greater depth in the third research question (see Section 4.3).

In total, 171,057 repositories have at least one restarted build, which represents 13.06% of all the repositories that triggered a build during our study period. Note that most repositories that include restarts only restarted a few builds. This is confirmed by Figure 4, which presents how frequently projects restart builds. This chart shows that 59.54% of the repositories restart only one build, and 33.56% restart between 2-5 builds. Restarting a build is, overall, a rare operation in the development workflow. However, some projects restart builds far more frequently. For example, the project getsentry/sentry[6] restarted 231 builds (out of 1,613) during our study period. This represents a restart rate of 14%, much higher than the average that we observed. One possible reason for this this high number of restarted builds is a high number of flaky tests; indeed, we found 33 pull requests to this project that mention flaky tests.

**Answer to RQ1. How frequently do developers restart builds? How often do restarts identify flaky behavior?** We observe that developers indeed restart builds. In our study, we observe that 1.72% of the builds are restarted. However, restarting a build is not a common task, and most of the projects only restart one to five builds during the studied time period. Interestingly, 46.77% of the restarted builds change their failing state to a passing state after the restart. This suggests that the restarted builds suffer from flaky behavior.

[6] https://github.com/getsentry/sentry

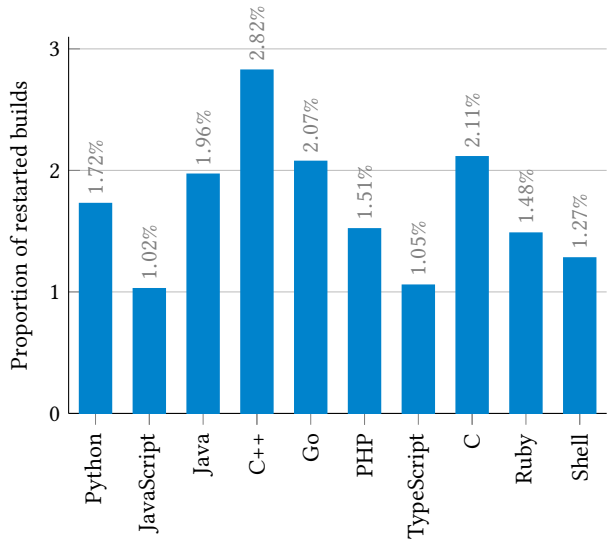


**Figure 5: The restarted rate per language.**

## 4.2 RQ2. Projects that restart builds

In this second research question, we analyze the general characteristics of the projects that restart builds. The goal is to identify patterns that differentiate the restarted builds from the non-restarted ones. We analyze: (1) the programming languages, (2) number of builds triggered by the repositories, (3) execution time, and finally (4) popularity of the repository (number of stars).

Figure 5 presents the restarting rate per language for the ten most frequent languages in Travis CI. JavaScript and TypeScript stand out. Only 1.02% of the JavaScript builds, and 1.05% of the TypeScript builds are restarted, whereas the other languages have build restart rates between 1.27 and 2.82%. To seek to explain this difference, we analyze the distribution of build state between the languages (see Figure 6), the number of builds for each languages (see Figure 7) and the build execution times Figure 8). This analysis is based on the complete Travis CI build history of 171,057 projects, which represents a total of 75,139,295 builds.

Figure 7 presents the success rate for the top 10 programming languages. The boxes with vertical lines present the success rate of the projects that did *not* restart builds in the studied time period. The boxes without vertical lines are the success rate for the projects that have at least one restarted build. A higher success rate for JavaScript and TypeScript would explain the lower number of restarted builds. However, as this figure attests, we did not observe any major differences between the success rate of JavaScript and TypeScript and the other languages.

Figure 7 presents a box chart of the median number of Travis CI builds per project for each language. The gray box presents the value for the projects that do not have a restarted build, and the blue bar presents the value for the projects that have at least one restarted build. We also did not identify a significant difference between JavaScript/TypeScript and other languages.

Finally, Figure 8 presents box plots of the median build duration for each language. The gray box presents the value for the projects that do not have a restarted build, and the blue bar presents the value for the projects that have at least one restarted build. Here, we do observe a difference between JavaScript and TypeScript and other languages: JavaScript and TypeScript builds are substantially faster than those in other programming languages. A shorter build execution time typically suggests that the builds are less complex than those for other languages and, therefore, likely less prone to producing unexpected behavior requiring restarts.

**Table 4: Number of stars of 21,775 projects that have restarted builds compared to the 153,666 that do not have a restarted build. The number of projects differs from Table 1 and Table 2 because we lack data on the repositories that are no longer accessible since initial data collection.**

| Metric | Restarted | Non-Restarted |
|---|---|---|
| # Stars | 8,256,003 | 20,837,112 |
| Average | 379 | 136 |
| Median | 3 | 1 |

Turning to project popularity: Table 4 presents the number of stars for the projects with a restarted build compared to the other projects. We observe that the projects with restarted builds are more popular than the others. They have, on average, 379 stars, with a median of 3 stars. The other projects only have an average of 136 stars and a median of 1.

Based on Figure 6, Figure 7, Figure 8 and Table 4, we observe a difference of behavior of the projects that have restarted builds. These projects have a lower build success rate, a much higher number of builds, a much higher build execution time, and are more popular than the projects that do not have restarted builds. Moreover, those observations are consistent for the top 10 languages, without exception. We interpret those differences as the projects with restarted builds are more complex, suggesting perhaps they have more experienced developers that are more likely to use advanced features such as build restart.

**Answer to RQ2. Do the projects that restart builds have the same characteristics as other projects that use Travis CI?** We observe that some languages are more likely to be associated with build restarts. In particular, we observe that JavaScript and TypeScript projects only restart around 1% of the builds where C++ projects restart 2.82% of the builds. Our hypothesis to explain this difference argues a relationship between programming language and build process complexity. Simple and faster builds, such as most JavaScript and TypeScript builds, are restarted less frequently than C++ builds, for instance. We also observe that, on average, projects that restart builds differ from projects without restarts in terms of success rate, number of builds, execution time, and popularity. Our interpretations of these results are that more complex and more experienced projects restart builds more often.

## 4.3 RQ3. Causes of Restarted and Flaky Builds

In the third research question, we study the potential reasons that lead developers to restart the builds. Due to the large scale of our study, it was not possible to manually annotate all restarted builds.

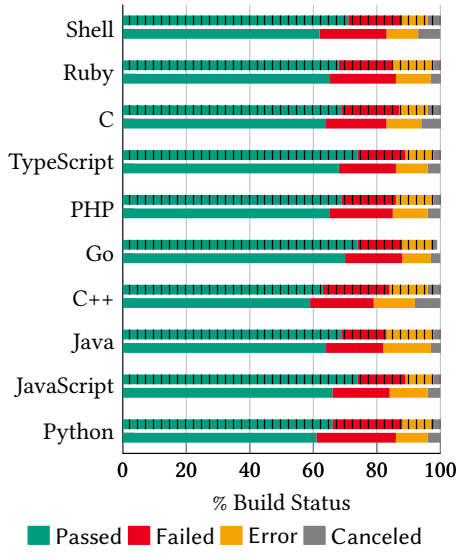


**Figure 6: Comparison of the distribution of build status between the projects with restarted builds (plain box) vs. the other projects (boxes with vertical lines).**

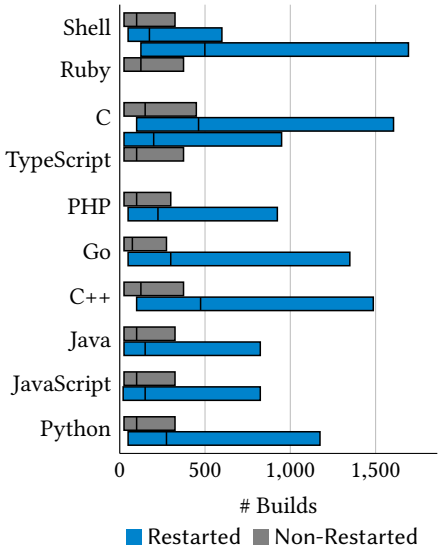


**Figure 7: Box plot of the number of builds for the projects that have restarted builds (blue box) vs. the projects that do not have (gray box). It shows that the projects with restated builds have more builds that the other projects.**

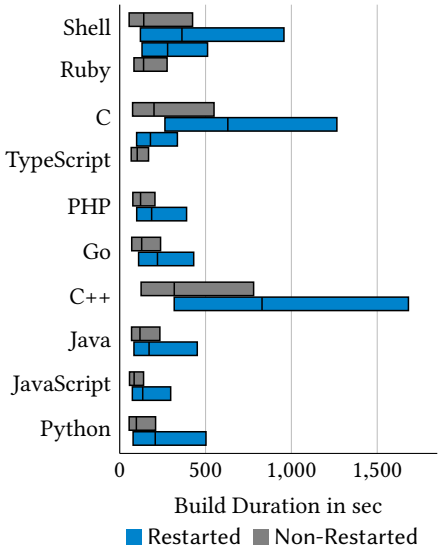


**Figure 8: Box plots of the execution time of projects that have restarted builds (blue box) vs. the projects that do not have (gray box). It shows that the execution time of projects that have restarted builds are much higher.**

Therefore, we extracted, using 93 regular expressions, failure reasons from the logs of restarted builds. We were able to extract failure reasons for 19,730 builds. We manually analyzed a sample of the passing builds that were restarted, to further try to surface reasons why developers would restart a passing build.

Figure 9 presents the ten most frequent types of failure in the execution logs. Each bar presents the number of jobs that contains the associated type of failure. Each color represents the state of the build after the builds have been restarted. For example, it shows that 8,384 builds failed because of test failures, and after the restart 4,586 of those builds are now passing.

We observe that the most frequent cause of build failure is test failure. This is followed by failures related to the Travis CI environment, i.e., a build timeout, or no log output for 10 minutes, which has been observed by prior studies as well [8, 9], These results are interesting, because they imply that an important number of restarted builds are restarted for reasons that are related to the source code of the project and not only because of CI environment problems. This confirms prior observations [8].

Moreover, it is interesting to observe that builds that are restarted in response to test failures have a lower restart success rate, i.e., the original build was failing and it is now passing after the restart, as compared to the other type of failures. It indicates that developers have more trouble determining which builds to restart when the failures are related to tests. The other failure types have a much higher success rate of restarts, achieving more than 78% of successful restart for the `Error install dependencies` failure type. This type of failure could, therefore, be automatically restarted with a minimal cost for the CI service.

We also did not expect to see such a large number of passing builds restarted. Offhand, there is no obvious reason for a developer to restart a passing build. However, 17.17% of the restarted builds are passing builds that are restarted, and 92.62% of the time, the passing builds continue to pass after the restart. We investigated this case, but unfortunately, did not succeed in identifying the reason behind this practice. We looked at the type of project, execution time, the difference between the execution times of the original and restarted builds, the Travis CI configuration, if the restarted passing builds are more related to pull requests. However, we did not succeed in identifying any specific reason that would explain why developers restart passing builds — this remains an open question.

**Answer to RQ3. Why do developers restart builds?** We observe that failing test cases are the main reason that developers restart a build. It is followed by failures that are inherent to the Travis CI environment, such as execution timeout. We also observe that the success rate of restarted builds largely varies between the causes of restarts. A build that is restarted after an error during a dependency installation is more likely to succeed than one that is restarted in response to a failing test case. Further studies should also focus on restarted passing builds, in order to understand developer's goal.

## 4.4 RQ4. Impact of the Restarted Builds

In this final research question, we explore the impact of restarted builds on the development workflow.

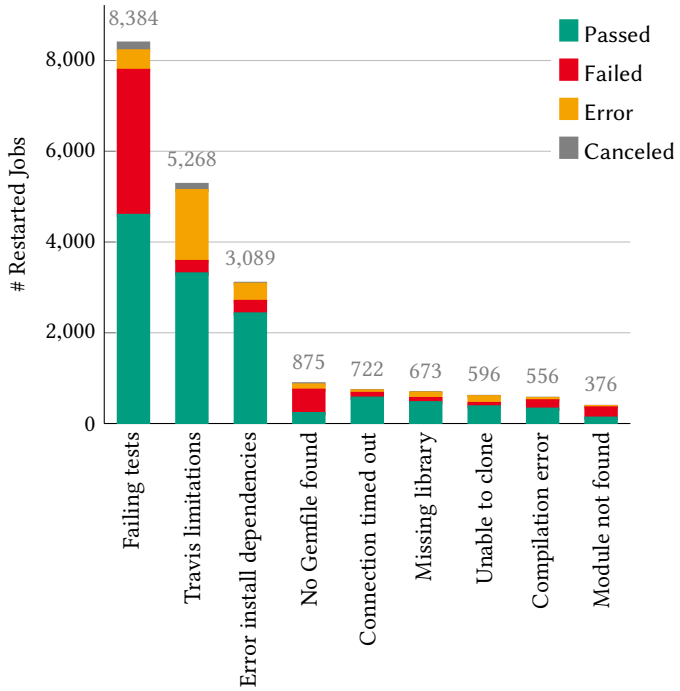


**Figure 9: Reasons of failures of the restarted builds. The state of the restarted build is represented with by the colors, green for passing, red for failure, orange for error, and gray for canceled.**

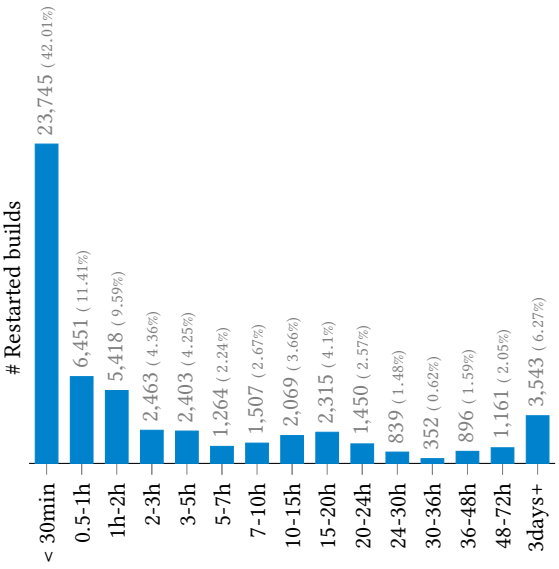


**Figure 10: Amount of time between the original builds and the restarted builds.**

*4.4.1 Developer workflow.* We first look at when, temporally, developers restart builds relative to the original build. Figure 10 shows the distribution of the amount of time separating the beginning of the original build and the beginning of the restarted build. We see that 42.01% of the builds are restarted in less than 30min, and 63.01% of the builds are restarted within 2 hours. This suggests that a developer saw that a build is failing, analyzed the failed build, and decided to restart it quickly after the end of the original execution. By implication, this means that developers are either watching the build as it executes, or must stop what they are doing to analyze the build and restart it. This implies that the developer has to switch between potentially complex tasks, as observed by Czerwinski et al. [4], and also that they are interrupted, a source of stress, higher frustration, time pressure and effort [12]. This is particularly an issue when a build is flaky, because these are ultimately unnecessary interruptions that lead to longer wait times and more interruptions.

*4.4.2 Pull Requests.* We now analyze the impact of restarted builds on pull requests. A pull request is a code modification (one or more commits) proposed by a developer which is typically reviewed before being merged into the project. To study the impact of restarted builds, we look at the merging time of the pull requests. The merging time is the amount of time between the pull request is opened until it is accepted and merged into the project source code.

Figure 11 presents box plots of merging time of all pull requests that executed a build during our study. The $y$ axis shows the relationship between the author of the pull request and the project: `None` indicates that there is no relationship between the author and the project, e.g., the author never contributed to it; `Contributor` means that the author contributed to the project at least once previously; `Member` means that the author is part of the project organization; `Collaborator` means that the author has been added as collaborator by the project owner; and finally, `Owner` means that the author of the PR is the owner of the project itself. Each relationship type is divided in three: the pull requests from projects that do not have any restarted builds (gray box), the pull requests from project that have at least one restarted build, but not in the current pull request (orange box), and the pull requests that have at least one restarted build (blue box). The $x$ axis presents the amount of time requires to merge the pull request.

Figure 11 shows that the merging requires substantially more time when the pull request has a restarted build (blue box). This difference cannot be completely explained by a difference of project characteristics , since the difference between the projects that do not have a restarted build and the projects that have a restarted build, respectively the gray and orange boxes, is much smaller. The median time of the pull request with a restarted build is **11x** larger than the median time of projects that do not have restarted builds. It brings the median time of merging time from 4h to 48h. The merging time is **3x** larger compared to the projects that have at least a restarted builds, which takes the median merging time from 16h to 48h.

The difference in merging time can be explained by several potential factors. The build has to be reexecuted, which requires time to reexecute the build along with the time required for a developer to manually initiate the build restart. Moreover, only project members (if they have permission on the specific project), collaborators, and owners have sufficient permission to restart a build. A contributor therefore needs someone else to restart the build for them.

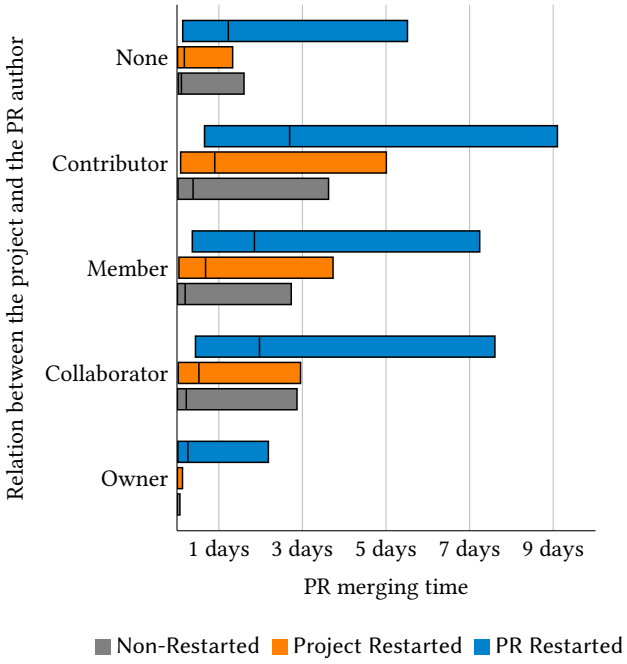


**Figure 11: Box plot of the pull request merging time. The $y$ axis presents the relationship between the project and the author of the pull request. The $x$ axis contains the amount of time requires to merge the pull request. The gray boxes are for the PR in projects that do not have restarted build. The orange boxes are the PRs that are from projects that have at least one restarted build. The blue boxes are the PRs that have restarted build.**

A second factor is that a restarted build can also start a discussion related to the flaky behavior of the build. For example, in pull request #13477 of keras-team/keras project,[7] the author and a member of the project discussed what to do with a flaky test. This discussion takes time, and therefore slows down the merging of the pull request.

**Answer to RQ4. What is the impact of flaky builds on development workflow?** We observe that the restarted builds have an impact on the development workflow. We firstly show that developers restart builds shortly after the build failed. This indicates that developers are waiting for the results or stopping their current task to analyze and restart the builds. Secondly, we observe that restarted build slow down the merging process of pull requests. We measure a slowdown of 11 times compared to projects that do not have restarted build and three 3 compared to projects that have at least a restarted build.

## 5 DISCUSSION

As shown in our results for RQ1, the majority of restarted builds change their status between the original and restarted executions. This indicates that developers are relatively successful in identifying which builds would have a different CI result if they are restarted. We also show that those restarted builds are associated with a considerably slower merging of pull requests. It is, therefore, important to study potential solutions that can be proposed to developers to mitigate this problem. An obvious mitigation strategy would be to restart certain builds automatically; however, there is a trade-off to balance reliability and cost in terms of execution cost and feedback delay, i.e., sending a notification to the developer.

In RQ3, we observe that the main cause of restart is related to failing tests, followed by causes related to network issues, e.g., connection timeout, and Travis CI limitations, e.g., execution timeout. The main cause, flaky tests, is most difficult to identify in terms of whether a build should be restarted. Indeed, detecting test flakiness requires platform specific techniques that require additional instrumentation and execution [2]. This makes it difficult to scale the detection of flaky tests. At best, any such solution is computationally expensive. The causes related to network issues and Travis CI limitations are easier to identify, and present a high restart success rate (see Figure 9). Even in those cases, an automatic restart is not a trivial task because the restart should be carefully planned. It would not make sense, for example, to restart a build that is failing because a website is temporarily out of service; a restart needs to wait for the website to become available. We plan in future work to design a new approach to automatically restart builds on Travis CI and study how developers perceive this type of amortization.

## 6 THREATS TO VALIDITY

*1) Internal: Did we skew the accuracy of our results with how we collected and analyzed information?* Collecting restarted builds requires constant monitoring of Travis CI activity, as it cannot be collected after the fact. During the study period, we faced several server crashes and a disk crash. Those issues forced us to pause data collection for several weeks. This may introduce non-representativeness in the data. However, we ran the collection over a large period of time and collected millions of builds, which reduce this risk.

We manually analyzed logs from failing builds to identify regular expressions to identify causes of failure. There is a risk that we could miss important causes of failure due to the volume of logs that needed to be analyzed. To mitigate this risk, we provide the complete list of regular expressions in our GitHub repository. The failing logs and the extracted failure reasons are also provided to allow future researchers to use and verify our work.

*2) External: Do our results generalize?* A potential threat is that the study period is non-representative due to abnormal activity on Travis CI. We reduce this risk by collecting data during a large period of time, from Tue, 08 Oct 2019 to Thu, 19 Dec 2019. During that period, we monitored the Travis CI and GitHub announcements to ensure that nothing abnormal was happening.

To enable our results to generalize as much as possible, we selected a large and diverse set or projects in various languages, across a wide range of projects on GitHub. However, all of our projects are open source, so we cannot make claims about how our results might generalize to proprietary projects.

[7] https://github.com/keras-team/keras/pull/13477#discussion_r337469914, visited March 27, 2020

*3) Replicability: Can others replicate our results?* To support others in replicating our results, we make our data and infrastructure code available for other researchers and users. The source code of TravisListener is available on GitHub with the DOI: 10.5281/zenodo.3709181. The data is available on Zenodo with the following DOI: 10.5281/zenodo.3601137.

# 7 RELATED WORK

This section presents the related works of this contribution. We focused on two research fields: the works related to Travis CI and the works related to flaky builds.

## 7.1 Related work studying Travis CI

Beller et al. [3] exploits TravisTorrent to study the build behavior of the projects that use Travis CI. Hilton et al. [7] study the use of continuous integration in open-source projects. They show that continuous integration has a positive impact on the projects, and it is used in 70% of the most popular projects on GitHub. Zhao et al. [16] study the impact of Travis CI on development practices. Their main finding is that GitHub pull requests are more frequently closed after the integration of Travis CI. Widder et al. [15] present a study analyzing the reasons projects leave Travis CI. They observed that this phenomenon is related to build duration and repository language. They showed that C# repositories were more likely to quit Travis CI because Travis CI (at the time) did not support Windows virtual machines (support has since been added). On the contrary, repositories that have long build are more likely to continue to use Travis CI. Durieux et al. [5] present a study on 35 million jobs of Travis CI. They analyze how developers are using Travis CI and for which purposes. They find that Travis CI is still mainly used for building and testing applications. Compared to these prior studies, our study targets different aspects of the CI process: We focus on restarted builds and their impact on the development workflow.

Rausch et al. [14] present a study on 14 open-source projects that use GitHub and Travis CI. They analyzed the build failure and identified 14 different error categories. They presented several seven observations, such as "authors that commit less frequently tend to cause fewer build failures", or "Build failures mostly occur consecutively". The difference between this study of build failure and our study is that this related work focuses on the context of the build failure, where we are interested in the cause of failure, e.g., which errors produce build failures, as well as build flakiness.

## 7.2 Related work studying flaky tests

Luo et al. [11] presented the first empirical analysis of flaky tests. They manually analyzed 201 commits from 51 Apache projects that fix flaky tests. Based on this analysis, they produced a taxonomy of the most common root causes of the flaky tests, as well as identify strategies to identify and fix certain types of flakiness. Compared to this work, we focus on flaky builds, studying what causes build flakiness and flakiness impact on development workflow. In future work, we plan to analyze the flaky tests that we mined and verify that Lui et al.'s taxonomy can be applied to them.

Eck et al. [6] study the developer's perspective on the flaky tests. They build a dataset of 200 flaky tests and then ask the original developers that fixed the problem to analyze the nature, fixing effort and the origin of the flakiness. Developers indicated that flaky tests are rather frequent and a non-negligible problem. They only considered that reproducing the context leading to the test failures and understanding the nature of the flakiness are the most important and most challenging needs. In our study, we did not contact the developers to understand their perspective on the flaky tests. However, we provide a dataset and framework that could help the community to further investigate flaky tests.

Bell et al. [2] proposed DeFlaker, an automated technique that identifies flaky tests by running a mix of static and dynamic analyses. Lam et al. [10] presented iDFlakies, a tool that identifies tests that are flaky due to order execution. Compared to those two contributions, our contribution does not aim to detect automatically flaky tests. However, a new benchmark of flaky tests could use our approach of restarted build detection to identify projects that have potential flaky tests.

Micco [13] presents the current state of continuous integration at Google. He indicates that 84% of transitions from passing to failing tests at Google are from "flaky" tests. During our study, we did not observe the same behavior in the open-source project. This probably due does a different nature of the testing that is made at Google compared to testing in open-source projects.

# 8 CONCLUSIONS

In this study, we analyzed the restarted builds from Travis CI, one of the most popular build systems. We collected 56,522 restarted builds from 22,345 projects. We show that 13.06% of the restarted builds are passing. We identify older and more complex projects are more prompted to restart a build. Interestingly, the different programming languages are not subject to the same level of build restart. JavaScript and TypeScript are the languages with the lower restart rate, and C++ is the language with the most frequent request.

We identify that test failures and tests in error are the most common causes of build restarts, followed by the causes that are related to the Travis CI environment, i.e., execution timeout. We also identify that the different failure reasons have a different restart success rate.

Moreover, we observe that a restarted build entails an impact on the development workflow. Developers need to wait for the outcome of the builds and therefore introduce a serious delay when the build needs to be restarted several times. Moreover, we observe that restarted builds introduce a serious delay in pull request merging. Indeed, on average, the pull requests that have a restarted build are 11 times slower than the pull requests from projects that do not have restarted build.

Finally, we discuss the possibility of restarting fully automatically some builds that are failing due to reasons that have a high restart success rate. This automatization would save developers' save time and effort, hence reducing the delay of the contribution of an external contributor.

# ACKNOWLEDGMENTS

This material is based upon work supported by Fundação para a Ciência e a Tecnologia (FCT), with the reference PTDC/CCI-COM/29300/2017. Thomas Durieux was partially supported by FCT/Portugal through the CMU-Portugal Program.

# REFERENCES


[1] Kent Beck. 1999. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., USA.

[2] Jonathan Bell, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tifany Yung, and Darko Marinov. 2018. D e F laker: automatically detecting flaky tests. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, New York, NY, USA, 433–444.

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

[4] Mary Czerwinski, Eric Horvitz, and Susan Wilhite. 2004. A Diary Study of Task Switching and Interruptions. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI âĂŹ04)*. Association for Computing Machinery, New York, NY, USA, 175âĂŞ182. https://doi.org/10.1145/985692.985715

[5] Thomas Durieux, Rui Abreu, Martin Monperrus, Tegawendé F Bissyandé, and Luís Cruz. 2019. An Analysis of 35+ Million Jobs of Travis CI. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Piscataway, NJ, USA, 291–295.

[6] Moritz Eck, Fabio Palomba, Marco Castelluccio, and Alberto Bacchelli. 2019. Understanding Flaky Tests: The DeveloperâĂŹs Perspective. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 830âĂŞ840. https://doi.org/10.1145/3338906.3338945

[7] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. 2016. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, New York, NY, USA, 426–437.

[8] He Jiang, Xiaochen Li, Zijiang Yang, and Jifeng Xuan. 2017. What Causes My Test Alarm? Automatic Cause Analysis for Test Alarms in System and Integration Testing. In *Proceedings of the 39th International Conference on Software Engineering (ICSE âĂŹ17)*. IEEE Press, Piscataway, NJ, USA, 712âĂŞ723. https://doi.org/10.1109/ICSE.2017.71

[9] Adriaan Labuschagne, Laura Inozemtseva, and Reid Holmes. 2017. Measuring the Cost of Regression Testing in Practice: A Study of Java Projects Using Continuous Integration. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 821âĂŞ830. https://doi.org/10.1145/3106237.3106288

[10] Wing Lam, Reed Oei, August Shi, Darko Marinov, and Tao Xie. 2019. iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Piscataway, NJ, USA, 312–322.

[11] Qingzhou Luo, Farah Hariri, Lamyaa Eloussi, and Darko Marinov. 2014. An empirical analysis of flaky tests. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 643–653.

[12] Gloria Mark, Daniela Gudith, and Ulrich Klocke. 2008. The Cost of Interrupted Work: More Speed and Stress. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI âĂŹ08)*. Association for Computing Machinery, New York, NY, USA, 107âĂŞ110. https://doi.org/10.1145/1357054.1357072

[13] John Micco. 2017. The State of Continuous Integration Testing@ Google.

[14] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 2017. An empirical analysis of build failures in the continuous integration workflows of Java-based open-source software. In *Proceedings of the 14th international conference on mining software repositories*. IEEE Press, Piscataway, NJ, USA, 345–355.

[15] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2018. IâĂŹm Leaving You, Travis: A Continuous Integration Breakup Story. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR âĂŹ18)*. Association for Computing Machinery, New York, NY, USA, 165âĂŞ169. https://doi.org/10.1145/3196398.3196422

[16] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 2017. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, Piscataway, NJ, USA, 60–71.