Studying the Unfulfilled Promises of Continuous Integration

by

Taher Ahmed Ghaleb

A thesis submitted to the
School of Computing
in conformity with the requirements for
the degree of Doctor of Philosophy

Queen's University
Kingston, Ontario, Canada
September 2021

Abstract

Continuous Integration (CI) is a software practice that allows software developers to generate software builds automatically and more frequently. When adopting CI, developers anticipate quick and trustworthy feedback about software builds. However, due to configuration or environmental issues, CI might introduce overhead to the software development process. For example, a misconfiguration of CI can make builds run for too long (i.e., long build durations). In addition, CI builds can undergo unexpected breakages (i.e., errors or failures) due to environmental factors that are outside the ability of developers to control. As a consequence, developers might need to wait for builds to pass before engaging in other development activities. This dissertation conducts four empirical studies to understand the reasons behind the unfulfilled CI promises: (1) a study of the factors associated with long CI build durations, (2) a study of noisy (development-unrelated) CI build breakages and their impact on observations of prior research, (3) a study of the interplay between the durations and breakages of CI builds, and (4) a study of the considerations for adopting caching in CI builds. Overall, the research conducted in this dissertation highlights the best practices that help mitigate the burden of adopting CI in software projects and, hence, acquire the prospective benefits of CI.

Acknowledgments

First of all, I would like to sincerely thank Almighty Allah for giving me the ability and strength to accomplish this dissertation, which I dedicate to the memory of my deceased father, Ahmed, who always believed in my ability to be successful in computer science and academia. Father, you are gone but your belief in me has made this journey possible.

I would to express my deepest gratitude to my supervisor, Dr. Ying (Jenny) Zou, for her mentorship, guidance, and insights. Dr. Zou has provided me with an environment that helped me grow into the researcher that I am today. She has always shown immense understanding and empathy when I encountered family issues during my PhD. I would also like to thank my examining committee, Dr. Abram Hindle, Dr. Bram Adams, and Dr. Jianbing Ni, for dedicating the time to provide me with insightful feedback to improve my dissertation. I would also like to thank the great group of researchers at the Software Evolution & Analytics Lab (SEAL), including Daniel Alencar da Costa, Safwat Hassan, Stefanos Georgiou, Ehsan Noei, Yu Zhao, Mariam El Mezouar, Guoliang Zhao, Osama Ehsan, Omar El Zarif, Maram Assi, Aidan Yang, Nikhil Reddy, Jieyeon Woo, Tanghaoran Zhang, Hamza Sherik, Chunli Yu, Zitong Su, and Bihui Jin.

Exclusive thanks to my mother, Badriah, for being my first teacher in life, and for giving me enough love, support, and prayers to last a lifetime and beyond. My deepest gratitude to my wife, Khawla, who was my closest supporter and the backbone throughout this journey. Khawla, this achievement would not be possible without you. To my son, Yamen, and two daughters, Yumna, and Bayan, you have made my PhD life more colorful, challenging, and exciting. Thank you for teaching me patience and bringing me joy. Special thanks and appreciation to all my brothers, Aref, Waleed, and Mohammed, and sisters, Hayat, Najla, Arwa, and Somaya, for their ceaseless and lovely support. Finally, I am grateful to all my teachers, colleagues, friends, neighbors, and relatives who have been encouraging me and wishing me success throughout my graduate studies.

Statement of Originality

I am the primary author of this dissertation. The research papers published in the context of this dissertation are co-authored with Dr. Ying Zou, Dr. Ahmed E. Hassan, and Dr. Daniel Alencar da Costa. Dr. Ying Zou supervised all the research work conducted in this dissertation. The co-authors have contributed to the success of my work through valuable feedback and suggestions to improve the research quality.

The publications related to this dissertation are listed below.

- **An Empirical Study of the Long Duration of Continuous Integration Builds (Chapter 3)**, T. A. Ghaleb, D. A. da Costa, Y. Zou, in Empirical Software Engineering (EMSE), Springer, vol. 24, no. 4, pp. 2102-2139, 2019.

- **Studying the Impact of Noises in Build Breakage Data (Chapter 4)**, T. A. Ghaleb, D. A. da Costa, Y. Zou, Ahmed E. Hassan, in IEEE Transactions on Software Engineering (TSE), 2019.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Continuous Integration (CI) is a software development practice that allows developers to automate the generation of software builds more frequently (e.g., daily or even hourly). CI promises to provide development teams with early and trustworthy feedback about software builds. CI maintains a set of phases, including the installation of dependencies, code compilation, unit/integration testing, and build deployment. Developer teams receive feedback from CI service providers as soon as the builds *pass* or *fail*. However, due to configuration or environmental issues, CI promises might not be fulfilled. For example, a misconfiguration of CI can lead builds to take longer to generate (i.e., long build durations). In addition, CI builds can undergo unexpected breakages (i.e., errors or failures) due to environmental factors that are outside the ability of developers to control. As a consequence, adopting CI may introduce overhead to the software development process, since developers would need to wait for builds to pass before engaging in other development activities. This chapter gives background about CI (Section 1.1), followed by the research challenges (Section 1.2), thesis statement (Section 1.3), objectives (Section 1.4), and outline (Section 1.5).

1.1 Background

Building software is the process of automatically transforming software artifacts (e.g., source code) into deliverables, such as executables and libraries [93]. Build systems (e.g., `make` [37] and `ANT`[1]) allow developers to constantly re-build testable artifacts after performing code changes. Maintaining build systems requires substantial effort [5], since these build systems evolve along with software source code. Neglecting build maintenance may lead to a build failure [89]. Thanks to the advent of the Continuous Integration (CI) practices [41], builds can be generated more frequently (e.g., daily or even hourly), which allows the earlier identification of errors.

Continuous integration (CI) is a development practice that automates the builds of software projects every time a team member submits changes to a central code repository [41]. CI increases the frequency of code integration, testing, and packaging. Smaller and more frequent commits can help developers to reduce the debugging effort and keep track of the development progress. Hence, CI encourages developers to break down development tasks into smaller activities to improve software quality and to reduce any potential risks [30].

CI emerged in the eXtreme Programming (XP) community [12], the most widely used agile methodology. Nowadays, CI is broadly adopted in the pull-based development communities, such as GitHub, GitLab, and Bitbucket. As reported by Bernardo et al. [16] and Vasilescu et al. [103], CI enables development teams to address more pull requests than before. CI automates the build process, including unit and integration tests, under a unified infrastructure (i.e., Web frameworks and build

[1]http://ant.apache.org

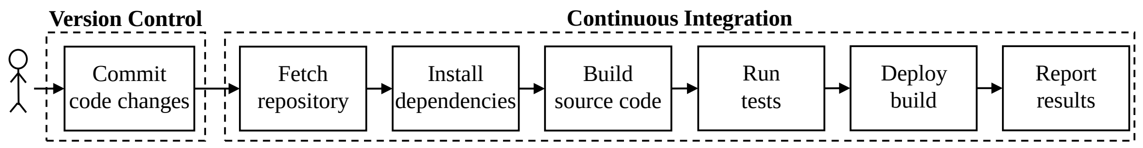


Figure 1.1: The CI build life-cycle

tools), which frees developers from the burden of maintaining their own local integration environments. The unified structure prevents the "*it works on my machine*" syndrome before projects are deployed [72]. Private organizations and open-source development communities may provide built-in CI services to their users. However, maintaining CI infrastructure can be a burden, since the infrastructure needs to scale up to deal with the increased load over time. Therefore, instead of investing in on-site CI infrastructure, organizations use cloud-based CI services, such as Travis CI,[2] CircleCI,[3] and AppVeyor.[4] Such CI providers offer scalable CI services that can be adopted by private companies and open-source development communities. Prior to August 2019, GitHub users were required to use one (or more) of the available cloud-based CI services. In particular, Travis CI has been the most widely adopted CI service in GitHub and supports 30 programming languages. Every GitHub repository can be configured to use Travis CI to automatically generate CI builds. Travis CI provides a free (restricted) service for open source projects and a paid service for private projects. Travis CI offers various subscription plans for both open source and private projects based on customer needs.

[2]https://travis-ci.org
[3]https://circleci.com
[4]https://appveyor.com

1.1.1 The CI build life-cycle

CI has a well-defined life-cycle when generating builds. Fig. 1.1 depicts the main phases of the CI build life-cycle. A CI build is normally triggered once a developer pushes a commit or submits a pull request to a remote repository in a version control system. A CI build can also be (re)started manually. The CI building process starts by cloning the remote repository into the CI server. After installing all the required dependencies, the production code is built, followed by running the unit and integration tests. Finally, if all the previous phases are successful, the build is deployed and the development team is notified with the results. If any of the phases fails, the building process stops and feedback is sent to the development team.

Travis CI maintains a customizable CI build life-cycle. Every programming language has a default CI build configuration in Travis CI. However, developers can customize such configuration by modifying the Travis CI configuration file (i.e., *.travis.yml*) that is located in the root directory of every repository that adopts Travis CI. As shown in Figure 1.2, builds in Travis CI implement all the CI build phases in three main steps: *install*, *script*, and *deploy* phases. Each main build step consists of more than one sub-step that can be configured by developers. In the *install* phase, the remote repository is fetched and all the dependencies are installed. In the *script* phase, the software is built and tests are run. In the *deploy* phase, the software is packaged and deployed to a continuous deployment provider. Developers can define the preferred build environment(s), test commands, and dependencies. In addition, Travis CI allows to write custom instructions and options that are run *before*, *after*, or *in-between* the *install*, *script*, and *deploy* phases. The building process in Travis CI may be interrupted at any point due to an error, a failure, or a manual cancellation.

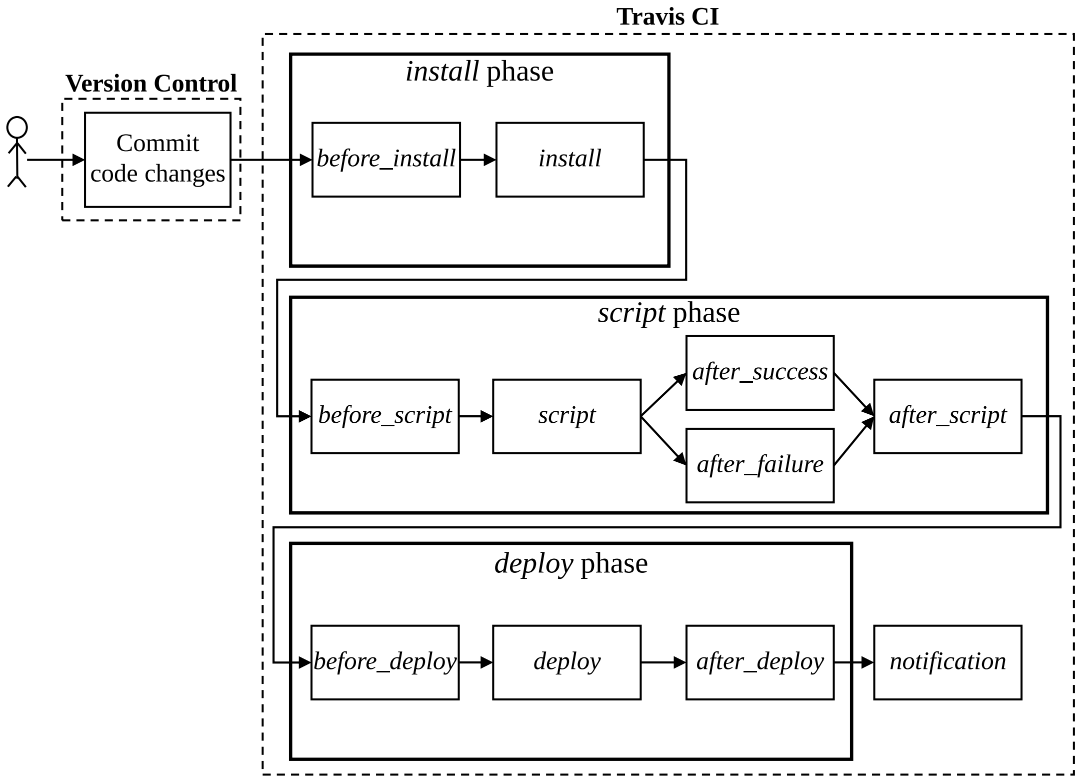


Figure 1.2: Build life-cycle in Travis CI

A build is considered successful if it passes all CI build phases. However, The deploy phase is optional and does not impact the build result.

1.1.2 CI build triggering events

Builds in CI can be triggered by different events. CI builds are often triggered when code changes are committed to the remote repository. In addition, developers can trigger CI builds without introducing new commits. We list below the different events in which CI builds (of Travis CI, in particular) can be triggered:

- **Pushed commits:** Builds are typically triggered when developers push direct commits to the repository.

- **Pull request:** Builds are typically triggered whenever a developer opens a pull request on a certain project.

- **API:** Developers can trigger a build by sending a post request directly to the Travis CI API.

- **Cron:** Developers can schedule builds to be triggered at regular intervals (e.g., every night) regardless of whether a commit has been pushed to the repository.

- **Restart button:** Developers can manually restart previously-triggered builds using a *restart* button on the build page.

1.1.3 CI build matrix and status

A build matrix allows builds to have a combination of configuration options (i.e., different integration and runtime environments). For example, a build matrix of a Ruby project may be composed of runtime (e.g., `rvm`: `1.9.3`, `jruby`, and `ree`), environments (e.g., `mongodb` and `mysql`) and gemfile (e.g., `gemfiles/Gemfile.rails-3.1.x` and `gemfiles/Gemfile.rails-edge`). This build matrix has a total of 12 (i.e., $3 \times 2 \times 2$) build jobs. Developers can also exclude some combinations or mark jobs as *allow failure*. Each CI build job runs independently from the other build jobs. Build jobs can run in parallel or staged to run one after another. A breakage in one build job means that the overall build is broken. However, marking a CI job as *allow_failures* leaves the CI build status unaffected when that job is broken. CI build (and jobs) can have one of the following status indicators (we refer to builds that are either *errored* or *failed* as a *broken* builds):

- **Passed**: A CI build job is considered *passed* if all build phases of that job are successful. A CI build passes if all its jobs pass.

- **Errored**: A CI build job is considered *errored* if errors occurred before entering the *script* phase, i.e., during the installation phases (e.g., while cloning a repository or installing a dependency). A CI build is *errored* if at least one of its jobs is *errored*.

- **Failed**: A CI build job is considered *failed* if something wrong occurred during the *script* phase (e.g., compilation errors or test failures). A CI build is *failed* if at least one of its jobs is *failed* with no *errored* jobs.

- **Canceled**: The *canceled* build status indicates that the build was manually stopped from running.

1.1.4 CI caching

On *December* 5th, 2013, Travis CI introduced a built-in mechanism for caching software artifacts (e.g., dependencies) to private software projects only.[5] A year later (i.e., on *December* 17th, 2014), caching was officially announced to be available to public (open source) software projects.[6] However, caching for open source projects was restricted to running builds under a container-based infrastructure. On *May* 3rd, 2016, such a restriction has been removed and CI caching has become available to builds that run under any infrastructure.[7]

To enable caching, developers need to configure CI builds to determine which artifacts to cache. This can be done by adding the `cache:` directive to the build configuration. Although developers are allowed to cache any directories from the project repository, Travis CI recommends caching dependencies or any artifacts that

[5]https://blog.travis-ci.com/2013-12-05-speed-up-your-builds-cache-your-dependencies
[6]https://blog.travis-ci.com/2014-12-17-faster-builds-with-container-based-infrastructure
[7]https://blog.travis-ci.com/2016-05-03-caches-are-coming-to-everyone

require installation from external servers [98]. Developers can also execute commands before uploading the cache to the CI server using the `before_cache:` directive. For example, developers can remove temporary files (e.g., log files) that may be generated while installing dependencies from the cached directories.

The first build after enabling CI caching uploads all specified directories to the CI cache. In the subsequent builds, such directories are downloaded from the cache rather than the original sources and only the files that have been changed are re-uploaded to the cache. According to Travis CI, failure to download or upload the CI cache does not mark builds as failed [98]. Developers can clear the cache using the Travis CI API or using the Travis CI web interface. Considering that builds in Travis CI may consist of multiple jobs, each job maintains a separate cache in the CI server.

Other CI services, such as CircleCI and GitHub Actions, also support CI caching. While caching adoption maintain a similar configuration to Travis CI (e.g., specify which contents to cache), the other CI services also allow developers to determine when and where to load the CI cache. Yet, we believe that additional caching configurations would open the opportunity for developers to misuse the CI caching configuration, thus leading to more build issues. Future research should investigate caching in other CI services to assess whether our results hold.

1.2 Research challenges

With the continuously growing numbers of software projects in open source communities, developers may find CI the best way to compile and test every single code change. However, due to the frequency of changing the code and, thus generating builds, in such projects, waiting for CI builds to run or to be fixed if broken can introduce overhead to the software development process [48, 84, 86]. Besides the increased energy costs of running CI builds [49], developers may feel disappointed when builds take a long time to be generated [48]. In addition, when CI builds are broken, developers may need to wait until the build is fixed before performing any further code changes. Such challenges encouraged researchers to investigate ways to reduce the time builds take to generate, identify/predict build breakages, and characterize build configuration issues, as follows.

Challenge 1: ***Builds might take long to run but the reasons are unknown.*** The time invested to build software should be as short as possible, since the frequency of generated builds increases in magnitude [18]. Developer studies suggest that the most acceptable build duration is *ten* minutes [18, 48]. Prior research reported that tests can cause a major delay to the CI building process [14]. To address this issue, previous studies proposed approaches to estimate the build duration [17] or split builds into incremental mini-builds [8]. However, the reasons behind long build durations are still vague. Therefore, it is important to perform an in-depth study of long build durations to help developers understand the actions to reduce the time CI builds take to generate.

Challenge 2: ***Build breakages are not always due to developer changes.*** A build breakage may occur due to several reasons, such as configuration errors, installation errors, compilation errors, or test failures. Much research has been devoted to studying the common causes of CI builds breakages [85, 117], predicting build breakages [61, 76, 85, 107], and investigating the factors associated with CI build breakages [61, 68, 76, 85]. However, prior research has paid little attention to builds that may break due to errors that are unlikely to be related to development activities. For example, a build may break due to environmental factors, such as timeouts, connection resets, or memory allocation errors. Studying build breakages without considering the aforementioned factors may lead to misleading conclusions. It is important for developers and researchers to identify such kinds of build breakages data to avoid any negative impact on their analyses.

Challenge 3: *Fixing build breakages can prolong builds, and vice versa.* Prior research has conducted independent studies of either reducing build durations [8, 48, 86] or build breakages [76, 85, 107]. Despite the valuable insights produced by prior research, little is known about the possible interplay between mitigating build durations and breakages. In particular, it is unclear from prior studies (i) whether actions to reduce build durations would reduce build breakages and (ii) whether fixing a build breakage would lead to longer build durations or vice versa. Therefore, it is important for developers to understand the potential dual or side effects when taking actions to reduce build durations or fix build breakages.

Challenge 4: Caching speeds up builds, but should be configured carefully. CI services offer certain build configurations to help reduce the build duration, such as caching artifacts that do not change frequently [98]. As reported by prior work, the adoption of CI caching helps to speed up builds [34]. However, the caching configuration is not enabled in CI builds by default, i.e., developers need to manually configure builds to specify which artifacts to cache. Yet, the challenges (e.g., maintenance effort) and issues (e.g., unexpected failures) that might arise when adopting CI caching remain unexplored. Studying the current practices, challenges, and issues of CI configurations, such as caching, can help developers avoid any unnecessary development overhead when adopting CI caching.

To tackle the four aforementioned challenges, this thesis conducts four empirical studies of build durations, breakages, and configurations. First, we explore the most important development and configuration factors that are associated with long build durations and provide developers with actionable recommendations to reduce the time builds take to generate. Second, we evaluate the impact of CI build breakages that can taint the conclusions of prior research. Third, we investigate the interplay between build durations and build breakages. Fourth, we study the CI caching configuration to understand its adoption rate, its maintenance activities, its impact on CI builds, and its commonly reported issues.

1.3 Thesis statement

CI fails to fulfill its promises to software developers due to challenges in handling build durations, breakages, and configurations. Prior research has studied the barriers that developers face when adopting CI in software projects. Yet, little is known about the actions that developers should take to mitigate CI building problems. This thesis sheds light on CI problems that can impede developers from engaging in other development activities. In particular, we hypothesize that (a) the reasons behind CI long build durations are beyond the common wisdom factors, e.g., project size and test density, (b) CI build breakages are not always caused by developers, (c) there are trade offs between fixing CI build breakages and reducing build durations, and (d) CI caching, though useful, can cause lead to maintenance and build issues. To investigate these hypotheses, this thesis conducts empirical studies to identify (1) the factors associated with long build durations, (2) the types of development-unrelated build breakages, (3) the dual and side effects between reducing build durations and fixing build breakages, and (4) the drawbacks of adopting CI configurations that are known to improve CI builds, such as caching. Findings of this thesis help (i) developers better understand the best practices to acquire the anticipated benefits of CI; (ii) researchers become more aware of the possible noise and side effects in CI build data; and (iii) CI services provide developers and researchers with more enriched information about the CI building process.

1.4 Thesis objectives

This section outlines the objectives through which this thesis addresses the aforementioned challenges. Figure 1.3 shows a high-level outline of the thesis in which we map the objectives listed below to the upcoming thesis chapters.

Objective I: Studying the factors that are associated with long build durations. Long build durations might not be always correlated with the lifetime of a project or with passed builds. Instead, the build duration can increase as a result of other factors. This study aims to show which factors have significant associations with long build durations. We analyze the proportion of build durations of open source projects to investigate the frequency of projects that suffer from the problem of long build durations. We model long build durations to reveal the most important factors that have direct or inverse relationships with long build durations. This study reports a set of recommendations to developers, researchers, and CI services to help reduce the build duration.

Objective II: Studying the impact of noisy (development-unrelated) build

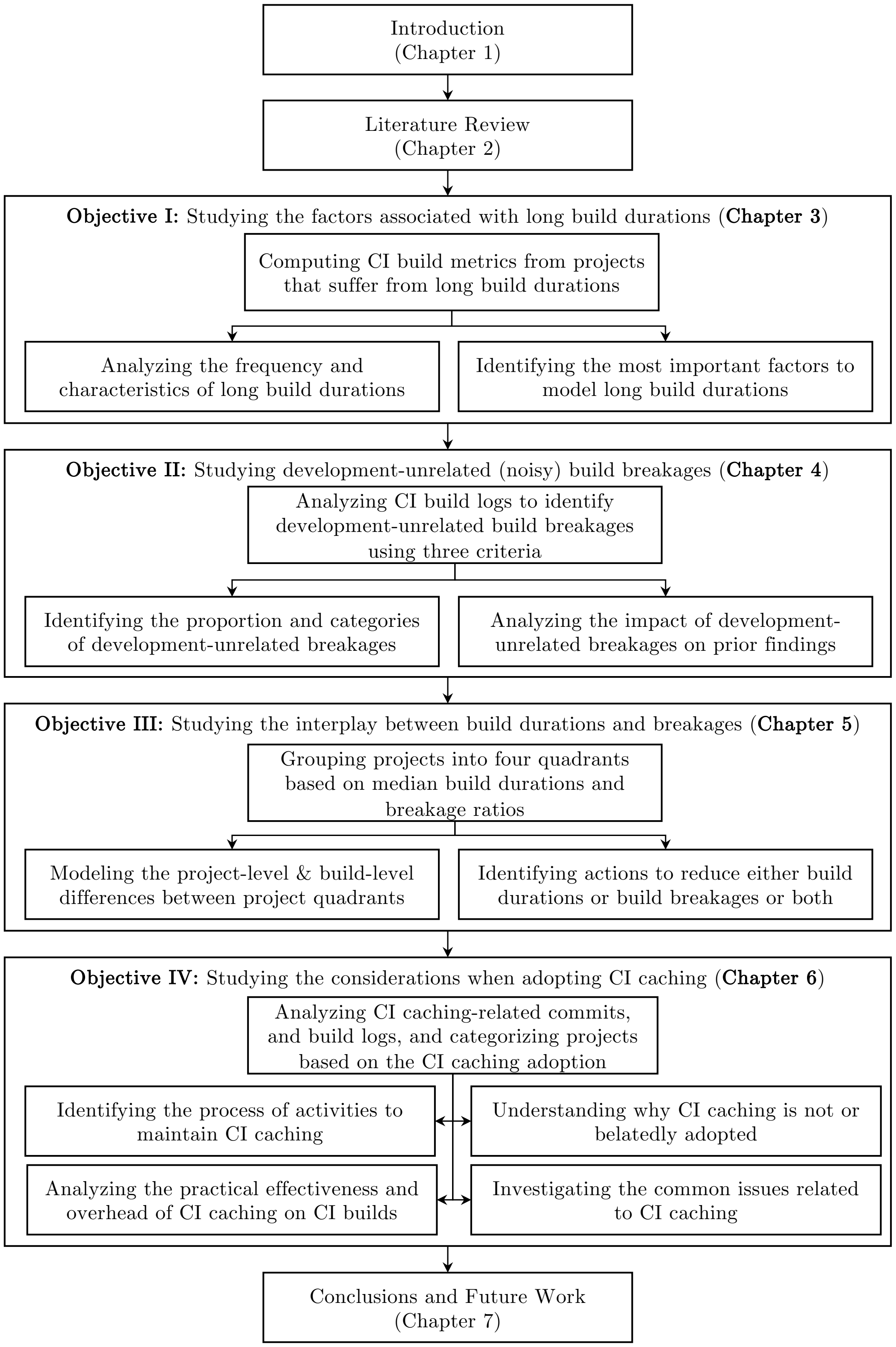

Introduction (Chapter 1)
Literature Review (Chapter 2)
Objective I: Studying the factors associated with long build durations (Chapter 3)
Computing CI build metrics from projects that suffer from long build durations
Analyzing the frequency and characteristics of long build durations
Identifying the most important factors to model long build durations
Objective II: Studying development-unrelated (noisy) build breakages (Chapter 4)
Analyzing CI build logs to identify development-unrelated build breakages using three criteria
Identifying the proportion and categories of development-unrelated breakages
Analyzing the impact of development-unrelated breakages on prior findings
Objective III: Studying the interplay between build durations and breakages (Chapter 5)
Grouping projects into four quadrants based on median build durations and breakage ratios
Modeling the project-level & build-level differences between project quadrants
Identifying actions to reduce either build durations or build breakages or both
Objective IV: Studying the considerations when adopting CI caching (Chapter 6)
Analyzing CI caching-related commits, and build logs, and categorizing projects based on the CI caching adoption
Identifying the process of activities to maintain CI caching
Understanding why CI caching is not or belatedly adopted
Analyzing the practical effectiveness and overhead of CI caching on CI builds
Investigating the common issues related to CI caching
Conclusions and Future Work (Chapter 7)

Figure 1.3: Thesis outline

breakages. It is normal for CI builds to break due to errors or failures while generating the build. However, a build breakage might not be always related to developer activities. This study aims to analyze which builds are broken due to reasons that are unrelated to the latest code changes (i.e., noisy build breakages). We define criteria to identify three types of noise breakages in open source projects, including breakages caused by environmental errors, formerly unfixed errors, and build jobs that are later deemed by developers as noisy. After that, we model build breakages using cleaned (after removing noisy data) and uncleaned datasets. This study reports the impact of such noisy breakages on the findings of prior research.

Objective III: Studying the interplay between build durations and build breakages. Developers might be interested in either reducing build durations or avoiding build breakages. However, reducing build durations can potentially cause unwanted breakages, while avoiding build breakages can also introduce unwanted delays to the building process. This study aims to investigate the dual and side effects of reducing build durations and avoiding or fixing build breakages. We group open source projects into quadrants based on their median build durations and build breakage ratios. We model the differences between the different groups of projects to understand the factors that make projects suffer from long build durations or frequent build breakages or both. This study reports actionable findings of the project-level and build-level factors associated with build durations and breakages as well as the dual and side effects of these factors.

Objective IV: Studying what developers should consider when adopting CI caching. CI caching is known to speed up builds. However, the challenges (e.g., maintenance effort) and issues (e.g., unexpected failures) that might arise when adopting CI caching remain unexplored. This study aims to investigate the adoption rate of CI caching (in particular, Travis CI) in software projects to understand the reasons behind the early or proactive adoption of CI caching and why projects still do not adopt or delay the adoption of CI caching. This study reports a set of considerations that developers, researchers, and software practitioners should take into account when dealing with caching in a CI context.

1.5 Thesis outline

We proceed by introducing a literature review of the related work in **Chapter 2**.

Chapter 3 describes our study of the most important factors to model long build durations. In this chapter, we describe the data collection and processing, including the calculation of build-level metrics that we use to model long build durations. Then, we present the approach we followed to address our research questions that investigate the frequency of long build durations and the different ways to model long build durations. We report our findings and discuss the implications of our results for developers, researchers, tool builders, and CI services.

Chapter 4 describes our study of noisy (development-unrelated) build breakages. In this chapter, we show, through an illustrative example, that looking at the build status provided by CI services can lead to a misinterpretation of which commit and developer broke a build. Then, we describe the criteria we use to analyze three types of noisy build breakages from build logs. After that, we present the approach we use to investigate the impact of noisy breakages on prior research findings. We highlight all the prior findings that do not hold after removing the noise from build breakage data. We provide a set of recommendations to developers and researchers to handle such noise.

Chapter 5 describes our study of the possible interplay between reducing build durations and fixing/avoiding build breakages. In this chapter, we describe our computed project-level and build-level metrics, along with the developer survey we conduct to validate our empirical results. Then, we present several approaches to study the metrics that have dual effects or side effects on build durations and/or build breakages. We report actionable findings to increase the awareness of developers and researchers about the metrics that have dual or side effects on either build durations and build breakages.

Chapter 6 describes our study of the Travis CI caching configuration and the important considerations for adopting caching in CI builds. In this chapter, we describe our data collection of caching-enabled projects and the analysis commit logs and build logs to identify maintenance activities and overhead of CI caching, respectively. Then, we describe our approaches to identify the common reasons and issues behind not adopting CI caching, mine the process of CI cache maintenance, and analyze the benefits and overhead of CI caching on the building process. We report our findings across five research questions and discuss how developers, researchers, and CI services can leverage our findings to increase the awareness of adopting CI configurations.

Finally, **Chapter 7** concludes the thesis and provide directions for future work.

Chapter 2

Literature Review

The goal of this thesis is to explore the reasons behind the unfulfilled promises of CI regarding build configurations, build durations, and build breakages. Research has studied the current CI barriers and propose possible solutions to mitigate such barriers. To achieve the goal of this thesis, it is important to understand the current state of the art in CI research.

In this chapter, we give an overview of the prior research related to the work conducted in this thesis across three categories: (1) the practices and challenges of adopting and configuring CI, (2) CI build breakages, and (3) CI build durations.

2.1 Practices and challenges of adopting and configuring CI

Adopting CI in software projects may not be a straightforward process. Developers need to learn how to configure CI builds, utilize the provided features, and what to expect after CI adoption. Software projects differ in terms of domain, programming language, and team size. Therefore, each project may experience different challenges and obstacles when adopting CI. In this section, we present the research studies that investigated existing challenges and practices of adopting CI in software repositories.

Beller et al. [14] reported that CI is largely and uniformly adopted by GitHub repositories of different most. Vasilescu et al. [102] found that more than half of the projects that adopted CI do not really use it. Research has defined catalogs of patterns and anti-patterns when adopting CI in software projects [30, 31]. However, complying with the CI standards can be challenging for development teams and, as a result, CI anti-patterns may occur. Sidhu et al. [91] performed an association rule mining to discover the usage patterns of CI build configurations. The authors observed that there is no significant configuration commonality between projects that use CI (i.e, CI specifications are broad and diverse). Vassallo et al. [104] found misuses and smells related to CI build configuration.

Wagner et al. [38] studied CI usage practices and found that projects tend to have infrequent commits during the development lifetime. Widder et al. [105] studied the reasons behind CI abandonment. Results have shown that developers abandon CI because of lack of experience, unstable build statuses, and long build durations. Gautam et al. [42] clustered projects that adopt CI based on activity, popularity, size, testing, and stability. The study reported a set of best practices for CI newcomers.

Research has studied the impact of adopting CI on the software development process. Atchison et al. [10] studied projects that adopt CI services and observed a steady growth in the number of builds over time. Stolberg suggests that developers should commit at least once every day [97]. Miller reported that Microsoft developers tend to commit once a day, whereas, due to network issues, off-shore developers commit less frequently [73]. Zhao et al. [115] observed that, after adopting CI, the number of closed issues slows down, while the number of closed pull requests remains relatively stable. They also found that manual code reviews still limit CI capabilities from scaling up. Yuksel et al. [111] reported that projects have an average of 33 commits per day after the introduction of CI. Leppanen et al. [65] conducted interviews with 15 companies and found a higher release frequency (i.e., decreased time-to-market) when continuous deployment was introduced. Rahman et al. [81] reported that CI promises can only be achieved if developers use CI more effectively (e.g., using frequent commits). Rahman and Roy [83] studied the impact of CI builds on the code reviewing process and found that passed builds encourage more code reviews than broken builds.

Bernardo et al. [16] studied how CI adoption has an impact on the pull request (PR) activity. The authors observed that the number of PRs has increased and development teams tend to merge PRs more quickly after adopting CI. Zampetti et al. [112] studied the interrelationship between PRs and CI adoption. The authors observed that, although PRs with passed builds are highly likely to be merged, development teams tend to merge PRs even if they break the build. The authors also reported that CI adoption has introduced more complexity to PRs, since developers need to resolve CI-related issues. Baltes et al. [11] studied projects that adopt CI and also found an increasing number of merged PRs. However, the increased number of merged PRs is unlikely to be due to CI adoption, since the authors found a similar increasing trend of merged PRs in projects that do not adopt CI. Vasilescu et al. [103] found that development teams that use CI discover significantly more bugs and are significantly more effective at merging PRs submitted by core developers. In addition, external developers tend to have fewer rejected PRs with the availability of CI. Yu et al. [110] reported that CI introduces a major latency to merging PRS, since developers need to wait for the feedback from the PR-triggered builds. Abdalkareem et al. [6] studied projects in which developers skip generating CI builds when they commit to remote repositories. The authors found developers are likely to skip CI for commits that update comments/documentation, prepare releases, change code style, or modify metafiles (e.g., the `.gitignore` file).

Research has explored the current practices and challenges of CI adoption. Hilton et al. [49] studied how projects adopt CI and found that CI is heavily used by the popular GitHub projects. The study shows that projects adopting CI release more often and accept pull requests faster. Hilton et al. [48] studied the trade-offs in CI and found that build configuration, long build durations, and frequent build breakages are the main concerns of developers when adopting CI. Pinto et al. [80] studied the developer's perception of CI and found that, despite the advantage of CI to automate builds and speed up locating bugs, developers may not rely on the result of CI builds. Ståhl and Bosch [95] studied CI practices and build breakages and found that developers may sometimes ignore test failures because they know such failures will eventually be fixed later [86]. As a result, ignoring build breakages leads to misinterpretation of the build history and makes it hard to draw correct conclusions. Rahman et al. [82] studied the link between builds with failing tests and the number of crash reports on the Firefox Web browser. The authors found that builds with all tests passing have a median of two crash reports, whereas builds with one or more failing tests are associated with a median of 508 and 291 crash reports for beta and production builds, respectively.

Summary:

Prior research has reported CI challenges and anti-patterns related to build configurations. However, developers are still unaware of the best practices to maintain fast and passing CI builds. In addition, it is unclear whether (mis)using certain CI configurations can lead to a trade-off between build breakages and build durations. In this thesis, we study the configuration factors that have side effects on either build durations of breakages. We also perform an in-depth study of the benefits and drawbacks of the CI caching configuration.

2.2 CI Build duration

CI promises to provide fast feedback to developers about the generated builds. However, a build may take too long due to (i) factors that developers can control or (ii) factors that are outside the ability of developers to control. In this section, we present the research studies that investigated build durations in CI.

Prior research has studied the concerns related to build durations. A study by Rogers [86] reports that a long build duration considerably interrupts the development process. The author suggests strategies to keep complex projects coping with CI, such as accumulating the integration to be performed once a week. It is also argued that every project should maintain a certain limit of build durations and keep up with that. Similarly, Rasmusson [84] studied how feedback and team spirits are negatively impacted when a build takes a long duration. Another study proposed an approach to split large builds into smaller builds and adopt the so-called *incremental builds* [8]. Incremental builds enhanced the overall build duration. However, none of such studies have investigated the factors that may be associated with long build durations.

Brooks [18] highlighted that long build durations affect the development flow and lead to changing the frequency and size of commits. According to reports of expert developers of a private company, a build duration of two minutes is considered optimal, while is acceptable to take up to 10 minutes. Nonetheless, such an assumption cannot be generalized, since builds of complex software systems (e.g., Linux) may require more time to process. On the other hand, building and testing tools may impact the duration of builds, and users have limited control over such tools [74]. Another user study by Hilton et al. [48] shows that developers desire to keep build durations below 10 minutes. The study also reported that developers solely rely on eliminating unnecessary dependencies or tests to reduce the duration of CI builds.

Atchison et al. [10] performed a time-series analysis of the history of CI builds. The authors observed seasonal trends in the number of builds over time. An approach was proposed to estimate the number of builds to be generated in the future. However, the evolution of build durations over time was not analyzed. Beller et al. [14] indicated that the time it takes to run a single test in CI is relatively short (i.e., $<$ 1 minute).

Bisong et al. [17] evaluated which prediction models are best to estimate build durations. However, the duration values used by Bisong et al.were misleading, since they do not represent the actual build duration perceived by developers but rather the cumulative build durations of all jobs. In addition, Bisong et al.did not study the factors associated with long build durations. Wagner et al. [38] analyzed build durations across projects from different languages (i.e., Ruby and Java) and sizes. However, the reasons behind the differences in build durations were unexplored. Moreover, similarly to Bisong et al., the analyses performed by Wagner et al.were based on the cumulative build durations of all jobs rather than the perceived build duration.

Summary:

Existing research lacks empirical evidence of the frequency and trends of long build durations in software projects. In this thesis, we empirically study the evolution patterns of build duration over time. We generate insights on best configuration practices to optimize and maintain an acceptable build duration.

2.3 CI Build breakages

Although CI allows the early identification of errors, build breakages are considered a major concern to developers. In this section, we present the research that has been conducted on identifying breakage types in CI, predicting build breakages, and fixing build breakages.

2.3.1 Types of CI build breakages

CI helps developers to identify the errors in the committed code changes earlier. However, developers may be unaware of the types of build breakages that are expected to occur in CI. In this section, we present the research studies that identified the different types of build breakages in CI.

Existing research has investigated the reasons behind CI build breakages. A study by Beller et al. [14] indicated that test failures are the cause of more than half of build breakages in CI. Rausch et al. [85] studied the causes of errors and failures of CI builds and found 14 common error categories of CI builds. The authors observed that around 30% of the errors occur in the first half of build execution, and that test failures are a major cause of build breakages. Kerzazi et al. [57] studied build breakages generated in a large web company and found that about 18% of the breakages have a potential investment of over $2,000$ man-hours. The authors observed that the most common breakages are caused by missing files in PRs, accidental check-in of experimental changes, missing specifications, and branch merges. Zolfagharinia et al. [117] performed an empirical study of build breakages on 30 million builds from the Comprehensive Perl Archive Network (CPAN). The authors found that the status of CI builds may be impacted by operating systems or runtime environments in which the code should be integrated. The authors found that programming and dependency errors are the main reasons for breakages.

Seo et al. [89] studied the compilation issues as a cause of breakage of over 26 million builds from C and Java projects. The authors found that undeclared variables, methods, and classes are the main causes of compilation errors. In addition, they observed that the time to fix these errors varies widely and developers need tool support to interpret and to avoid such kinds of errors. Similarly, Zhang et al. [114] classified the types of build breakages that are caused by compiler errors. The authors found that compiler errors may occur but do not affect the eventual build status (i.e., errors can be ignored or rerun). Zampetti et al. [113] studied the use of static analysis tools in CI pipelines. The study identified build breakages that are caused due to static analysis and found that the majority of errors are related to (non)adherence to coding standards.

Despite the valuable insights reported by existing studies, little is known about whether build breakages are really caused by the developers who triggered the builds. For example, build breakages can be are caused due to environmental or (unfixed) errors of previous builds.

2.3.2 Modeling CI build breakages

Developers may prefer to know whether the code changes will likely break the build before pushing such changes to the remote repository. In this section, we present the research studies that model build breakages for the sake of identifying the metrics that are associated with build breakages.

Research has investigated the possibility of predicting whether the CI build will pass or break [76, 107]. Xia and Li [107] built 9 prediction models to predict the build status. They validated their models using two scenarios: cross-validation and online. Their models achieved a prediction AUC of over 0.80 for the majority of the projects they studied. The authors observed that predicting a build status using the online scenario performed worse than that of cross-validation, with a mean AUC difference of 0.19. They also observed that the prediction accuracy falls due to the frequent changes of project characteristics, development phases, and build characteristics across different version control branches. Ni and Li [76] used cascade classifiers to predict build breakages. Their classifiers achieve higher AUC values than other basic classifiers, such as decision trees and Naive Bayes. They also observed that historical committers and project statistics are the best indicators of build statuses.

Rausch et al. [85] studied the association between build breakages and 16 process and CI metrics. The authors observed that process metrics (e.g., the complexity of changes) have a strong impact on build statuses, in addition to the historical failing ratio. In addition, they observed that developers who commit less frequently make fewer build breakages than developers who commit daily. Jain et al. [55] performed a causation analysis of build breakages on Java and Ruby projects and found that that larger team sizes make more breakages. The authors also observed that their findings are not sensitive to the differences in programming language or projects. Conversely, Islam et al. [54] found that the sizes of projects and teams are unlikely to be correlated with build breakages. The authors observed that builds are more likely to break because of the building tool and the complexity of changes. Souza et al. [94] performed a sentiment analysis over CI builds to study the potential association between developers' sentiment and build breakages. The authors found that negative sentiment can affect and be affected by the status of CI builds. In addition, they found that developers tend to be more positive when writing about CI services in commit messages.

Existing studies on build modeling and prediction did not consider that build data may contain noisy build breakages. For example, builds may break due to reasons that are unlikely to be related to the activities of developers. For example, a build may break due to timeout or external connection errors. Hence, not excluding noisy build breakages from prediction models may lead to misleading results.

2.3.3 Fixing CI build breakages

CI helps developers to identify the errors in the committed code changes earlier. Due to the diverse types of build breakages in CI, developers may fix such breakages in various ways. In this section, we present the research studies that investigated the practices developers follow to fix CI build breakages.

Wagner et al. [38] observed that build breakages take too long (e.g., days or even weeks) to be fixed. Zhang et al. [114] identified the different ways in which compiler-related breakages are fixed. The authors found that there are common patterns to fix the most frequently occurred compiler errors. Dmeiri et al. [25] proposed an approach that leverages CI to mine pairs of broken-passed builds from Java and Python projects and attempts to automatically reproduce these pairs in DOCKER containers. The authors assumed that passed builds are the fixes for formerly broken builds.

Although prior studies have investigated the fixing practices of build breakages in CI, little is known whether certain breakage fixing practices may consequently impact build durations. In addition, build breakages can be due to environmental errors and, hence, passed builds can be unrelated to the breakages occurred.

Summary:

There is a lack of attention to build breakages that are unlikely to be related to development activities. In this thesis, we investigate the scenarios in which builds may break even if the committed code changes are error-free. Moreover, none of the prior studies has investigated the interplay between build breakages and build durations. In this thesis, we fill such a gap by studying the factors that have side or double effects on build breakages and durations.

2.4 Summary

Much research has been invested to study CI adoption and build performance. Nonetheless, software developers still need to know the actions to take for optimizing CI builds and avoid any unexpected build configuration problems. This thesis highlights the important CI configurations and development practices that help developers act towards optimize CI builds and present the implications for software researchers, tool builders and CI service providers.

Chapter 3

A Study of the Factors Associated with Long Build Durations in CI

This chapter describes our study of the most important factors to model long build durations. Section 3.1 presents the research problem and motivation of our study. Section 3.2 presents the study design of our empirical study. Section 3.3 discusses the results and findings of our studied RQs. Section 3.4 discusses the implications of our findings for developers, researchers, tool builders, and CI services. Section 3.5 describes threats to the validity of our results. Finally, Section 3.6 summarizes the chapter and suggests future work.

3.1 Problem and motivation

When adopting CI, the time invested in building a project should be as short as possible, since the frequency of generated builds increases in magnitude [18]. In such a context, *long* build durations generate overhead to the software development process, since developers would need to wait for a long time before engaging in other development activities [48, 84, 86]. In addition, the energy cost of running CI builds increases as the build duration increases, which is an emerging concern [49]. Hilton et al. [48] found that developers may feel disappointed when builds take a long time to be generated. Such challenges encouraged researchers to study the approaches for reducing the durations of builds. For example, Ammons [8] proposed an approach to split builds into a set of incremental mini-builds in order to speed up the build generation process. Recent user studies suggest that the most acceptable build duration is 10 minutes [18, 48]. Bisong et al. [17] introduced a number of regression models that aim to estimate the build duration. The authors were motivated by the study performed by Laukkanen and Mäntylä [62], which has demonstrated the lack of quantitative analysis on build durations.

Although Bisong et al. [17] studied the build duration, the factors associated with *long* build durations were not investigated. In addition, the models presented by Bisong et al.were not entirely suitable for predictions, since post-build metrics were used in their models (e.g., the number of tests runs, test duration, and CI latency). Moreover, the models were built to estimate the individual durations of build jobs instead of the perceived durations of builds. Nevertheless, the reasons behind *long* build durations are still vague and need an in-depth study. In addition, our exploration of $104,442$ CI builds of 67 projects reveals that 84% of CI builds last more than the acceptable 10 minutes duration [18, 48].

Therefore, we empirically investigate which factors are associated with *long* build durations. This investigation is important to help software engineers to reduce the duration of CI builds. We perform our study using 67 GitHub projects that are linked with Travis CI. To collect our data, we use TravisTorrent [15], which is a publicly available dataset that contains information about CI builds of $1,283$ projects.

Despite the relatively small sample of projects in our study, we should note that our dataset contains well-known and previously studied projects (e.g., `rails`, `jruby`, and `openproject`). Moreover, our study selects projects with high variations of build durations where the problem of *long* build durations may occur. To this end, we select 67 projects that have a build durations Median Absolute Deviation (MAD) above 10 minutes [18, 48]. To capture the different characteristics of the studied projects in relation with *long* build durations, we use mixed-effects logistic models to model *long* build durations.

The goal of this study is to empirically conduct a study (a) analyze the frequency of *long* build durations; (b) investigate the most important factors that may have an association with *long* build durations; and (c) gain insights about CI build configurations and practices that may help developers to mitigate *long* build durations. Based on the above goals, we address the following RQs:

$RQ_{3.1}$: What is the frequency of long build durations?

> We observe that 40% of builds in the studied projects have durations of more than 30 minutes. We discover that build durations do not increase linearly over time in software projects. In addition, durations of *passed* builds are not always longer than the durations of *errored* and *failed* builds.

$RQ_{3.2}$: What are the factors associated with long build durations?

> We classify build durations to be either *short* or *long* based on the *lower* and *upper* quantiles of the perceived build duration. Then, we use mixed-effects models to study the association of various factors with *long* build durations. We observe that, besides common wisdom factors (e.g., project size, build configuration size, team size, and test density), *long* build durations can be explained

using less obvious factors. For example, triggering builds on weekdays or at daytime is most likely to have a direct relationship on *long* build durations. We also find that builds may have shorter durations if they are configured (a) to cache content that does not change often or (b) to finish as soon as all the required jobs finish. Yet, about 40% of the studied projects do not use or misuse such build configurations.

3.2 Study Design

This section presents the experimental setup of our empirical study. We explain how we collect and prepare the data for our studied RQs.

3.2.1 Data Collection

Fig. 3.1 gives an overview of our study. Our study is based on data collected from TravisTorrent [15]. TravisTorrent, in its 11.1.2017 release, stores CI build data of $1,283$ projects. These projects are written in three programming languages: Ruby, Java, and JavaScript. Each entry in TravisTorrent represents a build job. For example, if a build is triggered using x jobs, TravisTorrent records x data entries for that build. All the x entries are recorded using the same build *id* and a different job *id*. Nevertheless, the values of the selected metrics, including build duration values, are the same in all the job entries for a given build. Considering that our study focuses on *long* durations of builds, we keep a single entry for each build *id* and discard the rest of the entries of a build. We keep the information about how many jobs in each build. Each job has its own duration reported by Travis CI. However, given the fact that build jobs can run in parallel on Travis CI, using the sum of durations of all

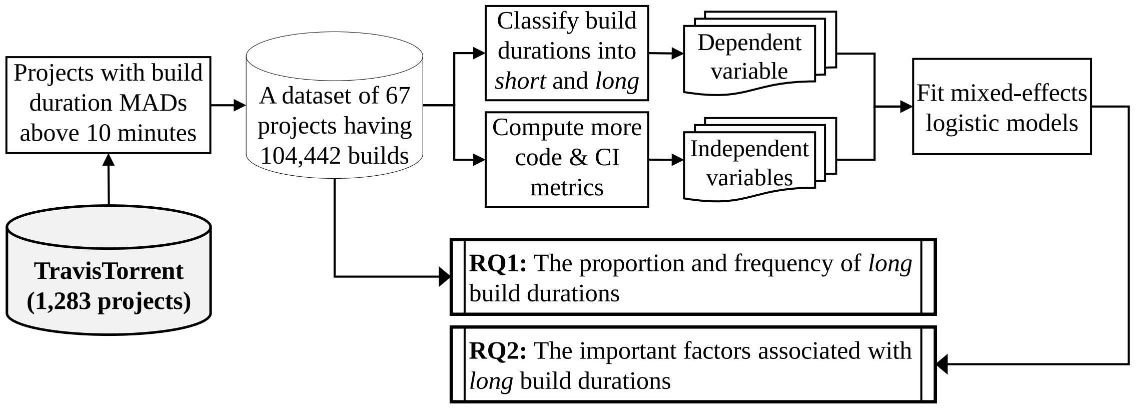

Projects with build duration MADs above 10 minutes
A dataset of 67 projects having 104,442 builds
Classify build durations into *short* and *long*
Dependent variable
Compute more code & CI metrics
Independent variables
Fit mixed-effects logistic models
TravisTorrent (1,283 projects)
RQ1: The proportion and frequency of *long* build durations
RQ2: The important factors associated with *long* build durations

Figure 3.1: Overview of our empirical study of long build durations

build jobs could be misleading (i.e., the perceived build duration is likely smaller than the sum of durations of all jobs). Therefore, we collect the perceived build duration represented by the difference between the time when the build started and the time when the build finished.

We identify a criterion to select the subject projects in our study based on the high variation of build durations. Given that we are interested in studying the factors that are associated with *long* CI build durations, we need to study projects that have higher variation in their build durations. For example, if the variation of build durations of a given project is relatively small (e.g, a few minutes more or less), developers may disregard such variation, since it could simply be caused by the load on CI servers. Therefore, we select projects that have a variation of build durations above 10 minutes [18, 48]. We use the Median Absolute Deviation (MAD) [51] to measure the absolute deviation from the median of a given distribution of build durations. The higher the MAD, the greater the variation of the build durations in each of the subject projects. We obtain the start and finish timestamps for each build of these projects from Travis CI in order to compute the perceived build duration. We compute the

MADs of the perceived build durations for each project. We filter out the projects for which the build duration MADs are less than 10 minutes. 67 projects survive this criterion.

Table 3.1 provides an overview of the 67 projects that satisfy the selection criterion. The studied projects form a variety of domains, including, but not limited to, programming languages, tools, applications, and services. In addition, these projects are of different sizes in terms of lines of code and development teams. The number of builds of the subject projects is $104,442$. We clone the Git repository of each studied project to compute CI build metrics. For example, we compute the experience of the developers who triggered the builds in terms of (a) the number of commits and (b) the number of days of development. We also analyze the Travis CI configuration file (i.e., *.travis.yml*) of each build to compute metrics related to the build configuration that we use in the models.

3.2.2 Data Processing

In this section, we explain how we process the data of the selected 67 projects. First, we show how we create the dependent variable based on build durations of the subject projects. Next, we discuss the selected independent variables.

Classification of Build Durations. We aim to study the factors that are associated with *long* build durations. Therefore, we classify build durations into *short* and *long*. To do so, we analyze the quantiles of build durations. Build durations that are above the third quantile (i.e., the upper 25% of the build durations) are classified as *long* and build durations that are below the first quantile (i.e., the lower 25% of the build durations) are classified as *short*. The resulting data column is our *dependent variable*. In summary, the dependent variable we use in the models is a binary factor

Table 3.1: An overview of the studied projects

Project name	Lang.	Domain	Lifetime	SLOC	Team size	# of builds	MAD (mins)
activerecord-jdbc-adapter	Ruby	Database drivers	2011-2016	8,995	11	604	28.52
ark	Ruby	Software archiving	2013-2015	596	5	88	14.76
balanced-ruby	Ruby	Online payment systems	2012-2015	1,707	10	383	13.17
blacklight	Ruby	Search Engines	2012-2016	6,233	13	1,998	13.22
blueflood	Java	Database systems	2013-2016	16,567	26	1,361	12.50
buck	Java	Build tools	2013-2016	192,058	47	1,568	14.68
cancancan	Ruby	Authorization services	2014-2016	1049	6	258	13.32
canvas-lms	Ruby	Learning management systems	2014-2014	141,681	41	311	11.44
cape	Ruby	Task automation	2011-2015	382	2	129	21.94
capybara	Ruby	Web applications	2011-2016	6,660	12	1,015	19.69
capybara-webkit	Ruby	Web development	2012-2016	935	10	269	10.60
ccw	Java	Programming languages	2013-2016	10,555	4	405	12.92
celluloid	Ruby	Build tools	2012-2016	2,909	13	1,411	10.55
celluloid-io	Ruby	Build tools	2012-2016	821	10	355	14.23
chef	Ruby	Configuration management	2013-2015	48,494	33	2,175	12.38
closure_tree	Ruby	Hierarchical data modeling	2012-2016	662	4	576	10.76
dcell	Ruby	Build tools	2014-2016	1,387	4	31	15.49
devise_cas_authenticatable	Ruby	Authentication services	2011-2016	516	4	167	11.66
diaspora	Ruby	Social networks	2011-2016	17,626	36	4,607	17.64
druid	Java	Distributed data storage	2016-2016	136,735	23	377	15.86
factory_girl_rails	Ruby	Build tools	2012-2016	213	7	88	18.41
flink	Java	Streaming	2016-2016	189,567	39	79	27.72
fluentd	Ruby	Logging systems	2013-2016	9,740	13	1,151	10.23
geoserver	Java	Data management	2013-2016	306,250	52	2,994	13.59
hapi-fhir	Java	Data management	2015-2016	533,980	5	683	13.59
jackrabbit-oak	Java	Content repository	2012-2016	109,827	8	8,183	16.36
java-design-patterns	Java	Development paradigms	2014-2016	8,657	9	1,049	11.66
Javaee7-samples	Java	Programming languages	2014-2016	19,162	3	94	28.28
jobsworth	Ruby	Project management	2011-2016	15,153	5	861	11.32
jruby	Ruby	Programming languages	2012-2016	155,721	39	12,056	19.15
js-routes	Ruby	Data exchange	2011-2016	185	5	243	10.40
kaminari	Ruby	Web development	2011-2016	873	10	435	18.06
killbill	Java	Business applications	2012-2016	61,986	4	2,756	22.98
librarian-puppet	Ruby	Repository management	2013-2016	1,268	8	369	11.66
LicenseFinder	Ruby	Search engines	2012-2016	4,334	14	708	16.09
lograge	Ruby	Logging systems	2012-2016	352	6	259	10.60
moped	Ruby	Database drivers	2012-2015	2,494	10	918	13.52
open-build-service	Ruby	Build tools	2012-2016	29,072	20	4,642	11.13
openproject	Ruby	Project management	2013-2015	53,422	47	7,088	12.59
opentsdb	Java	Database systems	2014-2016	29,085	4	440	11.45
oryx	Java	Business applications	2014-2016	10,209	4	910	17.35
paperclip	Ruby	File management	2011-2016	3,347	26	857	16.21
presto	Java	Query engines	2013-2015	149,663	20	2,153	14.78
promiscuous	Ruby	Database systems	2012-2016	3,054	2	453	12.91
rails	Ruby	Web applications	2011-2016	53,514	221	19,342	13.79
ransack	Ruby	Search engines	2011-2016	2,314	13	689	11.84
redmine_git_hosting	Ruby	Repository management	2014-2016	8,870	6	675	12.50
rollbar-gem	Ruby	Error management	2013-2016	1,939	10	1,204	10.42
rspec-rails	Ruby	Testing frameworks	2011-2016	1,899	19	1,308	22.10
ruboto	Ruby	Mobile App development	2013-2016	3,277	5	978	56.44
search_cop	Ruby	Search engines	2014-2016	739	1	126	16.51
shuttle	Ruby	Data exchange	2014-2015	10,493	6	183	18.78
Singularity	Java	Application containers	2016-2016	36,584	12	3,871	11.32
skylight-ruby	Ruby	App management	2013-2016	8,904	5	243	10.87
slim	Ruby	Syntax management	2013-2016	1,552	6	469	14.95
structr	Java	Web and mobile development	2012-2015	46,770	9	2,098	13.84
thinking-sphinx	Ruby	Search engines	2012-2016	4,026	5	425	17.99
titan	Java	Database systems	2012-2014	21,754	7	420	16.93
torquebox	Ruby	Programming languages	2014-2016	2,137	4	324	16.43
transpec	Ruby	Syntax management	2013-2016	4,115	2	603	13.32
twitter-cldr-rb	Ruby	Data exchange	2012-2016	7,421	6	810	12.00
uaa	Java	Authentication services	2013-2015	15,604	18	1,208	12.78
vanity	Ruby	Testing framework	2011-2016	2,369	6	380	12.33
wicked	Ruby	App management	2012-2016	267	2	159	13.12
xtreemfs	Java	Distributed data storage	2014-2016	157,655	13	831	10.80
yaks	Ruby	Data exchange	2013-2016	1,705	8	426	25.23
zanata-server	Java	Web applications	2013-2015	66,169	12	113	13.07

that consists of two values (i.e., *short* and *long*) to represent the build duration.

Metrics obtained from TravisTorrent: There are 61 data columns in TravisTorrent [15], in its 11.1.2017 release, that represent the characteristics of CI builds. In our analyses, we only consider the *build starting timestamp* and exclude the other timestamps (e.g., *the creation date of the first commit* and *the creation date of the PR*). In addition, we exclude the data columns in TravisTorrent that:

- do not represent factors, such as *build ids*, *pull request numbers*, *commit hashes*, and *test names.*
- contain values produced after the build is run, such as *the number of tests ran*, *build status*, and *setup time.* These metrics are not suitable for modeling *long* build durations, since we cannot obtain their values before starting the build [17].
- have high percentages of *zero* or *NA* values (e.g., *the number of comments on a commit*), since they can impact the results of the regression models.

After applying the exclusion criteria described above, only 20 metrics survived from our TravisTorrent data. The data of such metrics is collected from different sources, such as Git, GHTorrent, and Travis CI. After identifying the builds that belong to a specific project, the information of each build is collected from Travis CI. Then, all commits of the pushes that contain build-triggering commits are aggregated to obtain a precise representation of the changes that led to a specific build. Finally, Git and GitHub are used to collect commit information, such as developers, changed files, changed lines, and affected tests.

Metrics computed in this study: In addition to the 20 metrics obtained from TravisTorrent, we compute the following metrics:

- ***SLOC delta*:** We compute this factor by calculating the difference between the number of source lines of code of each build and its preceding build in the same branch.
- ***Day of week* & *Day or night:*** We compute these two metrics by deriving the day and the hour when the builds were started. Build starting timestamps are in the UTC time zone that is used by Travis CI.
- ***Lines of .travis.yml*:** We compute this factor by counting the number of instruction lines in the Travis CI build configuration file (i.e., *.travis.yml*) of each build. We exclude the blank and comment lines.
- ***Configuration files changed*:** We compute this factor by counting the number of project configuration files that have been changed by all the build commits. We consider a file as a project configuration file if it has one of the following extensions: `.yml`, `.xml`, `.conf`, `.rake`, `.rspec`, `.ruby-version`, `Gemfile`, `Gemfile.lock`, `Rakefile`, and `.sh`. To get the number of configuration files changed, we perform a `diff` at each build commit and count the number of unique files that end with the specified extensions.
- ***Configuration lines added/deleted*:** We compute these two metrics by counting the number of lines added or removed to/from all project configuration files. We perform a `diff` at each build commit on all the configuration files changed to get the total number of lines added/removed.

- ***Caching***: We compute this factor by analyzing the *.travis.yml* file to check whether it contains the `cache:` instruction.
- ***Fast finish***: We compute this factor by analyzing the *.travis.yml* file to check whether it contains the `fast_finish:` instruction.
- ***Travis wait***: We compute this factor by analyzing the *.travis.yml* file to check whether it implements `travis_wait:` in any of the build instructions.
- ***Retries for failed commands***: We compute this factor by analyzing the *.travis.yml* file to search for the maximum number of times to rerun failed commands using either a `--retry` command option or the `travis_retry` instruction.
- ***Author experience***: We compute two metrics as proxies for the experience of the developers who triggered the builds. We first get the name of the developer who authored the commit that triggered the build. Then, we search through all the commits that precede the build-triggering commit. We obtain the list of commits that were authored by the same developer who triggered the build. After that, we count the number of commits that each developer has and the number of days between the commit that triggered the build and the first commit of the developer in the repository. If a build was triggered by a commit that was authored by multiple developers, we obtain the experience metrics for the author with the largest number of commits and days.

In Table 3.2, we categorize the final set of metrics into five dimensions: CI metrics (8), code & density metrics (7), commit metrics (8), file metrics (7), and developer metrics (4). We use these metrics as independent variables in the models. We present a detailed description of each factor in the last column of Table 3.2.

Correlation and Redundancy Analysis. Regression models can adversely be affected by the existence of highly correlated and redundant independent variables [26]. Therefore, we perform correlation and redundancy analyses for the independent variables used in our models. We follow the guidelines that are provided by Harrell [47] to train regression models.

Correlation Analysis: Regression models can adversely be affected by the existence of highly correlated independent variables [26]. In this step, we perform a correlation analysis for the independent variables used in our models. We follow the guidelines that are provided by Harrell [47] to train regression models. To this end, we employ the Spearman rank ρ clustering analysis [88] to remove highly correlated variables in each of the subject projects. To this end, we use the `varclus` function from the `rms`[1] *R* package. For each pair of independent variables within all clusters that have a correlation of $|\rho| > 0.7$, we remove one variable and keep the other in the models. If two variables are highly correlated, we keep one variable in our models and remove the other variable. According to the principle of parsimony in regression modeling, simple explanatory variables should be preferred over complex variables [101]. Considering that our explanatory variables are equally simple (e.g., in terms of computation), we keep the variables that are more informative about the building process. For example, the *Lines of .travis.yml* is highly correlated with the *Number of jobs.* Therefore, we keep the *Lines of .travis.yml*, since it conveys more information about the build. Similarly, the *Test lines/KLOC* is highly correlated

[1]https://cran.r-project.org/web/packages/rms/rms.pdf

Table 3.2: Metrics used as independent variables in our mixed-effects logistic models

Dimension	Factor	Data type	Source	Description
CI metrics	Lines of *.travis.yml*	Numeric	Computed	Number of instruction (excluding blank and comment) lines in *.travis.yml*
	Number of jobs	Numeric	Computed	Number of jobs that are run in the current build
	Caching	Factor	Computed	Whether caching is enabled or not in *.travis.yml*
	Retries of failed commands	Numeric	Computed	Number of retries allowed for failed commands
	Fast Finish	Factor	Computed	Whether *fast_finish* is enabled in the build
	Travis wait	Numeric	Computed	The time used by the *travis_wait* feature for commands that take longer to run
	Day of week	Factor	Computed	The day of week in which the build is triggered (values $0-6$) with *Friday* as a reference level (alphabetically)
	Day or night	Factor	Computed	The time of the day in which the build is triggered: `day=0` & `night=1`, with *day* as a reference level (alphabetically)
Code & density metrics	Programming language	Factor	TravisTorrent	The dominant programming language of the build: `java=0` & `ruby=1`, with `java` as a reference level (alphabetically)
	Source Lines of Code (SLOC)	Numeric	TravisTorrent	Total number of executable production source lines of code of the project
	SLOC delta	Numeric	Computed	Difference between the SLOC of the current and the previous build
	Commits on touched files	Numeric	TravisTorrent	Number of unique commits on the files touched by the commits that triggered the current build
	Test lines/KLOC	Numeric	TravisTorrent	Number of lines in test cases per 1000 SLOC representing test density
	Test asserts/KLOC	Numeric	TravisTorrent	Number of assertions per $1,000$ SLOC representing asserts density
	Test cases/KLOC	Numeric	TravisTorrent	Number of test cases per $1,000$ SLOC representing test density
Commit metrics	Is pull request	Factor	TravisTorrent	Whether the current build was triggered by a commit of a pull request
	Commits in push	Numeric	TravisTorrent	Number of commits in the push that contains the build-triggering commit
	Source churn	Numeric	TravisTorrent	How much source lines of code changed by the commits in the current build
	Test churn	Numeric	TravisTorrent	Lines of test code changed by all the build commits
	Configuration lines added	Numeric	Computed	The number of added lines to configuration files
	Configuration lines deleted	Numeric	Computed	The number of deleted lines from configuration files
	Tests added	Numeric	TravisTorrent	The number of added tests
	Tests deleted	Numeric	TravisTorrent	The number of deleted tests
File metrics	Files added	Numeric	TravisTorrent	Number of files added by all the build commits
	Files deleted	Numeric	TravisTorrent	Number of files deleted by all the build commits
	Files changed	Numeric	TravisTorrent	Number of files changed by all the build commits
	Source files changed	Numeric	TravisTorrent	Number of source files changed by the commits in the current build
	Doc files changed	Numeric	TravisTorrent	Number of documentation files changed by all build commits
	Configuration files changed	Numeric	Computed	Number of configuration (e.g., `.xml`, `.yml`, and `.sh`) files modified by all the build commits
	Other files changed	Numeric	TravisTorrent	Number of other (not source, documentation, or configuration) files changed by all build commits
Developer metrics	Team size	Numeric	TravisTorrent	The number of developers in the team
	By core team member	Factor	TravisTorrent	Whether the build triggering commit was authored by a core member of the development team
	Author experience: # of days	Numeric	Computed	The experience of the developer who authored the build triggering commit in terms of number of days of experience
	Author experience: # of commits	Numeric	Computed	The experience of the developer who authored the build triggering commit in terms of number of commits

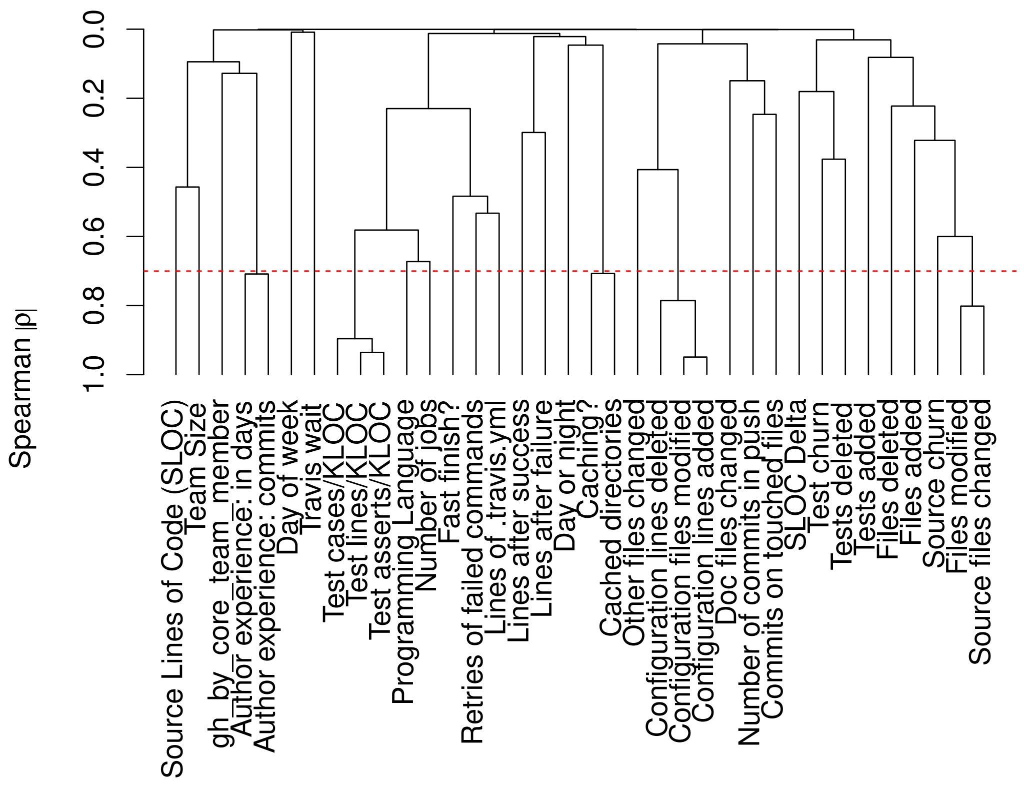


Figure 3.2: The hierarchical clustering of independent variables in the studied projects

with the *Test cases/KLOC*. Therefore, we keep the *Test cases/KLOC*, since it has more specific details than the *Test lines/KLOC*.

In Fig. 3.2, we show the dendrogram of the hierarchical clustering of independent variables for the subject projects. In this dendrogram, we observe five clusters of highly correlated variables ($|\rho| > 0.7$). In Table 3.3, we present the highly correlated variables and the selected variable within each cluster. We observe that there is an additional cluster (i.e., cluster 6) presented in Table 3.3. Such a cluster is formed after performing the variable selection in cluster 5.

Redundancy Analysis: Redundant variables can distort the relationship between

Table 3.3: Selected variables of the highly correlated variables in the projects

No.	Cluster of highly correlated variables	Selected variable
1	Author experience: # of days Author experience: # of commits	Author experience: # of days
2	Test cases/KLOC Test lines/KLOC Test asserts/KLOC	Test cases/KLOC
3	Lines of *.travis.yml* Number of jobs	Lines of *.travis.yml*
4	Configuration files changed Configuration lines added Configuration lines deleted	Configuration files changed
5	Files changed Source files changed	Source files changed
6	Source files changed Source churn	Source churn

the other independent variables and the dependent variable (i.e., *short* or *long* build duration) [47]. In this step, we perform a redundancy analysis on the remaining 26 independent variables (i.e., those that survive the correlation analysis step). To this end, we use the `redun` function from the `rms` *R* package, which models each independent variable using the remaining independent variables. If an independent variable can be estimated by other independent variables with an $R^2 \geq 0.9$, we discard such a variable. By performing the redundancy analysis, we observe that our dataset has no redundant variables.

3.3 Experimental Results

In this section, we discuss the motivation, the approach, and the findings of our research questions.

3.3.1 RQ$_{3.1}$: What is the frequency of long build durations?

Motivation. Studying the frequency and proportion of *long* build durations is important because it shows how critical is the situation. One could argue that the build duration is simply correlated with the lifetime of a project; i.e., *long* build durations are the most recent build duration of a project. Moreover, one could argue that *long* durations are associated with *passed* builds. To better understand *long* build durations, we study in this RQ the frequency and the characteristics of *long* build durations in CI. Studying whether the build duration increases over time is important because it helps to understand how the evolution of a project is associated with *long* build duration. It can also suggest that *long* build durations might be associated with other important factors than the evolution of a project. Studying the relationship between *long* build durations and the build status is important because it helps to understand whether *long* build durations are associated with only *passed* builds. It can also suggest that *long* build durations might be associated with other important factors even if the build is *errored* or *failed*.

Approach. In our analysis, the build duration represents the perceived build processing time of a build on Travis CI instead of the sum of the durations of build jobs provided by TravisTorrent. For example, if a build has 5 jobs and each of which takes 3 minutes to run, then the total duration of that build is 15 minutes. However, due to the fact that build jobs can run in parallel on Travis CI, this build may be generated in only 4 minutes. Therefore, we use the perceived build duration as it is more realistic. According to the subject projects, the perceived build duration is not correlated with the sum of the durations of build jobs (i.e., the Pearson's r value is 0.02).

To analyze the distributions of build durations in each of the subject projects, we perform the following:

- We use the Kruskal-Wallis test [59] to investigate whether a *long* duration is associated with the *passed*, *errored*, *failed*, or *canceled* build statuses. The Kruskal-Wallis test is the non-parametric equivalent of the ANOVA test [40] to check whether there are statistically significant differences between three or more distributions of build durations. Considering that the Kruskal-Wallis test does not indicate which build status has significantly different build durations with respect to others, we use the Dunn test [29] to perform individual comparisons. For example, the Dunn test indicates whether the build durations that belong to the *passed* builds are statistically different when compared to the build durations that belong to the *failed* builds. To counteract the problem of multiple comparisons [28], we use the Bonferroni-Holm correction [50] along with our Dunn tests to adjust our obtained *p-values*.

- We use Cliff's delta effect-size measures [20] to verify how significant is the difference in magnitude between the values of two distributions. The higher the value of the Cliff's delta, the greater the magnitude of the difference between distributions of build durations. For instance, a significant $p-value$ but a small Cliff's delta means that, although two distributions do not come from the same population, their difference is not significantly large. We use the thresholds provided by Romano et al. [87] to perform our comparisons: $delta < 0.147$ (*negligible*), $delta < 0.33$ (*small*), $delta < 0.474$ (*medium*), and $delta \geq 0.474$ (*large*).

- We use beanplots [56] to compare the distributions of build durations of the different projects. The vertical curves of beanplots summarize and compare the distributions of different datasets. The higher the frequency of data within a particular value, the thicker the bean is plotted at that particular value on the y axis.

Findings. ***We observe that over*** 40% ***of the builds in our dataset took over*** 30 ***minutes to run.*** We also observe that only 16% of the builds in our dataset have durations under 10 minutes. Fig. 3.3 summarizes the distributions of build durations of the 67 studied projects. In particular, Fig. 3.3 shows the distributions of the *minimum*, *lower quantile*, *median*, *upper quantile*, and *maximum* build durations of all the 67 projects. We observe that the median build duration varies in the studied projects, ranging from 8 minutes to 90 minutes (the overall median build duration is 20 minutes). We also observe that the 10-minute build duration is expressed by the median build duration of the *lower quantile* distribution. In addition, the distributions of the *max* and *min* build durations are highly right-skewed (skewness values of 6.3 and 4.7, respectively). The median build duration of the *max* duration distribution is 2.77 hours. The overlap between the different distributions of build durations suggests that we should consider (a) modeling the build duration differences between the studied projects and (b) varying the threshold of classifying build durations as *short* or *long*.

In Table 3.4, we show statistics about the build durations of the top four projects in terms of their number of builds. We observe in Table 3.4 that the median build durations of the projects range between 24 and 36 minutes. We also observe that the majority of build durations in `rails` (i.e., 90%) and `jruby` (i.e., 70%) are in the range of $1 - 10$ hours, with lower quantiles of around 2 and 0.7 hours and upper

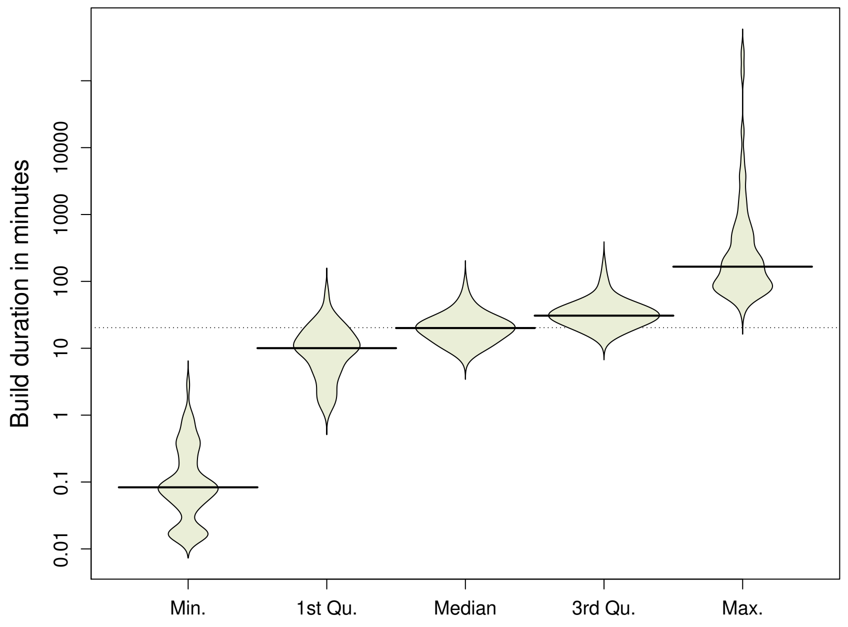


Figure 3.3: The distributions of build durations of the studied projects

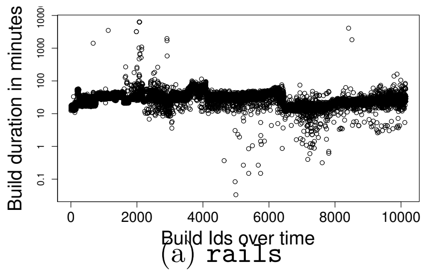
quantiles of 6 and 5 hours, respectively. Moreover, `jackrabbit-oak` has lower and upper quantiles of about 0.3 and 0.7 hours, respectively. Both `jackrabbit-oak` and `openproject` do not experience extremely `long` build durations as opposed to `rails` and `jruby`.

The build duration does not always increase over time. In Fig. 3.4, we show how build durations evolve over time for the top four projects. We show the durations of *passed* builds of such projects. We observe that build durations fluctuate as opposed to an increasing or decreasing trend of build durations over time. The fluctuation of build durations over time indicates that there are other possible factors that may have an association with the increase or decrease of build durations.

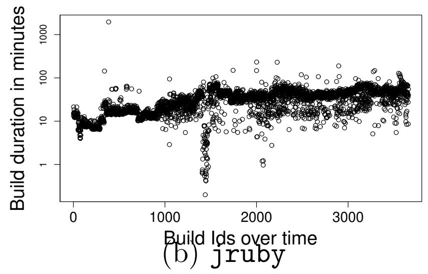
Durations of passed builds are not always longer than the durations of errored and failed builds. Table 3.5 shows the number and percentage of projects in which the build durations are significantly different between build statuses. By

Table 3.4: Statistics of build durations of the top four projects (statistics of build durations of the full set of projects is available in our online appendix [1])

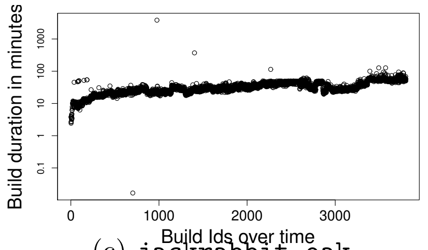
Project	**Build duration (*in minutes*)**		
	1^{st} Quantile	Median	3^{rd} Quantile
`rails`	18.18	27.18	36.75
`jruby`	13.17	24.09	40.05
`jackrabbit-oak`	20.32	29.45	42.48
`openproject`	28.80	36.50	46.28



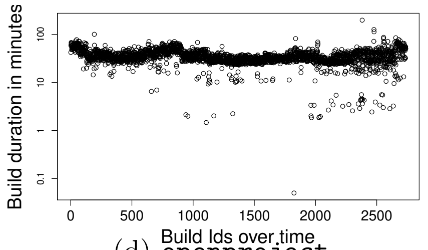
(a) `rails`



(b) `jruby`



(c) `jackrabbit-oak`



(d) `openproject`

Figure 3.4: Build durations over time for the top four projects (line plots for the full set of projects is available in an online appendix [1])

intuition, since *passed* builds complete all the build phases, they are expected to be longer than builds with any other statuses. Nevertheless, we observe in Table 3.5 that durations of *passed* builds are not always the longest durations amongst build statuses. Hilton et al. [49] performed a similar analysis on a different dataset of 34, 544 projects. They also found that *passed* builds run faster than *errored* and *failed* builds. Their speculation of the results suggests that many of the *passed* builds that run faster may not have generated meaningful results.

Table 3.5: The number and percentages of projects where build durations are significantly different from one build status to another

Status-pair build durations	**# of projects**	**% of projects**
Passed *longer than* Failed	29	43%
Passed *longer than* Errored	30	45%
Failed *longer than* Passed	9	13%
Failed *longer than* Errored	8	12%
Errored *longer than* Passed	13	19%
Errored *longer than* Failed	18	27%

Table 3.6: List of projects where *passed* run faster than *failed* and/or *errored* builds

Project name	**Passed *faster than* Failed**	**Passed *faster than* Errored**
`ark`	✓	✓
`canvas-lms`		✓
`capybara`		✓
`celluloid`	✓	✓
`celluloid-io`	✓	✓
`chef`	✓	
`diaspora`		✓
`druid`		✓
`jackrabbit-oak`		✓
`moped`	✓	✓
`openproject`	✓	
`promiscuous`	✓	✓
`ruboto`		✓
`skylight-ruby`	✓	
`twitter-cldr-rb`	✓	
`wicked`		✓
`xtreemfs`		✓

Our project-wise analysis of the results reveals that durations of *passed* builds are not significantly longer than durations of *failed* and *errored* builds in more than 50% of the projects. In addition, *passed* builds run faster than *failed* and/or *errored* builds in 17 (i.e., 25%) projects (shown in Table 3.6). In 5 projects, *passed* builds run faster than both *failed* and *errored* builds. In 4 projects, *passed* builds run faster than *failed* builds only, whereas *passed* builds run faster than *errored* builds only in 8 projects.

On the other hand, our results show that there is no significant difference between the build durations of *failed* and *errored* builds in 60% of the projects. Moreover, we observe that durations of *errored* builds are significantly longer than *failed* builds in 27% of the projects.

Therefore, we check if our results are sensitive to these types of noise in the build status data. Specifically, we consider that a build is (a) really *passed* if all jobs are *passed* and (b) really *broken* if all jobs are *broken*. Our cleaned dataset contains 64% of the total builds of our original data. We find that our observations hold for the majority of the studied projects. In particular, we observe that (1) the durations of *failed* builds are longer than the durations of *passed* in 6 projects, (2) the durations of *errored* builds are longer than the durations of *passed* in 8 projects, and (3) the durations of *errored* builds are longer than the durations of *failed* builds in 9 projects. Therefore, the noise in build statuses does not have a significant impact on our results. Moreover, the main focus of our study is to analyze *long* build durations.

We analyze a sample of the builds of the studied projects to investigate the reasons why the durations of *broken* builds could be longer than the durations of *passed* builds. In particular, we analyze builds of the `diaspora`[2] project. We find that the median duration of *passed* builds is 5.6 minutes, whereas the median durations of *failed* and *errored* builds are 12.6 and 13.2 minutes, respectively. We manually analyze the build logs of the broken build jobs of the `diaspora` project to investigate the reasons behind such results. We find that 43% of build failures in such jobs were due to commands that took longer than a certain limit of time to execute and were terminated by Travis CI. For example, running the '`./script/ci/build.sh`' script in build #`2671`[3]

[2]https://github.com/diaspora/diaspora
[3]https://travis-ci.org/diaspora/diaspora/builds/4033669

of `diaspora` took longer than 25 minutes and was terminated by Travis CI. As a result, job #3 *failed* and took 5-13 minutes longer than the other *passed* jobs of the build. In addition, connection and test timeouts were the reasons behind the failures of 12% and 2% of the build jobs, respectively. Moreover, we observe that, in 13% of the build job failures, Travis CI retried the failing commands two to three times with no success. For example, build *#4037*[4] of diaspora reran the command '`bundle install ...`' twice. However, job #4 *failed* while taking 8-24 minutes longer than the other *passed* jobs of the build.

Such observations indicate that certain build configurations (e.g., Travis waiting time and the number of times to rerun failing commands) may have an association with *long* build durations. Therefore, we use build configuration metrics, such as *Fast Finish*, *Travis wait*, and *Retries of failed commands*, as independent variables in our mixed-effects logistic models.

3.3.2 RQ$_{3.2}$: What are the factors associated with long build durations?

Motivation. RQ$_{3.1}$ shows that CI build durations may behave differently across projects. The fluctuating trend of build durations over time suggests that there are factors that may have an association with the increase or decrease of build durations. In this RQ, we aim to understand the different factors that may have an association with *long* build durations. We take into consideration both common wisdom factors (e.g., project size, build configuration size, team size, and test density) and other factors that may have an association with *long* build durations.

Approach. Our dataset contains builds from 67 studied projects. Such projects

[4]https://travis-ci.org/diaspora/diaspora/builds/10766342

are very different in terms of size and domain. As the results of $\text{RQ}_{3.1}$ suggest, *long* build durations are different from one project to another. Therefore, we use a mixed-effects regression to control the variation between projects in terms of build durations. Mixed-effects logistic models allow to assign (and estimate) a different intercept for each project [106]. Considering that we aim to study the relationships between *long* build durations and the metrics listed in Table 3.2, we particularly use the generalized mixed-effects models for logistic regression. Generalized mixed-effects models are statistical regression models that contain both fixed and random effects [36]. Fixed effects are variables with constant coefficients and intercepts for every individual observation. Random effects are variables that are used to control the variances between observations across different groups (i.e., projects). Our mixed-effects logistic models assume a different intercept for each project [66]. Traditional regression models, in contrast, use fixed effects only, which disregard the variances of build durations across projects [27].

Equation 3.1 shows the equation of the mixed-effects logistic model. In Eq. 3.1, Y_g denotes the binary build duration (i.e., *long* or *short*); β_0 demonstrates the constant intercept; X_i represents the independent variables; β_i represents the coefficients of each X_i; ϵ_g indicates the errors; and θ_g represents the intercepts that vary across each project. We use the `glmer` function in the `lme4` `R` package to use mixed-effects logistic models. We use the *binomial* distribution, *Laplace* approximation, and the *bobyqa* optimizer as parameters to the `glmer` function.

$$Y_g = \beta_0 + \theta_g + \sum_{i=1}^{n} \beta_i X_i + \epsilon_g \tag{3.1}$$

Significant independent variables are marked with asterisks in the output of the mixed-effects logistic models using the `ANOVA` test [79]. An independent variables is significant if it has $Pr(<|\chi^2|) < 0.05$. $Pr(<|\chi^2|)$ is the *p-value* that is associated with the χ^2-statistical test. The χ^2 (Chi-Squared) values show whether the model is statistically different from the same model in the absence of a given independent variable according to the degrees of freedom in the model. The higher the χ^2, the higher the explanatory power of an independent variable. We use *upward* ($\nearrow$) and *downward* ($\searrow$) arrows to indicate whether a variable has a direct or an inverse relationship, respectively, with the *long* build duration.

We compute the number of Events Per Variable (EPV) in the models. EPV shows the likelihood of a regression model to overfit [78]. EPV values represent the ratio of the number of builds with *long* durations to the number of independent variables. A dataset with an EPV above 10 is less risky to run into an overfitting problem [78].

We evaluate the performance of the models using the Area Under the Curve (AUC), the marginal R^2, and the conditional R^2. We describe each of our performance measures below:

- The Area Under the Curve (AUC) evaluates the diagnostic ability of the mixed-effects logistic models to discriminate *long* build durations [46]. AUC is the area below the curve created by plotting the true positive rate (TPR) against the false positive rate (FPR) using all the possible classification thresholds. The value of AUC ranges between 0 (worst) and 1 (best). An AUC value that is greater than 0.5 indicates that the explanatory model outperforms a random predictor.

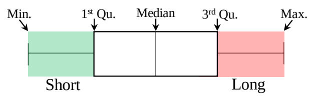
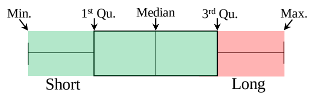
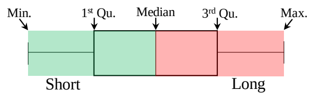
- The *marginal* R^2 is a measure of the goodness-of-fit of our mixed-effects models. It represents the proportion of the total variance explained by the fixed effects [75]. Higher values of the *marginal* R^2 indicate that fixed effects can well explain the dependent variable (in our case, *long* build durations).

- The *conditional* R^2 is a measure of the goodness-of-fit of the mixed-effects models. It represents the proportion of the variance explained by both fixed and random effects [75]. Higher values of the *conditional* R^2 indicate that the proportion of the variance that is explained by both fixed and random effects is higher than the proportion of the variance that is explained by fixed effects only. A high difference between the values of *conditional* and *marginal* R^2 suggests that the random effects significantly help to explain the dependent variable.

We conduct a sensitivity analysis using three scenarios of different classification thresholds for *long* build durations. We perform the sensitivity analysis to study how the model is sensitive to the classification threshold for *long* build durations. Table 3.7 presents the thresholds for classifying build durations into *short* and *long* using three classification scenarios. For each classification scenario, we present the obtained number of builds with *short* and *long* and the number of independent variables that survive the correlation and redundancy analyses. We explain the three classification scenarios below:

- **Scenario 1**: Build durations below the *lower* quantile are considered *short*, while the durations above the *upper* quantile are considered *long*. This scenario classifies 25% of the builds as *short* and 25% of the builds as *long*.
- **Scenario 2**: Build durations below the *upper* quantile are considered *short*, while the durations above the *upper* quantile are considered *long*. This scenario classifies 75% of the builds as *short* and 25% of the builds as *long*.

Table 3.7: Classification scenarios of build durations

Scenario #	**Classification threshold**	**Number of builds** *short*	*long*	**Number of variables**
Scenario 1	 Min. 1st Qu. Median 3rd Qu. Max. Short Long	26, 140	26, 113	28
Scenario 2	 Min. 1st Qu. Median 3rd Qu. Max. Short Long	78, 321	26, 113	30
Scenario 3	 Min. 1st Qu. Median 3rd Qu. Max. Short Long	52, 206	52, 208	30

- **Scenario 3**: Build durations below the median are considered *short*, while the durations above the median are considered *long*. This scenario classifies 50% of the builds as *short* and 50% of the builds as *long*.

We use the odds ratios [7] to measure the association of the dependent variable with the presence/absence of a binary independent variable (or the increase/decrease of a continuous independent variable) while holding the other variables at a fixed value. For example, odds ratios can explain how the *long* duration differs between builds that use and builds that do not use caching. We compute odds ratios by taking the exponentiation of the estimated coefficients obtained from the model for each independent variables. For the *Day of week* independent variable, the odds ratio for each day of week is computed over the reference day (i.e., *Friday*).

Findings. Table 3.8 shows the performance of the mixed-effects logistic models in terms of AUC, *marginal* R^2, and *conditional* R^2. Table 3.9 presents the variable importance results obtained from the model fit using the best performing scenario (i.e., *Scenario 1*). All the independent variables are sorted based on their χ^2 values in a descending order. For each independent variable, we show its estimated coefficient (estimated coefficients of the days of week are presented in Table 3.10), its χ^2 value, the *p-value* (represented by $Pr(<\chi^2)$), its significance to model *long* build durations, and whether each independent variable has a direct or an inverse association with *long* build durations (represented by the *upward* and *downward* arrows).

Our mixed-effects logistic models maintain a good discrimination performance when classifying long build durations using different duration thresholds. Our results indicate that the models maintain a high performance in all the classification scenarios for *long* build durations. We observe that the model using *Scenario 1* obtains a good AUC value of 0.87 for discriminating *long* build durations. In the second and third scenarios, the models obtain AUC values of 0.78 and 0.79, respectively. The *conditional* R^2 values of all the three models are higher than the values of the *marginal* R^2 values by 22%, 36%, and 19% respectively. The EPV values in the three scenarios are 932.61, 870.43, and $1,740.27$, respectively. Such high EPV values indicate that the models are less likely to be overfitting. In our subsequent analyses, we use the results obtained by the top performing modeling scenario (i.e., *Scenario 1*).

Build durations have a strong association with CI build metrics, such as (1) the build triggering time, (2) the number of times to rerun failing commands, (3) caching, and (4) finishing as soon as the required jobs finish. We observe that, as expected, the common wisdom factors (i.e., SLOC, lines of *.travis.yml*, team size, and test cases/KLOC) have significantly strong association with *long* build durations. However, we observe other less obvious factors to explain

Table 3.8: Performance of the mixed-effects logistic models

Classification scenario	**AUC**	***Marginal*** R^2	***Conditional*** R^2
Scenario 1	0.87	0.70	0.92
Scenario 2	0.78	0.45	0.81
Scenario 3	0.79	0.71	0.90

long build durations (e.g., caching, rerunning failing commands, time of triggering the build, and developer's experience). We also observe that commit-level metrics have a weak association with *long* build durations (e.g., tests added/deleted, files added/deleted, and the number of commits in a push). To better investigate the explanatory power of the less obvious factors to explain *long* build durations, we fit our top performing model without using the common wisdom factors (i.e., SLOC, lines of *.travis.yml*, team size, and test cases/KLOC). We observe that the model also maintains a good performance (i.e., the AUC values 85%). In addition, the model becomes more sensitive to project variances (i.e., the *conditional* R^2 is higher than the *marginal* R^2 by 0.46). Moreover, the less-obvious important factors preserve their explanatory power in both models (i.e., with and without the common wisdom factors).

Configuring CI builds to finish as soon as the required jobs finish is most likely to be associated with short build durations. The `fast_finish` setting in Travis CI allows builds to finish as soon as the status of the required build jobs is determined. In other words, the build is finished and its status is determined without the need to wait for jobs that are marked as `allow_failures`. Such a feature shows a high importance in producing short build durations. Our results reveal a

Table 3.9: Results of our mixed-effects logistic model – sorted by χ^2 descendingly

Factor	**Coef.**	χ^2	$Pr(<\chi^2)$	**Sign.**$^+$	**Relationship**
(Intercept)	2.59	12.0800	$5.1e^{-04}$	***	–
Fast finish	−0.72	1343.62	$< 2.2e^{-16}$	***	↘
Team size	4.53	843.73	$< 2.2e^{-16}$	***	↗
Retries of failed commands	0.61	667.21	$< 2.2e^{-16}$	***	↗
Test cases/KLOC	1.32	431.64	$< 2.2e^{-16}$	***	↗
SLOC	1.11	402.63	$< 2.2e^{-16}$	***	↗
Lines of *.travis.yml*	0.67	308.88	$< 2.2e^{-16}$	***	↗
Day of week	–*	214.59	$< 2.2e^{-16}$	***	–*
Is pull request	−0.19	163.78	$< 2.2e^{-16}$	***	↘
Day or night (night)	−0.14	25.69	$4.01e^{-07}$	***	↘
Caching	−0.08	13.82	$2.01e^{-04}$	***	↘
Source churn	−0.19	6.51	0.011	*	↘
Configuration files changed	0.04	5.32	0.021	*	↗
Test churn	−0.11	3.10	0.078	.	↘
Files deleted	0.02	1.93	0.165		↗
Files added	0.01	1.25	0.263		↗
Tests added	−0.01	1.22	0.270		↘
Tests deleted	−0.04	1.16	0.282		↘
Language (ruby)	−0.87	0.99	0.319		↘
Commits on touched files	−0.02	0.93	0.336		↘
Author experience: # of days	0.01	0.58	0.445		↗
Other files changed	−0.02	0.35	0.555		↘
Travis wait	0.01	0.18	0.672		↗
By core team member	0.00	0.06	0.809		↘
SLOC delta	0.00	0.02	0.882		↘
Number of commits in push	0.00	0.00	0.952		↗
Doc files changed	0.00	0.00	0.997		↘

$^+$Significance codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

*Individual coefficients for each day of the week are presented in Table 3.10

strong inverse association between fast build finishing and *long* build durations (χ^2 of 1343.62). Looking at the negative estimated coefficient (i.e., −0.72) of the *Fast Finish* independent variable, we observe that builds that are configured to finish as soon as the required jobs finish have significantly shorter durations than builds that are not configured with the `fast_finish` setting (*p-value* $= 2.2e^{-16}$). The odds of having a *long* build duration for builds with the `fast_finish` setting is 51% lower

Table 3.10: Estimated coefficients obtained from the mixed-effects logistic model for each *day of week*

Factor	**Coef.**	$Pr(<\|z\|)$	**Sign.**$^+$	**Relationship**
Saturday	-0.45	$4.46e^{-14}$	***	↘
Sunday	-0.51	$1.23e^{-15}$	***	↘
Monday	0.12	0.011	*	↗
Tuesday	0.09	0.037	*	↗
Wednesday	0.12	0.007	**	↗
Thursday	0.13	0.005	**	↗

$^+$Significance codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

than the odds for builds without the `fast_finish` setting.

To gain more insights about builds that are configured to perform fasting finishing, we analyze the *.travis.yml* file of the studied projects. We consider (a) builds that were triggered after supporting fast-finishing on Travis CI[5] (i.e., November 27th, 2013); and (b) builds that have `allow_failures` jobs. We find that 37 of the studied projects have builds that contain `allow_failures` jobs. However, we observe that only 16 (i.e., less than half) of these 37 projects have their builds configured with the `fast_finish` setting. In 10 projects, builds with the `fast_finish` setting run faster than builds without that setting (a median difference of 9 minutes). We investigate the projects that experienced no major reductions of build durations after enabling the `fast_finish` setting. We observe that the more `allow_failures` jobs a build have, the more likely for the `fast_finish` setting to speed up the build generation. For example, the `killbill`[6] project experiences the least benefit of the `fast_finish` setting among all the other projects. In `killbill`, the median percentage of the `allow_failures` jobs to the total number of jobs is 20%. On the other hand, the `ruboto`[7] project had the maximum reduction of build durations after enabling the

[5]https://blog.travis-ci.com/2013-11-27-fast-finishing-builds
[6]https://github.com/killbill/killbill
[7]https://github.com/ruboto/ruboto

`fast_finish` setting. In `ruboto`, the median percentage of the `allow_failures` jobs to the total number of jobs is 53%. Therefore, development teams should consider enabling the `fast_finish` setting to receive feedback about their builds as soon as the required jobs finish.

Caching content that does not change often has a strong inverse association with long build durations. Travis CI allows developers to cache contents (e.g., directories and dependencies) in their repositories onto the CI backend server. Caching enables Travis CI to upload the cache content only once and then use it while running all upcoming builds. Our results reveal a strong inverse association between caching and *long* build durations (χ^2 of 13.82). Looking at the negative estimated coefficient (i.e., -0.08) of the *Caching* independent variable, we observe that builds that use caching are significantly shorter than builds that do not use caching (*p-value* $= 2.01e^{-04}$). The odds of having a *long* build duration for builds that use caching is 8% lower than the odds for builds that do not use caching.

To gain more insights about why builds might experience *long* build durations even with the use of caching, we analyze the *.travis.yml* file of the studied projects. We consider builds that were triggered after December 17th, 2014 (i.e., after Travis CI introduced the caching feature for open source projects[8]). We find that 42 (i.e., 63%) of the projects have the caching feature enabled in their builds. We observe that caching was actively used in 30 out of the 42 projects (i.e., caching was enabled in more than 80% of the builds). We observe that caching reduced the build duration by a median of 11 minutes for only 13 of these projects. Moreover, we investigate the projects that have no notable reduction in the durations of builds that perform

[8]https://blog.travis-ci.com/2014-12-17-faster-builds-with-container-based-infrastructure/

caching. We observe that, in some projects (e.g., `killbill`[9] and `flink`[10]) caching was enabled in the build configuration without specifying the content to cache. In other projects (e.g., `vanity`[11] and `openproject`[12]), we observe that caching was applied mostly to `bundler`,[13] a `gem` for dependency management, rather than specific directories. Since `bundler` maintains frequent updates, caching it is less likely to have a significant reduction to build durations. As a consequence, caching content that changes more often can introduce overhead to the build generation process, since Travis CI may need to upload the cache frequently.

Maintaining a stable build status has a strong association with long build durations but with a negligible reduction in the build failure ratio. Our results reveals that there is a trade-off between *long* build durations and the attempts of developers to maintain *passing* builds. In particular, developers may configure their builds to rerun failing commands multiple times to avoid having many build failures. However, we observe that such a configuration has a strong association with *long* build durations. It is true that allowing builds to rerun a failing command several times may help to reduce the ratio of build failures. However, developers should take into consideration that the more times a command fails, the more duration the build would take. Most of Travis CI internal build commands can be wrapped with `travis_retry` to reduce the impact of network timeouts.[14] Looking at the positive estimated coefficient (i.e., 0.61) of the *Retries of failed commands* independent variable, we observe that the more reruns of a failed command in a build the

[9]https://github.com/killbill/killbill
[10]https://github.com/apache/flink
[11]https://github.com/assaf/vanity
[12]https://github.com/opf/openproject
[13]https://bundler.io
[14]https://docs.travis-ci.com/user/common-build-problems/#travis_retry

longer the duration of that build. Such an association between the number of retries of failed commands and *long* build durations is significant ($\chi^2 = 667.21$ and *p-value* $< 2.2e^{-16}$). In addition, a one-unit increase in the number of times of rerunning failed commands increases the odds of having a *long* build duration by 84%.

To gain more insights about builds that rerun failing commands, we analyze the projects that have explicit configuration instructions for specifying the number of times to rerun failing commands. We find 13 of the studied projects with such a configuration in their builds. We analyze projects that have at least 10% of their builds configured to retry failing commands multiple times. We observe that the median duration of builds that are configured to rerun failing commands is 13 minutes more than the builds without such a configuration. Although we observe that rerunning failing commands several times reduced the ratio of build failures in 60% of the projects, we find that the median reduction is only 3%. We manually investigate a sample of builds of the `jruby`[15] project. Such builds were configured to retry failing commands for 3 times. We find that the duration of build # `11843`[16] of `jruby` is more than the duration of its preceding[17] and succeeding[18] builds by 27 and 49 minutes, respectively. Although the commit[19] that triggered build # `11843` only updated the copyright year, the build was *errored*. The majority of the jobs of build # `11843` reran failing commands for $2 - 3$ times. Therefore, developers should carefully study the number of times to rerun failing commands, since it can generate unnecessary waiting durations in broken builds.

[15]https://github.com/jruby/jruby
[16]https://travis-ci.org/jruby/jruby/builds/108164066
[17]https://travis-ci.org/jruby/jruby/builds/108161963
[18]https://travis-ci.org/jruby/jruby/builds/108165671
[19]https://github.com/jruby/jruby/commit/30d975e6abdb1bdab1b80b0bfbd83313f139f8a2

Builds are more likely to have longer durations if they are triggered on weekdays or at daytime. We observe that the day of week is one of the important factors to model *long* build durations (χ^2=214.59 with a *p-value* $< 2.2e^{-16}$). Table 3.10 shows the individual estimated coefficients obtained from the mixed-effects logistic model for each day of the week. It is clear from Table 3.10 that *Saturday* and *Sunday* have an inverse relationship with *long* build durations, whereas the other weekdays have a direct relationship with *long* build durations. The estimated coefficient results implies that the odds of having *long* build durations on *Saturday* and *Sunday* is 36% and 40% lowers than the odds for *Friday* (*p-values* of $4.46e^{-14}$ and $1.23e^{-15}$, respectively). However, the odds of having *long* build durations on the other weekdays are $9-14\%$ higher than the odds for *Friday*(*p-values* of 0.011-0.037). Such a finding suggests that the servers of Travis CI have a higher workload on weekdays. Existing research has also found an association between the *Day of week* and buggy code changes [35, 92]. Furthermore, builds are more likely to have longer durations when they are triggered during the day. Looking at the estimated coefficient (i.e., -0.14) of the *Day or night* independent variable, we observe that builds triggered at night have a significant inverse relationship with *long* build durations (*p-value* = $4.01e^{-07}$). This suggests that the odds for having *long* durations of builds triggered at night is 13% lower than the odds for triggering builds during the day. Hence, builds are most likely to run faster because the servers of Travis CI have lower workloads at night.

3.4 Discussion

In this section, we discuss our findings about the important factors in terms of direct implications for developers, researchers, tool builders, and CI services.

3.4.1 Implications for developers

Developers should consider optimizing their core tests in addition to the removal of unnecessary tests. Developers acknowledge that tests are significantly associated with *long* build durations [48]. Much of developers' effort is usually invested to identify and remove unnecessary tests. However, developers should consider that the important software tests cannot be ignored. For important tests, the performance of test cases may be improved by reducing brittle assertions (i.e., assertions that depend on uncontrolled inputs) and unused inputs (i.e., inputs controlled by a test but not checked by an assertion) [52]. Tests can be optimized using a proper management of test dependencies [99]. Employing test case minimization techniques can improve the efficiency of software testing while maintaining an effective coverage [63, 64]. As a result, such techniques may help to reduce build durations. There exists a rule of thumb of restricting a test case to only one assertion [9]. Employing a single test assertion per test case improves fault localization [108] and test readability [71], but may have an association with *long* build durations. If the build duration is a very important factor to a development team, sacrificing the test readability factor might be a wiser choice. For example, developers may combine several test cases into a single comprehensive test case if such tests share similar characteristics. Developers may use proper explanatory messages for test assertions to distinguish the test assertions of each functionality in the case of test failures. Nevertheless, developers should take into consideration that removing a few tests in a commit (or a push of commits) will less likely have an association with the build duration. For example, the commit-level metrics, such as the *Test churn*, *Tests added*, and *Tests deleted*, have a weak association with *long* build durations. Therefore, development teams should consider performing test optimization whenever they produce a new release of the project.

To gain more insights about how the test density may be associated with the *long* build duration, we perform a manual analysis for sample test cases of the `structr`[20] project. We find five test cases (i.e., test methods) that contain $33 - 60$ test lines and $5 - 9$ test asserts. Although each of such test cases targets a certain system functionality, we observe duplicate code and asserts between them. For example, we find a test case for the functionality of moving a file to an arbitrary directory and another test case for moving a file to the root directory. Having a separate test case for each functionality of the system helps to locate test failures. However, if reducing the build duration is more important, refactoring the test code (e.g., by resource inlining or reducing the test data) [71, 99] would be a wiser option.

Not all build jobs are parallelized. Developers should realize that, for free subscriptions of Travis CI, there is a limit of 5 jobs to run simultaneously. If a build has more than five jobs, only five of them would be running in parallel. Once one of the jobs finishes, another job can start running along with the four running jobs, and so on. Development teams can maintain paid subscriptions depending on how many concurrent jobs are needed to run. On the other hand, build jobs may be configured to run in stages. A build stage may contain a set of jobs that can run in parallel.

[20]https://github.com/structr/structr

Jobs of a build stage do not run in parallel with jobs of other build stages. Instead, jobs of next build stages wait for the jobs of previous build stages to finish. Hence, maintaining build stages indicates that, even though parallelizing jobs significantly helps to reduce build duration, the number of jobs still matters when it comes to the build duration. Future research should study how open source projects (i.e., free CI subscribers) may become more costly than projects with a paid service based on the gain in terms of build durations.

3.4.2 Implications for researchers

Test optimization and prioritization may be useful to reduce the build duration. Researchers should explore ways to identify tests that may perform similarly (i.e., semantic test clones) in order to potentially reduce build durations. Developers add more tests whenever a new system functionality is introduced to the project. Hence, due to the frequent additions of tests, developers may neglect writing efficient test cases. In addition, due to parallel development activities, it could be hard for developers to identify whether a test is a duplicate of another existing test. Therefore, researchers may explore ways to prioritize software tests from a CI perspective [33, 67].

Longer build durations may indicate a potential low performance of the system at runtime. Researchers should investigate whether the build duration has a potential correlation with the performance of the system at runtime. If such a correlation exists, researchers may leverage existing performance optimization techniques to optimize existing software tests. Doing so may help to reduce the build duration.

3.4.3 Implications for tool builders

Tool for detecting cacheable spots of the project. Developers need tools to identify build configurations that may be associated with build durations. For example, it would be beneficial for developers to have a tool that detects parts of the project that do not change often. Developers can cache such parts to speed up running the builds.

Tool for detecting commands that often pass after multiple reruns. Developers may wish to know information about the commands that require multiple runs to pass. Developers can leverage such information to identify the cause of the frequent command failures and fix the issue accordingly. For example, if installing a dependency frequently fails, it is better for developers to find alternative mirrors or versions of that dependency.

3.4.4 Implications for CI services

The workload of CI servers can indicate latency in build generation. CI services (e.g., Travis CI) should provide mechanisms for developers to receive instant updates about the workload of their servers. Information about the current workload of a CI server can help developers to expect any possible delays that might impact the perceived build duration.

CI services should utilize the current behavior of builds to suggest possible build (re)configurations. Misusing a CI configuration may unintentionally be associated with *long* build durations. For example, developers may configure builds to update dependencies in every run to avoid breaking the build in the case of unexpected dependency updates. Therefore, it is better for services CI to optionally perform such an update only if a dependency is recognized to be not up to update. It is also better to send feedback to development teams about the possible (re)configuration performed on their builds. The feedback may also incorporate information about the reasons why recently triggered builds have longer build durations than the previously triggered builds.

3.5 Threats to Validity

In this section, we discuss the threats to the validity of our study.

Construct validity. Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. In our study, we rely on the data collected mostly from TravisTorrent and from the Git repositories that we clone from GitHub. Mistakenly computed values can have an influence on our results. However, we carefully filter and test the data to reduce the possibility of wrong computations that may impact the analyses of this study. In addition, the build status data may contain noise that may impact our obtained results. For example, *passed* builds may contain *broken* jobs while *broken* builds may contain *passed* jobs. We filter such noises and perform a status-wise analysis of build durations using cleaned data. We observe that noises in build statuses do not significantly impact our overall observations.

Internal validity. Internal threats to validity are concerned with the ability to draw conclusions from the relation between the independent and dependent variables. We study the factors that are strongly associated with *long* build durations. To do so, we use mixed-effects logistic models and study the explanatory power of the independent variables to explain *long* build durations. We also perform a sensitivity analysis with the use of three classification scenarios using different statistical thresholds for build durations: the *median*, the *lower* quantile, and the *upper* quantile.

In the mixed-effects logistic models, we use 28 metrics as independent variables spanning five dimensions: *CI metrics*, *code metrics* & *density metrics*, *commit metrics*, *file metrics*, and *developer metrics*. However, we are aware that these metrics are not fully comprehensive and using other metrics may affect our results. In our correlation analysis, deciding which variables to keep in the mixed-effects logistic models may have an impact on the results of the models. To make our obtained results reproducible, we explicitly define our choices of variables for all the possible pairs of highly correlated variables.

External validity. External threats are concerned with our ability to generalize our results. Our study is based on builds that are collected from a set of 67 projects. Therefore, we cannot generalize our conclusions to other projects with different characteristics. Nevertheless, our study selects projects with high variations of build durations where the problem of *long* build durations may occur. To this end, we select the projects that have a build durations MAD above 10 minutes [18, 48]. However, despite the relatively small sample, our dataset contains well-known and previously studied projects (e.g., `rails`, `jruby`, and `openproject`) Projects with lower build durations MADs (i.e., less than 10 minutes) are less likely to suffer from *long* build durations. Still, future work should investigate whether lower MADs of build durations would produce different results as compared to our findings. Moreover, a replication of our work using projects written in other programming languages is required to reach more general conclusions.

3.6 Summary

In this chapter, we conduct an empirical study to investigate *long* build durations. We study the *long* duration of $104,442$ CI builds over 67 GitHub projects that are linked with Travis CI. We model *long* build durations using mixed-effects logistic models. We use mixed-effects logistic models to identify the most important factors to model *long* build durations. Finally, we gain more insights about the relationship between the most important factors and *long* build durations by performing manual analyses of the studied projects. We observe the following:

- About 40% of build durations take over 30 minutes to run.
- Build durations may increase or decrease over time, which indicates that there exist important factors that have a strong association with such a fluctuation.
- Durations of *passed* builds are not always longer than durations of *errored* or *failed* builds.
- Triggering CI builds during the day or on weekdays is most likely to be associated with *long* build durations.
- Short build durations are associated with builds that are configured (a) to cache content that does not change often or (b) to finish as soon as all the required jobs finish. However, misusing such configurations may not help to reduce the build duration.

- There is a tradeoff between maintaining stable build statuses and *long* build durations. In particular, configuring builds to rerun failing commands multiple times has a strong association with *long* build durations with a negligible reduction in the build failure ratio.

In the future, we plan to perform a qualitative study to investigate how developers deal with *long* build durations. We also aim to extend our experimental study to include an industrial setting.

Chapter 4

A Study of the Impact of Noisy Build Breakages in CI

This chapter describes our study of noisy build breakages. Section 4.1 presents the research problem and motivation of our study. Section 4.2 discusses build breakage examples that motivate our work. Section 4.3 presents the study design of our empirical study. Section 4.4 describes the criteria to identify build breakages that can potentially introduce noises to build breakage data. Section 4.5 presents the results and findings of our study. Section 4.6 discusses the implications of our findings for researchers and tool builders. Section 4.7 describes the threats to the validity of our results. Finally, Section 4.8 summarizes the chapter and suggests future work.

4.1 Problem and motivation

CI builds consist of multiple jobs, each of which runs on a different runtime environment. A CI build can break if any of its jobs breaks. A build breakage may occur due to several reasons, such as configuration errors, installation errors, compilation errors, or test failures [85, 117]. Much research has been devoted to studying breakages of CI builds. In addition, researchers studied the possibility of predicting build breakages [61, 76, 85, 107]. Other studies investigated the factors that share a strong association with CI build breakages [61, 68, 76, 85]. These studies found that process metrics (e.g., commit complexity and file types), developer roles (i.e., core or peripheral), and build breakage history are among the factors with the strongest association with the likelihood of a build breakage.

However, prior research has paid little attention to builds that may break due to reasons that are unlikely to be related to the activities of developers. For example, a build may break due to environmental factors, such as timeouts, connection resets, or memory allocation errors. Zhao et al. [115] studied the types of build breakages in 250 open source projects spanning a 10-month period. They observed a decreasing trend over time in the number of build breakages that are caused due to timeouts or missing dependencies. Rausch et al. [85] studied build breakages in 14 open source Java projects and observed a non-negligible amount of noise in build breakage data.

Despite the valuable insights reported by existing studies, there was no discussion as to whether build breakages are really caused by the developers who triggered the builds. In particular, build breakages that are caused due to environmental errors should not be used to study the association of build breakages with development activities. Additionally, prior studies did not consider that builds can break due to (unfixed) errors introduced in previous builds or due to noisy build jobs that were excluded from the build later. Although prior studies discussed noises that may exist in build breakage data, the impact of modeling build breakages using clean data (i.e., after removing the noises) has not been investigated. Studying build breakages without considering the aforementioned factors may lead to misleading observations. For example, development activities (e.g., being a core developer) may be deemed as the culprits for breaking CI builds when it is not the case.

The goal of this study is to (a) identify breakages that can potentially introduce noises in build breakage data and (b) measure the impact of such noises on the observations reported in the literature. Noisy build breakages may occur due to environmental factors (e.g., errors in CI servers), previous (unfixed) build errors, or build jobs that are later deemed by developers as noisy. We conduct an empirical study on data collected from TravisTorrent [15], a commonly used dataset to study CI build breakages [14, 68, 76]. Our dataset contains 350, 246 CI builds from 153 GitHub projects linked to Travis CI. Based on the above goals, we address the following RQs:

$RQ_{4.1}$: What is the proportion of noisy build breakages?

> We define three criteria for identifying CI build breakages that can taint the results of the prior research. Our results show that 33% of broken builds are impacted by environmental factors that are unlikely to be related to development activities. We identify 60 (sub)categories of environmental build breakages. We also observe that 9% of broken builds are primarily broken by jobs that were later deemed by developers as noisy.

$RQ_{4.2}$: How do noisy breakages impact build breakage models?

> We measure the impact of using uncleaned build breakage data (i.e., containing noisy build breakages) on modeling CI build breakages and the subsequent analyses. We observe that using uncleaned build data to model CI build breakages has a considerable impact on the performance of models (i.e., an AUC

reduction of 6%). In addition, we observe that models that use uncleaned build breakage data can lead to misleading conclusions. For example, in the model that uses uncleaned data, build breakages are significantly associated with core developers, whereas the model fitted using the cleaned data shows no evidence for such an association.

$RQ_{4.3}$: How do noisy breakages impact prior findings?

We compare our findings with the findings reported by Rausch et al. [85]. We observe that observations reported by prior research (e.g., pull requests cause more breakages or less frequently committers cause fewer build breakages) do not hold if noisy build breakages are filtered out from build breakage data.

4.2 Motivating Examples

Prior studies have relied heavily on the build statuses generated by Travis CI to perform the analyses [14, 76, 85, 107, 116]. For example, researchers have proposed models to study the relationship between the build status (i.e., *passed* or *failed*) and several metrics that are related to the development process (e.g., committing code changes by core developers or at night) [85]. However, in this section, we illustrate three types of build breakages that could potentially impact the results reported by these models.

Environmental breakages. *Environmental* breakages are caused by the environment that generates the builds (e.g., the Travis CI servers). Examples of *Environmental* breakages are connection timeouts or exceeded time limits when running commands or tests. Such breakages are unlikely to be caused by developers. A real

Table 4.1: An excerpt of CI builds in the `Diaspora` project

Build No.	Developer*	Status	Broken jobs	Detailed status	Breakage reason
191	C	passed	0	Successful	
192	A	errored	1	Timeout	No actual breakage
193	A	errored	1	Timeout	No actual breakage
194	B	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	First occurrence of the breakage
195	A	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Unrelated to the breakage
196	C	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Unrelated to the breakage
198	B	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Failed Fix attempt
199	D	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Unrelated to the breakage
200	B	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Failed Fix attempt
201	B	errored	6	***Failure/Error: Resque.should_receive(:enqueue)***	Failed Fix attempt
202	A	errored	1	Timeout	Successful Fix attempt
203	B	errored	0	Errored, but all jobs have passed	Unrelated to the breakage
204	E	errored	0	Errored, but all jobs have passed	Unrelated to the breakage
208	E	passed	0	Successful	

* Developer names are encoded

example from the `Diaspora` project[1] is illustrated in Table 4.1. As shown in the table, a sequence of builds were generated from *September 7th, 2011* to *September 8th, 2011*. At a first glance, one might assume that developer A broke the build (i.e., build `192`) and, after several failed attempts to fix the breakage (i.e., at builds `193-204`), developer E finally fixed the breakage (i.e., at build `208`). Nevertheless, by taking a deeper look at the logs of the builds listed in Table 4.1, we uncover a different story. In reality, developer A did not break the build, since the real reason for the breakage of build `192` was a server connection timeout (see the "Detailed Status" column of Table 4.1). In fact, developer B broke the build when generating build `194`. Moreover, developer E did not fix the breakage. In fact, developer A fixed the breakage when generating build `202`. However, build `202` was broken due to another connection timeout. Interestingly, the two consecutive builds (builds `203` and `204`) were not really broken. Both builds received a broken status due to a bug on the Travis CI service. As a result, all the jobs of builds `203` and `204` have passed, however, Travis CI wrongly generated broken statuses.

Cascading breakages. *Cascading* breakages occur because of inherited mistakes

[1] https://github.com/diaspora/diaspora

from previous builds. For example, considering builds ranging from `194` to `201`, one might assume that they all represent different breakages. However, by analyzing their logs, we observe that only build `194` has a new error when compared to its predecessor (see "Detailed status", in Table 4.1). The remaining builds (i.e., `195-204`) share the same error as build `194`, meaning that these builds are unlikely to represent new breakages. In other words, the newly pushed commits are unlikely to be the cause of the breakage. Therefore, researchers should be careful when including *Cascading* breakages in their models if the main goal is to predict new build breakages.

Allowed breakages. *Allowed* breakages occur when builds are broken by jobs that are disregarded by developers and later deemed as noisy. Builds can have jobs with integration environments that are under experimentation. If these jobs recurrently break the builds, developers may decide later to exclude such jobs in one of the three forms: (a) marking the job as *allow_failures*, (b) completely removing the job from the build, or (c) changing the job configuration to a different environment. Therefore, builds that are broken only because of such jobs could have passed if such jobs were excluded earlier by developers. To illustrate this type of breakage, we show in Table 4.2 an example of the `Puma`[2] project. Table 4.2 shows a sequence of 17 builds that were generated from *December 3rd, 2013* to *January 25th, 2014*. As shown in Table 4.2, there are 14 builds that are broken due to different job breakages. We observe that job `C` and job `D` are recurrently broken and, as a result, 14 builds are broken. However, in build `473`, both jobs are marked as *allow_failures*, which allowed build `473` to pass. As a consequence, build `456` and builds `458-472` could also pass if either jobs or both of them were excluded earlier. Build `471` could also pass, since

[2]https://github.com/puma/puma

Table 4.2: An excerpt of CI builds in the `Puma` project

Build No.	Status	Broken jobs*	Build configuration action
450	passed		
453	failed	A & B & C & D	
454	failed	A & B & C & D	
456	failed	C & D	
457	passed		
458	errored	D	
459	failed	D	
461	failed	C & D	
462	failed	C & D	
466	errored	C & D	
467	failed	C & D	
468	failed	C & D	
469	failed	C & D	
470	failed	D	
471	errored	C & D & E & F	Jobs added: E & F & G
472	errored	C & D	Jobs removed: E & F
473	passed		Jobs marked as allow_failures: C & D

* A: Ruby: `1.9.3`, B: Ruby: `2.0.0`, C: Ruby: `jruby-19mode`, D: Ruby: `rbx`, E: Ruby: `jruby-18mode`, F: Ruby: `1.8.7`, G: Ruby: `2.1.0`

its status was partially impacted by both jobs and partially by jobs `E` and `F`, which were removed in build *472*. However, builds `453` and `454` would not passed, since they were broken due to two other broken jobs (i.e., `A` and `B`).

4.3 Study Design

This section presents the experimental setup and the steps of collecting and processing the data for our studied RQs.

4.3.1 Data collection

Fig. 4.1 shows an overview of our empirical study. We collected data from TravisTorrent [15]. TravisTorrent, in its 11.1.2017 release, stores CI build breakage data of 1,283 projects. Prior research regarding CI builds relies heavily on the TravisTorrent data when conducting empirical studies on CI [14, 76, 85, 107]. TravisTorrent

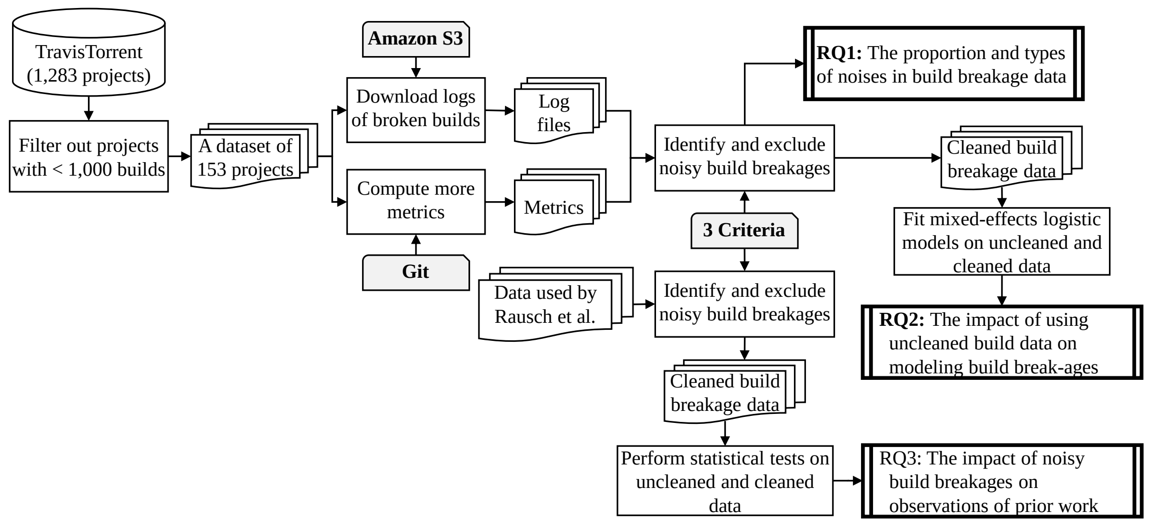

TravisTorrent (1,283 projects)
Filter out projects with < 1,000 builds
A dataset of 153 projects
Amazon S3
Download logs of broken builds
Log files
Compute more metrics
Metrics
Git
Identify and exclude noisy build breakages
RQ1: The proportion and types of noises in build breakage data
Cleaned build breakage data
Fit mixed-effects logistic models on uncleaned and cleaned data
RQ2: The impact of using uncleaned build data on modeling build break-ages
3 Criteria
Data used by Rausch et al.
Identify and exclude noisy build breakages
Cleaned build breakage data
Perform statistical tests on uncleaned and cleaned data
RQ3: The impact of noisy build breakages on observations of prior work

Figure 4.1: Overview of our empirical study of the noises in build breakage data

contains projects that are written in three programming languages: Ruby, Java, and JavaScript. We filter the studied projects using two criteria to ensure that we have sufficient data for our analyses. Some projects in TravisTorrent do not actively generate CI builds (e.g., toy projects), which leads to a small number of builds for such projects. Therefore, we select projects with at least $1,000$ unique builds in TravisTorrent. Using this criterion, we obtain 154 projects (66 Java, 87 Ruby, and one Javascript). We exclude the Javascript project, since it cannot be used as a representative of the Javascript language. Our dataset contains builds that are triggered by different events: git pushes (83.69%), pull requests (15.98%), API requests (0.32%) and scheduled Cron jobs (0.01%).

We show a complete overview of the 153 studied projects in our online appendix [2]. These projects are of a range of domains, including, but not limited to, applications, programming languages, and tools. The number of CI builds of our selected projects is $350,246$, which represents about 52% of the total builds in TravisTorrent. The total number of build jobs in our dataset is $1,927,239$ with an average of 5 jobs per build.

Each build job runs (and may pass or break) independently from each other. We use the `TravisPy`[3] API to collect more metrics about build jobs (e.g., the job status and configuration). We also use the `boto3`[4] API to download the plain-text build logs from Amazon S3, the storage backend of Travis CI. . Moreover, we compute additional metrics for builds (e.g., the number of configuration files that are modified) using the commits that trigger the builds in our dataset.

4.3.2 Data processing

We sort builds of our studied projects in a chronological order according to the triggering time of each build. We also exclude *Canceled* builds because they are incomplete and do not give a clear picture about whether they would pass or fail if they had not been interrupted. Considering that a build may contain multiple jobs, the breakage could be partial (i.e., some jobs are broken) or complete (i.e., all jobs are broken). In our analyses, we only consider the broken jobs of broken builds. We use the build logs for such jobs to investigate the categories of environmental build breakages. We include *allow_failures* jobs in our analyses at the job level. However, we disregard the *allow_failures* jobs when we label a breakage as *Allowed* breakage (i.e., at the build level).

4.4 Criteria to Identify Noisy Build Breakages

In this section, we explain how we apply three criteria to identify (1) *Environmental* breakages, (2) *Cascading* breakages, and (3) *Allowed* breakages in the studied

[3]https://pypi.python.org/pypi/TravisPy/0.1.1
[4]https://aws.amazon.com/sdk-for-python

projects. We consider these types of build breakages as '*noise*', since they are unlikely to be related to the commits that trigger CI builds. Keeping such breakages in build breakage prediction models can affect modelling performance and produce misleading results. Noisy build breakages can also be referred to as flaky breakages, in which a build might pass or fail even if the codebase does not change (i.e., a non-deterministic build status). Hence, after we apply the above three criteria, we clean our studied dataset by excluding builds that have any of the three types of breakages. A replication package of the heuristics used in this Section is available in our online appendix [2].

4.4.1 Identifying ***Environmental*** breakages

To identify *Environmental* build breakages, we analyze the raw build log files for all broken jobs that belong to broken builds in our dataset. The total number of broken jobs is $321,855$. We label build logs of broken jobs based on the error or failure messages found. Our log labeling process is semi-automated as it involves manual and automated analyses of build logs. In the manual analysis, we scan build logs to find any error messages that correspond to the build breakage. Travis CI may recover from errors but keeps a record of such errors in the build log. Therefore, we label build logs according to the last logged error message. For example, a build may experience a dependency installation error but, because failed commands may rerun several times, the command that installs the dependency may succeed later. After a successful dependency installation, however, the build may experience an environmental error (e.g., a timeout error). In such a case, we ignore the message of the recovered error and label the build log according to the last error message (i.e., the timeout error).

Build logs contain heterogeneous and inconsistent error messages. A certain build error may be reported using different forms of error messages. For example, "`The command xyz exited with 8`", "`ERROR 404: Not Found`", and "`Error: 404 Client Error`" are different forms of error messages to report a server connection error. Therefore, we employ a thematic analysis [39] to manually identify themes (i.e., categories) of *Environmental* build breakages. Two researchers perform the manual analysis using a statistically significant sample of 384 out of $321,855$ build logs (a confidence level of 95% and a confidence interval of $\pm5\%$). We use open coding [23] to produce an initial set of categories of *Environmental* breakages. In addition, we tag each build log with one of three labels: *Developer*, *Environmental*, and *Suspicious*. We assign the *Developer* label to the jobs in which the build breakage is most likely caused by errors introduced by developers. We assign the *Environmental* label to the jobs in which the build breakage is most likely caused by environmental factors. We assign the *Suspicious* label to the jobs in which we could not identify the underlying cause of breakage (e.g., empty logs, accidentally trimmed logs, or terminated logs with a successful exit code). Each build log is assigned a single label, which represents the label of the broken job. We conduct consensus meetings to resolve all labeling and categorization disagreements.

After the manual analysis, we use the identified labels and categories to generate heuristics (i.e., python scripts [2]) that automate the process of identifying error messages, categorizing, and labeling the build logs in our dataset. After the automated labeling, the two co-authors perform a manual analysis of another statistically significant sample (i.e., additional 384 build logs) to validate the labels and categories generated by the automated labeling. We use the Cohen's *kappa* inter-rater agreement statistic [21] to measure how reliable is the manual validation of the automatically generated labels. To compute the agreement level, we used two codes in which the raters indicate whether there is a *match* or *mismatch* between the automatically generated and manually assigned labels We discuss the cases in which there is (i) a disagreement between the raters or (ii) a mismatch between the manual and automated labeling. We resolve the disagreements using consensus meetings, refine our heuristics, and repeat the same process for the set of unlabeled build logs. We provide more details about the obtained level of agreement in Section 4.5$-RQ_{4.1}$: Findings.

We approach online resources to identify whether an error messages is related to *Environmental* issues. For example, we study issues related to these error messages on different issue reporting websites (e.g., Travis CI,[5] rvm,[6] and rubygems[7]). Some build breakages require a deeper investigation prior to classifying them as *Environmental*. For example, in the cases where errors are caused due to missing objects (e.g., files, configuration, or gems), we analyze the commits that trigger the broken builds to check whether such objects exist in the repository or not. To do this, we automatically perform a `git diff` command for the commits that triggered the build. Then, we check whether the reported objects are truly missing in that revision. If a missing object is found in the repository, we consider the build breakage as *Environmental*. Otherwise, we consider the build breakage as *Developer*.

Considering that a build may have more than one job, we label builds using the labels of their jobs. We exclude the jobs that are marked as *allow_failures* when we label builds, since such jobs do not break the build if they are broken. We assign the *Developer* label to builds that have at least one broken job with the *Developer* label.

[5]https://github.com/travis-ci/travis-ci/issues
[6]https://github.com/rvm/rvm/issues
[7]https://github.com/rubygems/rubygems/issues

We assign the *Environmental* label to builds that have all the broken jobs labeled as *Environmental* or *Suspicious*. Builds that have the *Developer* label are the only builds that we should associate with developers' activities. However, environmentally broken builds cannot be considered as *passed*, since *Developer* errors may potentially occur if such builds are restarted. Therefore, a clean build breakage dataset should contain only build breakages that have the *Developer* label.

4.4.2 Identifying *Cascading* breakages

To identify *Cascading* build breakages, we sort the builds of each branch of the studied projects in a chronological order based on the build triggering timestamp. Then, we analyze every pair of consecutively broken builds to identify which builds are simply broken because of a former (unfixed) breakage. First, we compare the number of broken jobs between each pair of consecutively broken builds. If both builds have a matching number of broken jobs, we compare the number of errors in all broken jobs. If both builds have a matching number of errors, we compare their error messages. If both builds have identical error messages, we consider that the current build breakage is broken due to existing (unfixed) errors introduced by the commits that triggered by the previously broken build. However, we cannot assume that the commits of the currently broken build are free of errors, since errors of former commits may hinder showing the errors that might be introduced by current commits.

4.4.3 Identifying *Allowed* breakages

To identify *Allowed* build breakages, we use the chronologically sorted builds to identify the broken jobs that are later removed from the builds or marked as *allow_failures*.

The excluded jobs may contain errors that are deemed by developers to be noisy or unimportant at a particular point of development. We identify unique build jobs by two attributes: the language runtime version (e.g., `Ruby: 2.1`) and the integration environment (e.g., `DB=mysql2`). If a build has multiple broken jobs, we consider the build to have an *Allowed* breakage if all the broken jobs are later excluded from the build. Builds in which only a subset of the broken jobs are excluded (i.e., other jobs are fixed by developers) are considered to have *Developer* breakages.

4.5 Experimental Results

This section discusses the results of our research questions.

4.5.1 RQ$_{4.1}$: What is the proportion of noisy build breakages?

Motivation. Despite the research invested on build breakages [48, 57, 85, 89], there is a lack of awareness about *Environmental*, *Cascading*, and *Allowed* breakages. These types of build breakages can potentially taint the current conclusions about build breakages that exist in the literature. Therefore, it is important to study the proportion and frequency of build breakages that can potentially introduce noises in build breakage data.

Approach. To address this RQ, we analyze the results obtained from the build breakage categorization and labeling using the three proposed criteria proposed in Section 4.4. We use the breakage labels at both the job and build levels. We report statistics about the proportion of *Environmental*, *Cascading*, and *Allowed* breakages in our studied projects. In addition, we report the ratio and frequency of each (sub)category of *Environmental* build breakages. We also perform additional manual analyses to gain more insights about the identified noisy build breakages.

Findings. ***About* 55% *of build breakages are environmental, cascading, and/or allowed.*** Fig. 4.2 shows the ratios of *Environmental*, *Cascading*, and *Allowed* build breakages in our dataset. As Fig. 4.2 indicates, there is an overlap of 17% between the three types of breakages. For example, ignoring environmentally broken builds may cascade the breakage to the next builds. Likewise, jobs with *Environmental* breakages, which are allowed to fail later, may generate *Cascading* breakages (in addition to *Allowed* breakages).

Environmental build breakages occur in* 39% *of the broken jobs. We observe that 39% of the analyzed broken jobs (i.e., $126,673$ out of $321,855$) are impacted by environmental factors. At the build level, we observe that 33% of broken builds in the studied projects (a median of 30%) are broken due to environmental factors. Approximately, one-third of environmentally broken builds experienced test failures, suggesting that such builds may contain flaky tests.

The statuses of* 29% *of builds are likely due to cascading breakages. Our results show that 29% of builds in our dataset are unlikely to be caused by the developers who triggered those particular builds. In addition, 76% of these builds are triggered by different developers from the developers who first introduced the breakage.

About* 9% *of broken builds are allowed breakages. We identify $2,022$ (i.e., 8%) of the unique jobs in the studied projects that are later excluded from broken builds (15% marked as *allow_failures* and 85% completely removed from the builds). Although we find that 33% of broken builds in our dataset contain excluded jobs, only 9% of such builds are primarily broken by those jobs. To understand the reasons behind job exclusion, we manually analyze a statistical sample of 62 commits (a confidence level of 95% and a confidence interval of ±10%) in which developers mark jobs as *allow_failures*. We find that developers identify the excluded jobs to be noisy (or flaky) in 68% of the cases (e.g., "*CI: allow JRuby build to fail, too flaky to be useful*"[8]). In 16% of the cases, developers express their inability to fix the breakage (e.g., "*Sick of jruby breaking travis*"[9]) or indicate that the breakage is unimportant (e.g., "*allow jruby 1.9 mode to fail, cuz I don't care*"[10]). In 16% of the cases, developers do not provide clear reasons why they exclude jobs (e.g., "*[travis-ci] allow failures for truffle for a green build*"[11]). Therefore, we advise researchers to filter builds affected by those jobs out, since the breakages could be due to the abnormal behavior of the excluded jobs.

There are 60 ***(sub)categories of Environmental build breakages.*** Table 4.3 presents the ratio and frequency of each *Environmental* (and *Suspicious*) breakage identified in our dataset. Each breakage subcategory represents the potential cause of the *Environmental* breakage. The last column of Table 4.3 shows the number of projects in which a breakage (sub)category exists. We obtain a strong inter-rater agreement (i.e., $k = 0.82$) with a strong observed agreement of 0.93 in manually validating the automated labeling and categorization of build breakages. We observe that, in 40% of the cases in which there was a disagreement with the automated breakage categorization, build logs are unexpectedly terminated (e.g., *logging stopped progressing*). In such cases, build logs are automatically categorized

[8]https://travis-ci.org/rails/rails/builds/125719232
[9]https://travis-ci.org/middleman/middleman/builds/1641894
[10]https://travis-ci.org/teamcapybara/capybara/builds/1923320
[11]https://travis-ci.org/jruby/jruby/builds/94375071

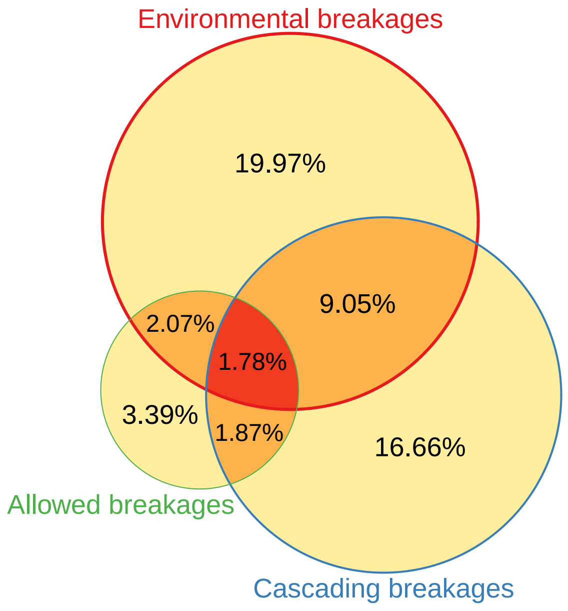


Figure 4.2: Ratios of *Environmental*, *Cascading*, and *Allowed* build breakages

according to the last logged error messages. However, since the termination of such build logs is abnormal (i.e., the underlying cause of breakage is unknown), we consider them as *Suspicious*. In addition, 32% of disagreement was related to error messages that are associated with other preceding error messages (e.g., '`bundle: command not found`' should precede '`Command failed with status (127)`').

The majority (i.e., 78%) of Environmental build breakages are caused by internal CI errors or issues related to external connections and exceeding limits. Internal CI errors introduce about 39% of the total *Environmental* breakages. We identify 18 reasons that may influence internal CI errors and, consequently, break CI builds. We observe that both the *Connection issues* and *Exceeding*

Table 4.3: Categories of *Environmental* build breakages

Category	Subcategory	#	%	*Freq.*
Internal CI issues	**Unidentified branch/tree/commit**	29,297	23.10%	132
	Failure to fetch resources	7,658	6.00%	106
	Error building gems	7,107	5.60%	58
	Logging stopped progressing*	1,632	1.30%	90
	Error fetching CI configuration	1,092	0.90%	49
	Error finding gems	753	0.60%	27
	Cannot execute git command	506	0.40%	24
	Multithreading issues	405	0.30%	3
	Unknown Travis CI error	241	0.20%	34
	Build script compilation error	142	0.10%	8
	Cannot access GitHub	883	0.70%	87
	Empty log*	65	0.00%	20
	Caching problems	44	0.00%	5
	Writing errors	32	0.00%	2
	Remote repository corruption	20	0.00%	2
	Cannot allocate resources	15	0.00%	3
	Storage server offline	9	0.00%	2
	Path issues	7	0.00%	3
	Total	**49,908**	**39.40%**	**152**
Exceeding limits	Stalled build (not output received)	16,798	13.30%	136
	Command execution time limit	3,816	3.00%	78
	Log size limit	3,613	2.80%	52
	Test running limit	2,691	2.10%	59
	Time limit waiting for response	259	0.20%	12
	Job runtime limit	234	0.20%	7
	API rate limit	67	0.00%	3
	Total	**27,478**	**21.70%**	**144**
Connection issues	**Connection timeout**	4,763	3.80%	95
	Connection refused, reset, closed	4,716	3.70%	111
	Server or service unavailable	3,370	2.70%	89
	Broken connection/pipes	3,072	2.40%	23
	Unknown host	2,927	2.30%	16
	Connection credentials error	2,242	1.80%	11
	Remote end hung up unexpectedly	377	0.30%	36
	Network transmission error	177	0.10%	26
	SSL connection error	152	0.10%	17
	Connection, proxy, & sync errors	79	0.10%	9
	SSL certificate error	18	0.00%	9
	Total	**21,893**	**17.30%**	**140**
Ruby & bundler issues	No compatible gem versions	5,521	4.40%	36
	Cannot find, parse, execute gems	2,477	2.00%	51
	Command loading failure	2,839	2.20%	52
	Bad file descriptor	1,515	1.20%	5
	Dependency request error	1,009	0.80%	36
	Bundler not installed	873	0.70%	35
	Total	**14,234**	**11.20%**	**81**
Memory & disk issues	**Out of memory/disk space**	3,395	2.70%	45
	Segmentation fault	1,924	1.50%	41
	Core dump problems	1,301	1.00%	33
	Memory stack error	540	0.40%	18
	Corrupted memory references	7	0.00%	2
	Total	**7,167**	**5.70%**	**84**
Platform issues	**Language installation issues**	1,785	1.40%	59
	Invalid platform	13	0.00%	2
	Unexpected failure	4	0.00%	1
	Total	**1,802**	**1.40%**	**60**
Virtual Machine issues	**Improper VM shut down**	1,114	0.90%	51
	VM creation error	82	0.10%	10
	VM connection problem	17	0.00%	7
	Invalid VM state	12	0.00%	4
	Stalled VM	11	0.00%	6
	Total	**1,236**	**1.00%**	**61**
Accidental abruption	**Build crashes unexpectedly**	2,182	1.70%	39
	Total	**2,182**	**1.70%**	**39**
Buggy build status‡	**Build exited successfully***	425	0.30%	28
	Jobs passing but build broken*	114	0.10%	20
	Total	**539**	**0.40%**	**44**
Database (DB) issues	DB creation quota	159	0.10%	1
	DB connection error	37	0.00%	2
	Total	**196**	**0.20%**	**2**
External bugs	**E.g., interpreter bugs**	38	0.00%	8
	Total	**38**	**0.00%**	**8**
Total Environmental Breakages		**126,673**	**100%**	**152**

∗ Suspicious build breakages, i.e., the cause of the breakage is unidentified

‡ Issues reported to Travis CI [12,13,14]

– **Bold** subcategories represent breakages that are likely to be exclusively environmental *limits* categories form around 39% of *Environmental* build breakages. Connection-related breakages mostly occur due to requesting data from external servers. Limits are usually set by TRAVIS CI to prevent builds, commands, and tests from running forever. Developers may reduce the amount of logged information in order to not exceed the TRAVIS CI log size limit of 4 MB. Developers may also customize build configuration to fix certain *Environmental* breakages. For example, the `travis_wait` attribute allows developers to let TRAVIS CI wait for commands that may take longer than the default 10 minutes.[15] Developers may also fix issues related to dependencies or long-running commands. However, in many cases, developers have no other way to fix *Environmental* breakages than restarting builds. In Table 4.3, we highlight (in **bold**) the (sub)categories of *Environmental* breakages that are most likely to be exclusively outside the ability of developers to control. Moreover, we find that changing build configuration may not always be deemed as the cause or fix of a breakage. Therefore, we consider all (sub)categories of *Environmental* breakages to be noisy.

4.5.2 RQ$_{4.2}$: How do noisy breakages impact build breakage models?

Motivation. Uncleaned build breakage data can distort prediction/regression models as broken builds are assumed to be associated with the development process (e.g., triggering commits) and developer metrics. Not considering the breakages that can introduce noises to build breakage data may affect the accuracy of prediction/regression models and drive researchers to misleading conclusions.

[13]https://github.com/travis-ci/travis-ci/issues/646
[14]https://github.com/travis-ci/travis-ci/issues/891
[15]https://github.com/travis-ci/travis-ci/issues/2533
[15]https://docs.travis-ci.com/user/common-build-problems/#build-times-out-because-no-output-was-received

Approach. We use the build status (i.e., *passed* or *broken*) as a dependent variable in both models. We use a set of metrics (presented in Table 4.4) as independent variables in both models. These metrics have been used by prior research to study their association with build breakages [68, 76, 85, 107]. All these metrics are computed at the commit level. We compute the *previous build status* metric for the uncleaned datasets and recompute such a metric again for the cleaned dataset. For example, assume we have the builds $\{B_1$:'*passed*', B_2:'*broken*', B_3:'*broken*'$\}$ and the breakage of build B_2 is identified to be noisy. In such a case, the previous status of build B_3 is '*broken*' in the uncleaned data but '*passed*' in the cleaned data. We remove highly correlated variables, since they can adversely affect our models [26]. To this end, we use the `varclus` function (from the `rms`[16] *R* package) that performs the Spearman rank ρ [88]. For each pair of correlated variables that have a correlation of $|\rho| \geq 0.7$, we remove one variable and keep the other in our models. For example, we remove the *Configuration lines added* and *Configuration lines deleted* metrics, since they are highly correlated with the *Configuration files changed*. Similarly, we remove the *Source files changed* metric, since it is highly correlated with the *Files changed* metric. The number of builds in the uncleaned and cleaned datasets are $350,246$ and $296,982$, respectively.

Builds in our datasets are from different languages, projects, and ages. We use these variables as random effects in our models to control (i) the overrepresentation of Ruby projects, (ii) the variation between projects in terms of sizes and domains, and (iii) the potential impact of triggering builds at different stages of CI adoption on the obtained results. To this end, we use the Generalized Linear Mixed Models (GLMM) for logistic regression. GLMM uses mixed (i.e., fixed and random) effects

[16]https://cran.r-project.org/web/packages/rms/rms.pdf

Table 4.4: Metrics used as independent variables in our models

Metric	Data type	Used by	Description	Effect
Project name	Factor		The name of the project in which a build is triggered	Random
Language	Factor		The programming language of the project in which a build is triggered (Ruby or Java)	
Build age	Factor		The difference (in days) between a build starting date and the date of the first build in a project	
Core developer	Factor	[68, 85, 107]	The author(s) who triggered the build have committed at least once in the last three months	Fixed
Team size	Numeric	[68, 107]	The number of developers in the team	
Source churn	Numeric	[68, 107]	The number of source lines of code changed by the commits that trigger the builds	
Test churn	Numeric	[68, 107]	The number of test lines changed by the commits that trigger the builds	
Doc files changed	Numeric	[68, 107]	The number of documentation files by all build commits	
Other files changed	Numeric	[68, 76, 107]	The number of non production nor documentation files by all build commits	
Previous build status	Factor	[68, 76, 85, 107]	The status of the build that precedes the triggered build (*passed*, *broken*, or *NA*)	
Files changed	Numeric	[68, 76, 85, 107]	The number of files modified by all the build commits	
Source files changed	Numeric	[68, 76, 107]	The number of source files changed the commits that trigger the builds	
Configuration files changed	Numeric	[76]	The number of build configuration files (e.g., `.xml` and `.yml`) modified by the build commits	
Configuration lines added	Numeric	[76]	The number of lines added to configuration files	
Configuration lines deleted	Numeric	[76]	The number of lines deleted from configuration files	
Tests added	Numeric	[68, 107]	The number of added test cases	
Tests deleted	Numeric	[68, 107]	The number of deleted test cases	
Commits on touched files	Numeric	[68, 107]	The number of unique commits on the files touched by the commits that triggered the build	
Is a pull request	Factor	[68, 85]	Whether a commit belongs to a pull request or a Git push	
Time of day	Factor	[85]	The time-zone adjusted time of day of the commit that triggered the build	
Day of week	Factor	[85]	The time-zone adjusted day of week of the commit that triggered the build	
Belongs to the default branch	Factor	[49]	Whether the build is triggered by a commit that belongs to the default branch of the repository	

to investigate the variables that are associated with build breakages [36]. This means that our models assume a different random intercept for each category of the random effect [66]. We use the ANOVA test [79] to find the significant variables to model build breakages (i.e., variables that have $Pr(<|\chi^2|)$ less than 0.05). $Pr(<|\chi^2|)$ is the p-value that is associated with the χ^2 test, which shows if our model is statistically different from the same model in the absence of a given independent variable—according to the degrees of freedom in our model. We also use *upward* and *downward* arrows to indicate whether a variable has a direct or an inverse relationship, respectively, with build breakages.

We evaluate the performance of our models using the Area Under the receiver operating characteristic Curve (AUC), the marginal R^2, and the conditional R^2. The AUC evaluates the diagnostic ability of our mixed-effect models to discriminate *broken* builds [46]. AUC is the area below the receiver operating characteristic (ROC) curve created by plotting the true positive rate (TPR) against the false positive rate (FPR) using all possible classification thresholds. The value of AUC ranges between 0 (worst) and 1 (best). An AUC that is greater than 0.5 indicates that the explanatory model outperforms a random predictor. We use the method proposed by DeLong et al. [24] to compare the ROC curves of the models. We compute the number of Events Per Variable (EPV) for the uncleaned and cleaned datasets to investigate the likelihood of our models to be overfitting. EPV measures the ratio of the number of build breakages to the number of factors used as independent variables to train the models [78]. The EPV values of the cleaned and uncleaned datasets are $4,726$ and $2,127$, respectively (i.e., EPV ≥ 40). Hence, our models are considered stable and have reliable AUC values (i.e., the optimism is small) [96]. Moreover, our models are unlikely to have overfitting problems [78], since the EPV values are above 10. Higher values of the conditional R^2 indicate that the proportion of variance explained by both fixed and random effects is much higher than the proportion of variance explained by fixed effects only. A high difference between the conditional and marginal R^2 values suggests that random effects significantly help explain the dependent variable (i.e., the build breakage).

One could argue that the differences between the results of the two models could be influenced by the differences in the sizes of the cleaned and uncleaned datasets. To study this argument, we randomly remove sample builds from the original (uncleaned) datasets. The size of the removed sample builds is equal to the size of the builds identified as noisy. We repeat this process 10 times, each with a different random set of samples.

Findings. Table 4.5 shows the variable importance results obtained from our both mixed-effect logistic models. All the metrics are descendingly sorted based on the χ^2 values of the cleaned model. For each metrics, we show its χ^2 value, the p-value ($Pr(< \chi^2)$), its significance, and whether the variable has a direct or an inverse impact on build breakages (the *upward* and *downward* arrows).

***The model fit on the cleaned data significantly improves the AUC by* 6%.** Our model fit on the clean data obtains a good AUC value of 0.83 for discriminating build breakages. This AUC outperforms the model fit on the uncleaned data, which obtains an AUC value of 0.77. By comparing the ROC curves of the two models, we observe that the model fit on the cleaned data has a statistically significant improvement in discriminating build breakages (i.e., we obtain a DeLong's test p-value $< 2.2e^{-16}$). Moreover, when we remove build samples randomly from the

Table 4.5: Results of our mixed-effects logistic models — sorted by $Pr(<\chi^2)$ of the cleaned model

Metric	**Cleaned model**				**Uncleaned model**			
	χ^2	$Pr(<\chi^2)$	Signf.*	Rel.	χ^2	$Pr(<\chi^2)$	Signf.*	Rel.
(Intercept)	0.0176	$< 2.2e^{-16}$	***	-	0.0623	$< 2.2e^{-16}$	***	-
Previous build status	0.9672	$< 2.2e^{-16}$	***	-	0.5550	$< 2.2e^{-16}$	***	-
Is Pull Request	0.0044	$< 2.2e^{-16}$	***	↗	0.0820	$< 2.2e^{-16}$	***	↗
Belongs to the default branch	0.0040	$< 2.2e^{-16}$	***	↘	0.2425	$< 2.2e^{-16}$	***	↘
Source churn	0.0036	$< 2.2e^{-16}$	***	↗	0.0270	$< 2.2e^{-16}$	***	↗
Commits on touched files	0.0016	0.000	***	↗	0.0019	0.014	*	↗
Day of week	0.0007	0.005	**	-	0.0170	0.000	***	-
Files changed	0.0004	0.001	**	↗	0.0018	0.016	*	↗
Triggered at night	0.0002	0.025	*	↘	0.0009	0.089	.	↘
Configuration files changed	0.0001	0.070	.	↗	0.0002	0.415		↗
Team size	0.0000	0.259		↘	0.0045	0.000	***	↗
Other files changed	0.0000	0.272		↗	0.0008	0.108		↗
Tests added	0.0000	0.294		↗	0.0011	0.057	.	↗
Doc files changed	0.0000	0.445		↗	0.0002	0.427		↘
Core developer	0.0000	0.451		↘	0.0019	0.013	*	↗
Tests deleted	0.0000	0.458		↘	0.0001	0.531		↗
Test churn	0.0000	0.969		↗	0.0008	0.114		↘

*Significance codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

uncleaned data, we observe that the (i) average AUC obtained from all the models is 0.77 and (ii) the significant variables obtained from the uncleaned model hold in 90% of the models (results of all the generated models can be found in our online appendix [2]). This result suggests that the noises in build breakage data is likely to distort prediction/regression models for build breakages. In particular, not excluding builds with noisy breakages may influence models to generate incorrect associations between the independent variables and the build breakage.

The uncleaned breakage data generates models that are more sensitive to language, project, and build age variations. The marginal R^2 of the model fit on the cleaned data is 0.21 and the conditional R^2 is 0.36 (i.e., an increase of 71%). On the other hand, the marginal R^2 of the model fit on the uncleaned data is 0.07 and the conditional R^2 is 0.30 (i.e., 329% more). Such results suggest that the difference between the conditional and marginal R^2 values in the cleaned model is smaller than the difference in the model fit on the uncleaned data.

Build breakages have conflicting associations with the variables before and after cleaning the dataset. We observe that the model fit on the cleaned data produces contradicting results compared to the model fit on the uncleaned data. The significant variables observed in the uncleaned data are in agreement with the results reported in prior studies on modeling build breakages [85]. However, according to the model fit on the cleaned data, the vast majority of the explanatory power of the model comes from the status of the previous build (i.e., $\chi^2 = 0.97$). The explanatory power of the status of the previous build in the uncleaned data is most likely reduced due to noisy build breakages. In conclusion, our findings using the cleaned data disagree with the findings of prior research, in the sense that:

- builds triggered at night are less likely to produce build breakages.
- being a core developer does not have a strong association with build breakages.
- team size is unlikely to be associated with build breakages.

4.5.3 RQ$_{4.3}$: How do noisy breakages impact prior findings?

Motivation. Uncleaned build breakage data can distort prediction/regression models as broken builds are assumed to be associated with the development process (e.g., triggering commits) and developer metrics. Not considering the breakages that can introduce noises to build breakage data may affect the accuracy of prediction/regression models and drive researchers to misleading conclusions.

Approach. We use the data used by Rausch et al. [85] containing 121,258 builds from 14 projects. We apply our criteria to identify *Environmental*, *Cascading*, and

Allowed breakages in the dataset. Then, we process the data similarly to Rausch et al., i.e., (a) we stratify the data by removing builds with unknown previous builds; and (b) we remove data that are outside the 99^{th} percentile of numeric metrics (e.g., the number of commits or lines added). After that, instead of removing builds that perpetuate or fix build failures (as Rausch et al. have done), we remove builds that have *Environmental*, *Cascading*, and *Allowed* breakages. Our results are unlikely to be impacted by the differences in the sizes of the uncleaned and cleaned datasets, since Rausch et al. also filtered out builds from the original dataset. Finally, we perform the statistical tests used by Rausch et al. to study the association between build breakages and 16 metrics. Similarly to Rausch et al., we exclude inconclusive results obtained from projects that have insufficient data for some metrics (e.g., missing build types). Finally, we compare our observations on the cleaned version of the data used by Rausch et al. with their reported findings.

Findings. ***The use of noisy build breakages may have a considerable impact on the observations reported by prior research.*** We compare the findings reported by Rausch et al. [85] with our observations, as follows:

- "*The day of week has no significant influence on build breakages, with little evidence that the time of day influences build breakages.*" However, we observe that builds triggered on weekends have significantly less breakage ratios than builds triggered on weekdays in 80% of the projects. We also observe that late night commits cause fewer breakages in 75% of the projects.

- "*Despite the significant impact of file type changes on build breakages, there is no evidence that certain file types lead to breakages more frequently.*" However, we observe that commits that change *system* files (either alone, with *test*, or with *configuration* files) cause more breakages in 60% of the projects.

- "*Pull requests cause more breakages.*" However, we observe that, in 67% of the projects, integration and merge commits generate builds with higher breakage ratios than any other commits. A possible reason behind such higher breakage ratios is the incompatibility or conflicts that may occur when integrating or merging development branches.
- "*Developers who commit less frequently cause fewer build breakages (in* 4 *out of* 6 *projects).*" However, we observe that there is a slight difference between the cases in which developers who commit daily or less frequently cause fewer breakages. In particular, we observe that both daily and less frequently committers cause significantly fewer breakages in 4 and 5 out of 9 projects, respectively. This result suggests that frequently committing is likely to be associated with other factors (e.g., complexity of changes) to impact on build breakages.

4.6 Discussion

Build breakage data has been used as the foundation for several empirical conclusions regarding CI [48, 49, 61, 76, 85, 107], which is widely used in open source and industry settings. In this section, we discuss how our findings of noisy build breakage data may lead to direct implications for CI researchers and tool builders.

4.6.1 Implications for researchers

Researchers should be more careful about the quality of historical build breakage data. Researchers have invested a considerable effort to studying CI builds. Hilton et al. [48] found that build breakages introduce a barrier that hinders developers from adopting CI. Hence, studies on CI builds have been conducted to help development teams overcome that barrier. A build breakage may occur due to several reasons, such as configuration errors, installation errors, compilation errors, or test failures [85, 117]. Researchers also studied the possibility of modeling build breakages to understand the factors that share a strong relationship with CI build breakages [61, 76, 85, 107]. However, our results show that build breakages can be noisy and have a great impact on the analyses of CI build breakage data. Therefore, we advise future research to revisit the prior analyses regarding CI build breakages to verify whether the prior observations and insights would change considerably after taking noisy builds into account.

4.6.2 Implications for tool builders

Feedback on CI builds should be enriched with more information about the breakage. Existing CI tools provide developers with an abstract build status (i.e., *passed* or *broken*). Tool builders should consider supplying developers with an enhanced user interface to help them understand (a) the differences between individual job breakages, (b) the types of build errors occurred, (c) whether the build would possibly pass if restarted, (d) whether certain build breakages occurred in the past, and (e) what actions developers have previously made to fix that breakages. Build logs are quite rich of information about the reasons behind build breakages. Therefore, such information may be regularly collected by a tool to build a history of the frequent root causes of build breakages. In addition, if builds are frequently broken due to connection errors, a useful tool may consider increasing the number of times to rerun failing commands. Likewise, if builds are frequently broken due to exceeding time or log size limits, a tool may consider increasing the time required to wait for CI commands to run or removing redundant or unnecessary information from being logged, respectively.

4.7 Threats to Validity

This section discusses the threats to the validity of our study.

Construct validity. Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. We use open coding to categorize and label build logs of broken jobs, which can be subjective. To mitigate such a threat, multiple iterations of manual analysis are performed by two co-authors of this work to validate the build log labeling and categorization using statistically significant samples of build logs. To identify *Environmental* build breakages, we rely on the last error message in the build logs.If the `script` phase fails, the build continues to run the subsequent phases (e.g., `after_success`, `after_failure`, or `after_script`).[17] Errors in the subsequent phases do not impact the build status, except for timeout errors. Hence, the last error message in the build log might not represent the actual breakage cause. To mitigate this threat, we validate the results of $1,512$ cases (i.e., 1.6%) of environmentally broken builds that are configured to implement the aforementioned phases. We find that: 63% of the builds are *errored* (i.e., errors in the `install` phase), 15% are virtual machine issues, memory issues, or abrupt crashes, 14% are connection or resources issues 6% are failed due to '`unknown host nexus.codehaus.org`' (an issue reported to TRAVIS CI[18]), and 2% are timeout

[17]https://docs.travis-ci.com/user/job-lifecycle#breaking-the-build
[18]https://github.com/travis-ci/travis-ci/issues/4629

errors. In addition, after investigating a statistically significant sample of 81 cases (a confidence level of 95% and a confidence interval of ±10%), we find that the vast majority (i.e., 96%) of the errors occurred during the *script* phase. Hence, errors that occur in the phases that follow the *script* phase are unlikely to impact our obtained results.

We consider a build breakage to be *Environmental* if the error is documented or reported to be related to environmental issues. Nevertheless, developers may commit code changes that may likely induce environmental breakages. For example, adding expensive testing code (e.g., poor choice of algorithms) may potentially lead to time-exceeding errors, which may break the build unexpectedly. To mitigate such a threat, we perform additional manual analyses on statistically significant samples of environmental build breakages (95% confidence level and ±10% confidence interval) that might possibly be caused by developers. Overall, we find less than 2% of *Environmental* breakages in which developers change build configuration (i.e., script file), with only 0.2% not being identified as *Cascading* or *Allowed* breakages. We analyze 58 builds having build script compilation errors. We observe that, in 93% of the cases, developers do not change the script file. We also analyze 92 builds having tests that timed out due to exceeding predefined limits. We observe that developers add more expensive code (e.g., loops) only in 9% of the cases. Of such cases, we find that only one case in which a passing build that follows a breakage contains code changes that fix the issues of long-running tests. Moreover, we analyze a similarly statistical sample of builds that have dependency-related breakages. We find that developers do not add or change dependencies in 89% of the cases. Such results suggest that environmental build breakages are most likely to be related to environmental issues rather than development activities.

Developers may exclude jobs at later stages (i.e., in builds that are beyond our dataset). Predicting whether a developer will exclude a certain job at a later stage is out of the scope of this study. Therefore, we assume that a build breakage is an *Allowed* breakage if the dataset has evidence that the jobs breaking the build are excluded by developers later. Moreover, we manually investigate the reasons behind excluding build jobs. We find that developers identify the exclude jobs to be noisy or flaky in 69% of the time. However, identifying a job to be noisy at a certain point of time may not imply that the job was noisy in the project all the time. Still, we advise researchers to filter builds that are primarily broken by such jobs out, since the breakages could be due to the abnormal behavior of the excluded jobs.

All of reported results and statistics about noisy build breakages are computed using our written Python and R scripts. Our scripts may contain defects that might affect the reported results. To address this threat, we perform additional manual analyses to verify the results of each of our proposed criteria.

Internal validity. Internal threats to validity are concerned with the ability to draw conclusions from the relationship between build breakages and other build characteristics. Our study does not consider *passed* builds as noisy. However, in rare cases, RSpec in Ruby may sometimes return 0 due to overwriting the `at_exit` handler by different gems.[19] Our scope in this study is to focus on the factors that may lead to noisy breakages. We leave the investigation of the potential noises in *passed* builds for future work.

External validity. External threats are concerned with our ability to generalize

[19]https://docs.travis-ci.com/user/common-build-problems

our results. Our study investigates noises in build breakage data of 153 projects that have enough samples (i.e., over $1,000$ builds per each project) to perform our analyses. Although the studied projects are of different languages (i.e., Ruby and Java), code sizes, team sizes, and domains, we cannot generalize our conclusions to other projects where these settings heavily vary. For example, we may observe different kinds of build breakages and findings if we analyze data of projects written in other programming languages (e.g., Python or JavaScript) or linked with different CI build services (e.g., Circle CI or Appveyor). Replication of this work using additional software projects and other CI services is required in order to reach more general conclusions.

4.8 Summary

In this chapter, we conduct an empirical study to investigate the noises that may exist in build breakage data. We define three criteria to identify noises in build breakage data. In particular, our criteria identify the builds that have (1) *Environmental* breakages, (2) *Cascading* breakages, and (3) *Allowed* breakages. We first clean our studied dataset and propose a catalogue of the categories of possible *Environmental* build breakages in CI. Second, we measure the impact of using noisy build breakage data on modeling build breakages. We observe the following:

- 55% of broken builds in our dataset are impacted by *Environmental*, *Cascading*, and/or *Allowed* breakages.
- Internal Travis CI errors, connection issues, and exceeding limits issues cause most (i.e., 78%) of *environmental* breakages in CI builds.
- Noisy build breakages reduces the of performance of build breakages models.

- Noisy build breakages distort the association between build breakages and other metrics.

- observations reported by prior research may not hold and researchers may gain misleading insights if noisy build breakages are filtered out from build breakage data.

Our study suggests that researchers and practitioners should be more careful when they deal with build breakage data. In particular, the build statuses (i.e., *errored*, *failed* and *passed*) provided by CI services are not reliable enough to judge that builds are broken due to development activities. In fact, builds may be broken due to environmental factors or simply cascade previous (unfixed) breakages. Therefore, using such noisy data may consequently lead to lower performance and incorrect observations when modeling build breakages. Although our study identifies the proportion of noise in build breakage data, we observe that projects still have long sequences of consecutive breakages.

We aim in the future to conduct a qualitative study to investigate other reasons behind not fixing build breakages as soon as they occur. Furthermore, we aim to study the common development practices to deal with build breakages.

Chapter 5

A Study of the Build Durations and Breakages Interplay in CI

This chapter describes our study of the possible interplay between reducing build durations and fixing/avoiding build breakages. Section 5.1 presents the research problem and motivation of our study. Section 5.2 presents the study design of our empirical study. Section 5.3 presents the results and findings of our study. Section 5.4 discusses the implications of our findings for researchers and tool builders. Section 5.5 describes the threats to the validity of our results. Finally, Section 5.6 summarizes the chapter and suggests future work.

5.1 Problem and motivation

Long build durations and frequent build breakages introduce overhead to the software development process. Waiting until a build completes and fixing build errors or failures, in the case of build breakages, may impede developers from engaging in other development activities [103]. Moreover, CI resources can be excessively consumed if builds take longer or re-triggered more than once after fixing build breakages [49].

Therefore, it is important to understand the best practices that enable developers to maintain both timely and successful builds.

Previous research has conducted independent analyses on either build durations or build breakages. In particular, some studies have investigated the impact of long build durations on software development [18, 84, 86] and proposed recommendations to reduce the build duration [8, 43, 48, 86]. Other studies have also proposed approaches to model and predict build breakages [44, 76, 85, 107]. However, these studies performed independent analyses to study build durations and breakages in software projects [38].

Despite the valuable insights produced by prior research, little is known about the possible interplay between reducing build durations and fixing build breakages. In particular, it is unclear from prior studies (i) whether actions to reduce build durations would reduce build breakages and (ii) whether fixing a build breakage would lead to longer build durations, or vice versa. For example, configuring a build to rerun failed commands multiple times may help avoid undesirable build breakages but, in return, may prolong builds.[1] Therefore, it is important for developers to understand the potential dual or side effects when taking actions to optimize CI builds (i.e., reduce build durations or fix build breakages).

The goal of this study is to study the interplay between *build durations* and *build breakages*. We investigate the dual and side effects of improving one performance measure of CI builds (e.g., the *build duration*) on the other measure (e.g., the *build breakage*). To this end, we extend an existing dataset called TravisTorrent [15] to (a) exclude inactive projects and (b) collect recent builds of active projects. As a result, we conduct an empirical study on 924, 616 CI builds from 588 GitHub projects that

[1] https://github.com/travis-ci/travis-ci/issues/3031

are linked with Travis CI. In addition, we survey developers who have contributed to the projects in our dataset to get their feedback on our empirical observations. To model the interplay between build durations and breakages, we use them as two dimensions to group the studied projects into four quadrants (or states) as shown in Figure 5.1: Dominantly Timely/Passing (the majority of the builds finish timely and pass), Dominantly Long/Passing (the majority of the builds pass but take longer to finish), Dominantly Timely/Broken (the majority of the builds finish timely but break), and Dominantly Long/Broken (the majority of the builds break and take longer to finish). Then, we perform project-level and build-level analyses of each quadrant to explore factors associated with build durations and breakages. Based on the above goal, we address the following RQs:

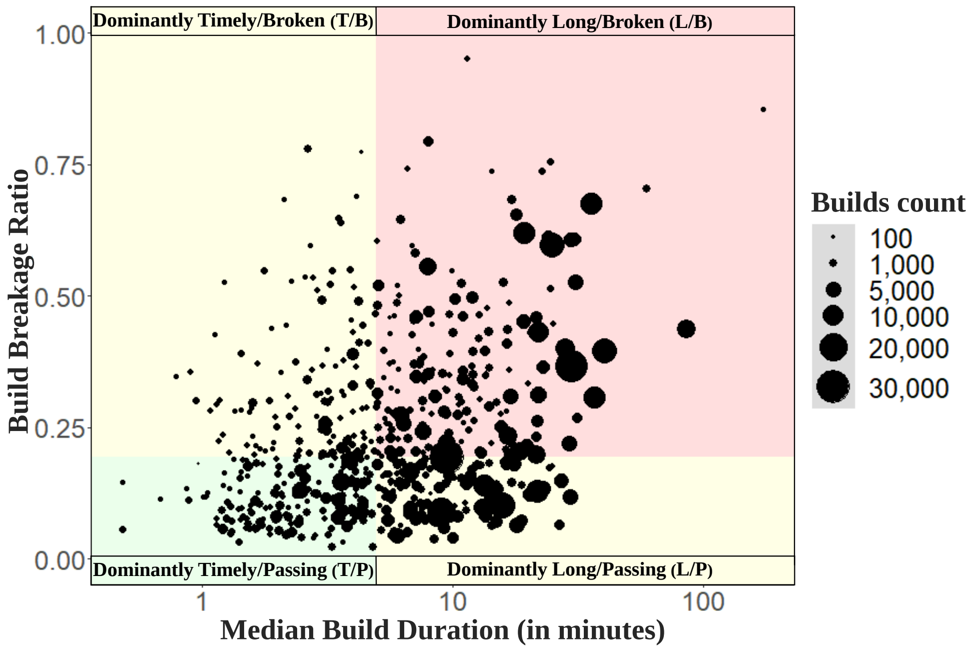


Figure 5.1: Quadrants of the studied projects (each point is a project) using the median build duration (x-axis) and the breakage ratio (y-axis)

$\boldsymbol{RQ_{5.1}}$: **What project characteristics are associated with build performance?**

Previous studies report inconsistent findings on CI builds [32, 95]. However, little is known about whether the characteristics of projects are associated with build performance. Our analysis of 18 project-level metrics shows that the context of a project (e.g., being a large or active project) has a strong association with frequently long and broken builds. Developers are encouraged to change the build configuration only when needed, e.g., to optimize or fix problems related to build performance. Researchers should also take into account the differences between projects when studying build durations and breakages.

$\boldsymbol{RQ_{5.2}}$: **What build-level metrics are associated with build durations and/or breakages?**

Previous research has paid little attention to the metrics that are associated with both build durations and build breakages. In this RQ, model each quadrant with respect to the other quadrants using 40 build-level metrics. We find frequent committers are more careful about build breakages than occasional committers. Developers should keep an eye on their builds, since certain build configurations, such as changing infrastructure or retrying commands, can lead to undesirable build performance.

$\boldsymbol{RQ_{5.3}}$: **How should developers act to optimize build performance?**

When projects encounter long build durations and/or build frequent breakages, developers need to act toward optimizing CI builds. In this RQ, we explore the possible actions that developers can take to improve build performance. We observe that CI practices are more associated with build performance than development practices. We also find that experienced developers have a positive effect when they join a project, making their contributions more valuable towards build performance.

***$RQ_{5.4}$*: How frequently does build performance change over time?**

Experiencing ups and downs in build performance of a project at a certain point in time can indicate a change to the development process. In this RQ, we identify 16 patterns in which projects may undergo changes to build performance. We find that over two-thirds of the projects have had persistent build performance over time. Moreover, we observe that changes to the build duration or breakage over time may be associated with changes to the development practices. Therefore, developers are encouraged to constantly explore ways to improve CI builds.

Overall, our developer survey shows that software developers agree with the majority of the findings of our studied RQs. Feedback from survey respondents indicates that, while build performance could be improved using workarounds, developers should rather focus on addressing the root causes of problems.

5.2 Study Design

This section presents the experimental setup of our empirical study. We explain how we collect and prepare the data for our studied RQs.

5.2.1 Data Collection

Figure 5.2 shows an overview of our study. Our study is based on data collected from TravisTorrent [15], a commonly used dataset to study CI builds [14, 68, 76].

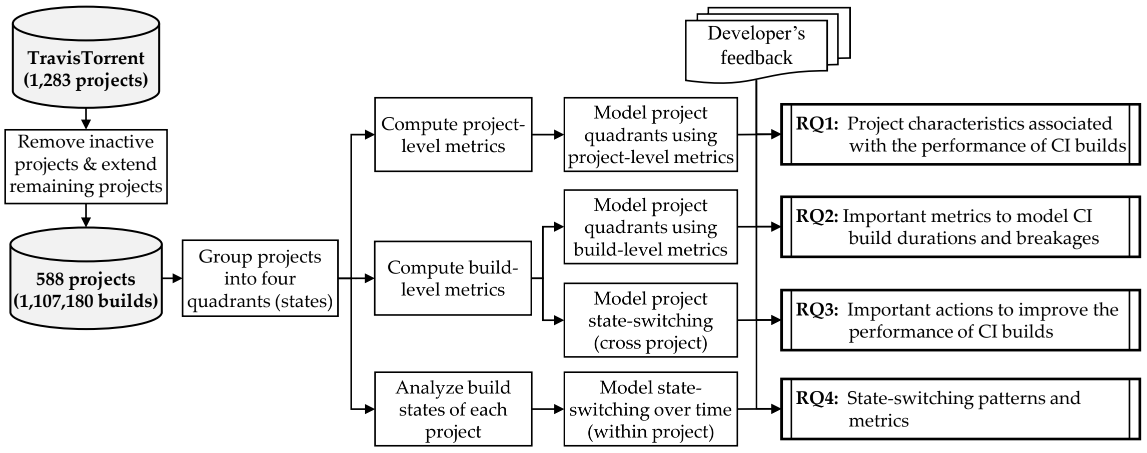

TravisTorrent (1,283 projects)
Remove inactive projects & extend remaining projects
588 projects (1,107,180 builds)
Group projects into four quadrants (states)
Compute project-level metrics
Compute build-level metrics
Analyze build states of each project
Model project quadrants using project-level metrics
Model project quadrants using build-level metrics
Model project state-switching (cross project)
Model state-switching over time (within project)
Developer's feedback
RQ1: Project characteristics associated with the performance of CI builds
RQ2: Important metrics to model CI build durations and breakages
RQ3: Important actions to improve the performance of CI builds
RQ4: State-switching patterns and metrics

Figure 5.2: Overview of our empirical study of the interplay between build durations and breakages

TravisTorrent contains builds from 1,283 projects: 886 Ruby, 393 Java, and 4 JavaScript projects. We exclude the 4 JavaScript projects, since they cannot be used as a representative sample of the Javascript language. The last build in TravisTorrent was triggered in *August 31, 2016*. Hence, we update the TravisTorrent dataset by collecting recently triggered builds up to *July 17, 2020*. We exclude the projects that (a) are longer available in GitHub, (b) stopped using Travis CI), or became less active (i.e., having less than 50 builds [15] during the updated period). As a result, we obtain 588 projects that actively use Travis CI. We update the list of builds for each of those projects using Travsi API[2] and collect the corresponding build metrics from GHTorrent.[3] We exclude *started* and *canceled* builds, since they are incomplete. The total number of builds of the projects in our dataset is 924,616.

TravisTorrent contains information about build durations and breakages. The build duration in TravisTorrent is computed using the sum of durations of all

[2]https://docs.travis-ci.com/api
[3]https://ghtorrent.org

build jobs. Using such duration values could be misleading, since build jobs can run in parallel on TRAVIS CI. Therefore, we compute the actual build duration by taking the difference between the starting time and finishing time of each build [43]. Besides, we clone the GIT repository of each project to compute additional project-level and build-level metrics (e.g., project age and developer experience). We also analyze build configuration files (i.e., *.travis.yml*) to compute metrics related to build configurations. Moreover, we collect CI metrics about builds (e.g., build integration environments) from TRAVIS CI.

5.2.2 Data Processing

In this section, we explain how we process the data of the 588 projects.

Grouping the studied projects into quadrants.

Each build in our dataset has two performance measures representing (i) the *build duration* (continuous values) and (ii) the *build breakage* (categorical: *broken* or *passed*). To group the studied projects, we compute the dominant performance of each project as follows.

- We summarize the build durations of each project in our dataset by taking the median build duration.
- We summarize build breakages by computing the proportion of broken builds of a project divided by the total number of builds of that project.

To visualize the studied projects, we plot the build breakage ratio of each project against the median build duration of that project. Then, we split the plot into four quadrants. We use the median value of every performance measure (i.e., the build breakage ratios and the median build durations) of all projects as thresholds to split the quadrants. The median measure is robust as it is not heavily influenced by outliers [109]. Figure 5.1 shows the distribution of the studied projects across the four quadrants. The x-axis represents the median build durations of the studied projects. The y-axis represents the build breakage ratios of the studied projects. Each point on the plot represents a project in our dataset. The position of each point (i.e., project) is based on the build breakage ratio and the median build duration of the project. The size of a point represents the number of builds of the projects. We use the median breakage ratio (i.e., 20%) and median build duration (i.e., 6 minutes) across all projects to group projects into the quadrants. A project may belong to one of the following quadrants:

- The lower-left quadrant (*dominantly timely/passing 'T/P'*): comprises 176 projects in which the majority of the builds finish timely and pass.
- The lower-right quadrant (*dominantly passing/long 'L/P'*): comprises 118 projects in which the majority of the builds pass but take longer to finish.
- The upper-left quadrant (*dominantly timely/broken 'T/B'*): comprises 118 projects in which the majority of the builds finish timely but break.
- The upper-right quadrant (*dominantly long/broken 'L/B'*): comprises 176 projects in which the majority of the builds break and take longer to finish.

Sensitivity analysis. Categorizing projects into quadrants using the entire build history might not be realistic. Therefore, in addition to using the entire build history of a project, we produce the quadrants for each quarter of the project lifetime. As a result, a project can belong to four different quadrants across its lifetime. For example, a project with a four-year lifetime can belong to a different quadrant every year. We analyze whether projects undergo changes between quadrants during their lifetime.

Project-level metrics. The build performance can be affected by the characteristics of software projects [95]. Therefore, it is important for developers to understand whether certain characteristics are associated with the build durations and breakages of a project. We collect project-level information about the studied projects. We study the importance of project characteristics in modeling the build performance by considering five facets or project-level metrics, as follows.

- *Programming Language:* Projects from different languages may adopt CI differently. We study whether the performance of Ruby builds differ from that of Java builds.

- *Project Maturity:* Mature projects can have a different building experience from those projects that are still at their early stages of development. We study whether project maturity (e.g., project age and test density) relates to the build performance.

- *Development Activity:* Projects have different levels of activities (e.g., committing more frequently). We study whether more active projects generate more successful builds than projects with fewer activities.

- *CI Activity:* Developers may need to maintain builds very often to cope with code changes. We study whether CI activities (e.g., frequent build configurations) are associated with the build performance.

- *Project Reputation:* Projects desire to generate acceptable CI builds as much as maintaining a better reputation. We study whether the reputation of projects (e.g., the number of stars) is associated with the build performance.

For each facet of project characteristics, we compute a set of project-level metrics (i.e., a single metric value for each project). In total, we compute 18 project-level metrics. Table 5.1 gives details about the project-level metrics and how they are computed. We use the project-level metrics to model the differences between projects across the four quadrants.

Table 5.1: Description of the project-level metrics used in our logistic regression models

Project characteristic	**Project-level metric**	**Description**
Programming Language	Language	The GitHub dominant programming language of a project
Code Maturity	Project age	Time difference (in terms of days) between the last build and project creation date
	Size (SLOC)	Number of source lines of code of a project
	Test density	Median number of test cases per $1,000$ SLOC of a project
Development Activity	# of commits per lifetime	Ratio of commits per a project lifetime
	Growth rate	Ratio of the relative increase/decrease (i.e., delta) in the lines of code
	# of branches	Number of branches of a project
	Unique developers	Number of unique developers contributed to a project
	Team size	Median team size at each build of a project lifetime
CI Activity	CI lifespan	Time difference (in terms of days) between the last and first builds of a project
	# of builds	Number of builds triggered by a project
	Building frequency	Frequency (in terms of days) of triggering builds by a project
	Configuration ratio	Ratio of commits that change build configurations per a project lifetime
	Configuration frequency	Frequency (in terms of days) of changing build configurations of a project
	Unique build environments	Number of unique integration environments have been used as build jobs in a project
	# of build jobs	Median number of jobs per build of a project
Reputation	# of stars	Number of GitHub stars of a project (i.e., being a favorite project)
	# of forks	Number of GitHub repository forks of a project

Build-level metrics. Despite the importance of project-level metrics, developers may be less capable to control certain project characteristics (e.g., the programming language). Therefore, in our work, we study the association of a set of lower-level metrics (i.e., at the build level) with the perceived build durations and breakages.

Differently from project-level metrics, each build of the studied projects has a single value for each of the build-level metrics. In total, we compute 40 exploratory metrics about the builds in our dataset across three facets: code, CI, and developer metrics. Table 5.2 presents a detailed description of the build-level metrics, their data types, and how they are computed. We use the build-level metrics to model build states and how projects switch from one state to another state. In particular, we aim to understand which build-level metrics are associated with build states and what actions developers can take to switch a project to a better build state.

Correlation and Redundancy Analysis. We use the computed project-level and build-level metrics as independent variables to fit our logistic regression models (See Section 5.3: *$RQ_{5.1}$*, *$RQ_{5.2}$*, and *$RQ_{5.3}$*). We first exclude the highly correlated independent variables, since they can adversely affect regression models [26]. We follow the guidelines provided by Harrell [47] on regression modeling. In particular, we use the Spearman rank ρ clustering analysis [88] (using the `varclus` function from the `rms`[4] *R* package) to identify highly correlated variables. For each pair of correlated variables of $|\rho| > 0.7$, we prefer the simple and more informative metric over the complex metric [101].

For the project-level metrics, we exclude the '*# of builds*' metric, since it is highly correlated with the '*# of commits per lifetime*' metric. We also exclude the '*# of build jobs*' metric, since it is highly correlated with the '*# unique build environments*' metric. We also exclude the '*# of forks*' metric, since it is highly correlated with the '*# of stars*' metric.

For the build-level metrics, we exclude (a) the '*Configuration lines added*' and

[4]https://cran.r-project.org/web/packages/rms/rms.pdf

Table 5.2: Description of the build-level metrics used in our logistic regression models

	Build-level metric	Type*	Description
Code metrics	Is pull request (PR)	C	Whether the build was triggered by a commit of a pull request
	All built commits	N	Number of commits integrated by the build (committed after the last build)
	Commits on touched files	N	Number of commits on the files changed by the commits in the build
	Source churn	N	Lines of source code changed by the commits in the build
	Test churn	N	Lines of test code changed by the build commits
	Files changed	N	Number of files changed by the commits in the build
	Source files changed	N	Number of source files changed by the commits in the build
	Documentation files changed	N	Number of documentation files changed by the commits in the build
	Configuration files changed	N	Number of configuration (e.g., `.xml` and `.yml`) files modified by the commits in the build
	Configuration lines added	N	Number of added lines to configuration files
	Configuration lines deleted	N	Number of deleted lines from configuration files
	Other files changed	N	Number of other files changed by the commits in the build
CI metrics	Build day/night	C	Whether the build is triggered during the working hours or at night (CI Server time-zone)
	Commit day/night	C	Whether code changes are committed during the working hours or at night (adjusted to time zone)
	Weekday/weekend	C	Whether the build is triggered during the week working days or on the weekend
	Is `cache` enabled	C	Whether the caching configuration is enabled or used in the build
	Is `fast_finish` enabled	C	Whether `fast_finish` configuration is enabled so that a build does not wait for *allow_failures* jobs
	Is `docker` used	C	Whether a `docker` container is used to run build jobs on
	Is `sudo` enabled	C	Whether the `sudo` configuration is enabled in the build
	Operating system (OS)	C	The operating system(s) used to run the build jobs on
	Jobs added	N	Number of jobs that are added to the build
	Jobs removed	N	Number of jobs that are removed from the build
	Jobs changed	N	Number of job changes (adding and removing) in the build
	Retry times	N	Number of times to rerun failed commands in the build
	Travis wait	N	Time (in seconds) used by the `travis_wait` build configuration to wait for long-running commands
	Install instructions	N	Number of *installation* instructions in `.travis.yml`
	Script instructions	N	Number of *script* instructions in `.travis.yml`
	After script instructions	N	Number of instructions in `.travis.yml` to run after the script is run
	Deployment instructions	N	Number of *deployment* instructions in `.travis.yml`
Developer metrics	Is core developer	C	Whether the build-triggering commit was authored by a core member of the development team
	Developer experience: # of days	N	Number of days the developer has been committing to the repository that encloses the build
	Developer experience: # of commits	N	Number of commits the developer has in the repository that encloses the build
	Developer total contributions	N	Number of contributions of developers to in the past three months
	Developer % of commits	N	Ratio of the commits of the developer to the total contributions
	Developer % of issues	N	Ratio of the issues raised by the developer to the total contributions
	Developer % of open pull requests	N	Ratio of open pull requests of the developer to the total contributions
	Developer % of merged pull requests	N	Ratio of merged pull requests of the developer to the total contributions
	Developer % of unmerged pull requests	N	Ratio of unmerged pull requests of the developer to the total contributions
	Developer % of reviews	N	Ratio of reviews performed by the developer to the total contributions

* **Type:** (C) **C**ategorical – (N) **N**umeric

'*Configuration lines deleted*' metrics, since they are highly correlated with the '*Configuration files changed*' metric, (b) the '*Source files changed*' metric, since it is highly correlated with the '*Source churn*' metric, (c) the '*Jobs removed*' metric, since it is highly correlated with the '*Jobs added*' metric, (d) and finally the '*Author experience: # of commits*' metric, since it is highly correlated with the '*Author experience: # of* *days*' metric.

Finally, we perform a redundancy analysis on the remaining variables, since redundant variables can distort regression models [47]. We use the `redun` function from the `rms` *R* package to identify the variables that can be estimated by other variables with $R^2 \geq 0.9$. We observe that none of the project-level or build-level metrics are redundant.

Developer Survey. Alongside our experimental results, we further we seek feedback and insights from software developers on the possible interplay between build durations and build breakages. In particular, we conduct a user study with developers who have contributed to the projects in our study to involve developers in confirming/validating our empirical findings. To do this, we send surveys to developers who have public E-mail addresses and have recent contributions to GitHub projects. Our questionnaire survey consists of a combination of Likert scale and open ended questions. We send $4,366$ email invitations for developers to participate in our questionnaire survey (our invitation letter and survey questionnaire are available in Appendix A). To compensate developers for their time, we offer 10$ Amazon gift cards to 30% randomly drawn survey respondents who complete the questionnaire and shared their contact information to receive the prize. Our invitation E-mail do not reach 366 developers due to having obsolete E-mails. In total, we receive 224 responses to our survey (i.e., 5.6% response rate after excluding the 366 unreachable E-mails). However, we found that 84 respondents provided partial responses in which not all the survey questions are answered, leaving us with 139 complete/submitted responses (i.e., 3.5% response rate).

We integrate the results of our developer survey with the empirical observations of each of our studied RQs. For each observation, we ask developers two questions: a Likert-scale question and open ended question. The Likert scale question asks re asks developers about the extent to which they agree or disagree with an observation from 1 (strongly disagree) to score 5 (strongly agree). The open ended question asks developers about their justification of the empirical result. To analyze the overall level of agreement of developers to each observation, we calculate the percentage of each score of each question. For open ended questions, we manually analyze the survey responses and summarize the common themes of justifications. We provide quotations of representative responses.

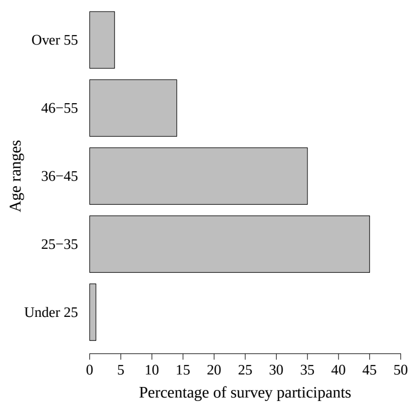
Figure 5.3 shows a summary of the demographics of our survey respondents. The majority of the respondents are males (i.e., 96%), act as maintainers or active contributors of the projects (i.e., 64%), and have over ten years of software development experience (i.e., 81%) and over five years of Travis CI experience (i.e., 54%). Therefore, all conclusions of this study are only representative of the study population.

5.3 Experimental Results

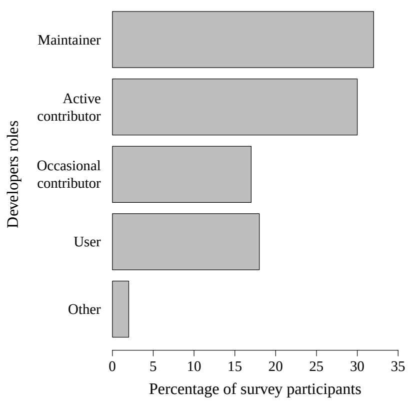
In this section, we discuss the motivation, approaches, and findings of our research questions. Beside reporting our findings, we discuss the side effects of the metrics reported by prior studies.

5.3.1 RQ$_{5.1}$: What project characteristics are associated with build performance?

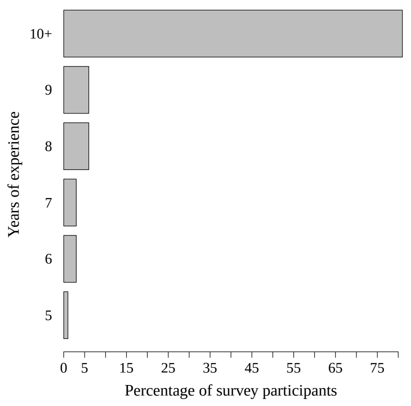
Motivation. Ståhl and Bosch [95] suggests that the variations in the observations reported by prior studies on build durations and breakages could be due to contextual



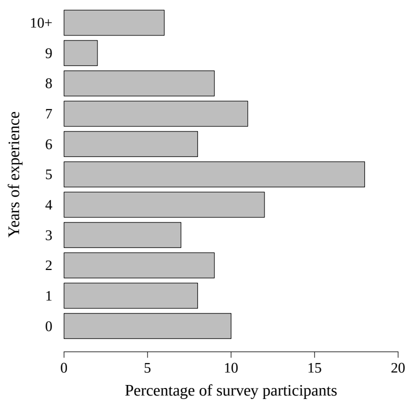
(a) Age



(b) Role



(c) Development experience



(d) Travis CI experience

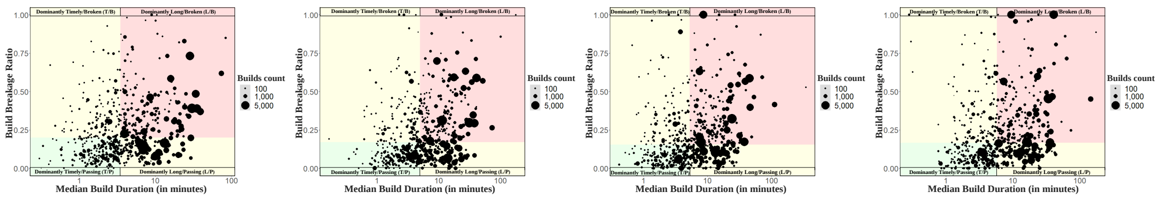
Figure 5.3: Demographics of survey respondents

differences in the studied projects. However, the role of project characteristics in affecting build performance remains unknown. In particular, it is unclear which project characteristics may have associations with build durations and breakages. It is important for developers to understand the characteristics of their projects to identify the best practices that suit their needs. In this RQ, we uncover the association of project characteristics with build durations and breakages.

Approach. We use logistic regression to model the differences in build durations and breakages between the four quadrants (shown in Figure 5.1). In particular, we fit a multinomial logistic regression model [69] using the `multinom` function provided by the `nnet`[5] R package. Our model maintains a categorical dependent variable representing the quadrants as four levels. We use the *dominantly timely/passed* quadrant as a reference level for modeling the other three quadrants. We use 14 project-level metrics as independent variables in our model. We use a stepwise algorithm that performs both forward and backward elimination of independent variables that have less contribution in modeling the difference between quadrants. Then, we use the ANOVA test [79] to compute the significance (in terms of χ^2) of each independent variable in the model. χ^2 tests if our model is statistically different from the same model in the absence of a given independent variable—according to the degrees of freedom in our model. We compute the percentage of χ^2 of each variable to the total χ^2 values of all the variables. We compute the odds ratios [7] (by exponentiating the estimated coefficients obtained from our model) to measure how a unit increases in an independent variable is associated with the dependent variable. We use *upward* and *downward* arrows to indicate direct and inverse relationships, respectively. Finally, we use the feedback from survey respondents to verify and justify our empirical findings.

Findings. ***The majority (80%) of projects retain their build performance for over three quarters of their lifetime.*** Figure 5.4 shows the results of grouping projects into quadrants for each quarter of their development lifetime. We find that the overall median build duration has steadily increased over time from 3.4 minutes (1st quarter) to 5.8 minutes (4th quarter), with an average difference of +0.8 minutes per quarter. However, we find that the build breakage ratio has decreased

[5]https://cran.r-project.org/web/packages/nnet/nnet.pdf



(a) 1st Quarter (b) 2nd Quarter (c) 3rd Quarter (d) 4th Quarter

Figure 5.4: Grouping projects into quadrants over the development lifetime (per quarter – 3 months)

Table 5.3: Results of the project-level stepwise multinomial model – Timely/Passed is the reference level

Project-level metrics	***Overall***		***Timely/Broken***		***Long/Passed***		***Long/Broken***	
	Signf.$^+$	χ^2%	Signf.	Rel.	Signf.	Rel.	Signf.	Rel.
Build environments	***	26.1			***	↗	***	↗
Building frequency	***	12.6	***	↗	***	↘	***	↘
Configuration frequency	***	11.9			***	↗	***	↗
Language (Ruby)	***	10.2	***	↗	***	↘	***	↘
Size (SLOC)	***	10.1			**	↗	*	↗
Configuration ratio	**	9.8	***	↗	***	↗	***	↗
# of commits per lifetime	**	8.6			***	↗	**	↗
Branch count	*	5.9	**	↘				
Team size	.	4.8	***	↘	***	↘	***	↘

$^+$Significance codes (*p-value*): 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

over time from 20% (1st quarter) to 17% (4th quarter), with an average difference of -1% per quarter. In addition, we observe that 28% of projects retain the same quadrant for the entire lifetime quarters and 53% of the projects alternate between only two quadrants (i.e., retaining their build performance for 75% of their lifetime). Only a very few projects (1%) alternate between the four quadrants during their lifetime quarters. Hence, in our subsequent analyses, we use the categorization of projects into quadrants resulting from the entire lifetime of projects (as shown in Figure 5.1). In Table 5.3, we show the results obtained from the multinomial regression model that we fit on the four quadrants of projects depicted in Figure 5.1.

Overall, about half of survey respondents acknowledge the association of project characteristics with build performance. As Figure 5.5 depicts, 19%

and 29% of survey respondents strongly agree and agree, respectively, that there is a potential association of the characteristics of a project with the build durations and breakages of that project. The common factors that respondents indicated to be associated with build performance include project age, size, dependencies, build complexity, and tests. Those factors are all related to project maturity, which we include them as factors in our regression model. However, we observe that only a few respondents referred to CI or developer factors, such as the number of jobs (six responses), experience of the development team (five responses), or lack of using certain build configurations such as caching (seven responses). This indicates that developers are missing CI-specific optimization opportunities for their builds.

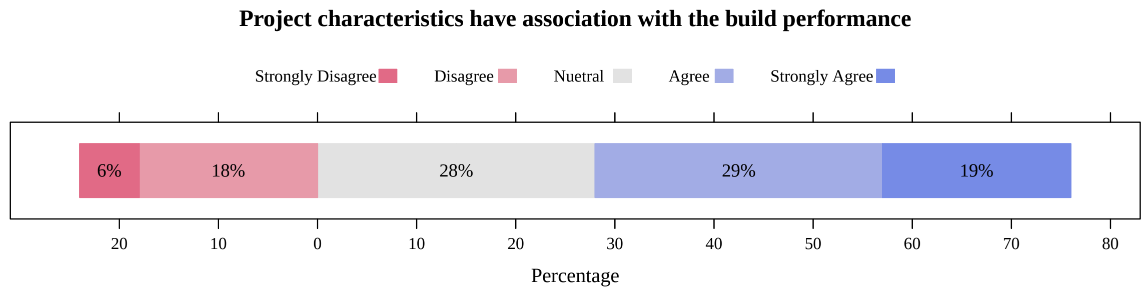


Figure 5.5: Developers' feedback about the association of project characteristics with build performance

Overly configuring CI builds is highly associated with longer build durations and frequent build breakages. The results of our model show that projects in which builds pass timely tend to configure CI builds less frequently (36% less) than other projects. As per Figure 5.6, 47% of survey respondents agree with our observation, with 28% of them neither agree or disagree. However, we found two points of views regarding such an association. On the one hand, 26% of responses indicate that changing the build configuration is likely to be the cause of long or broken builds; "*Any time configuration for a build system is changed it's likely going to cause some issues*"; "*If you're frequently modifying the build configuration then the likelihood of an error causing a build failure definitely goes up*". On the other hand, 36% of responses indicate the contrary, i.e., changes to the build configuration occurs as a reaction to optimize the build duration or fix build breakages; "*This is backwards. Builds don't fail because of modifying the build config, rather projects modify the build config because of failures*"; "*yes, this is often also our way to try to fix oddity*". Other responses (15%) indicate that the association can be in both directions depending on the type of change to build configuration; "*it can be associated with faster build when you optimize, but it can lead to some failures*". In general, respondents recommend to change the build configuration only on demand (i.e., when problems are encountered); "*Never change a winning horse. But if there are issues it makes sense to fix them*"; "*If it ain't broke, don't fix it*".

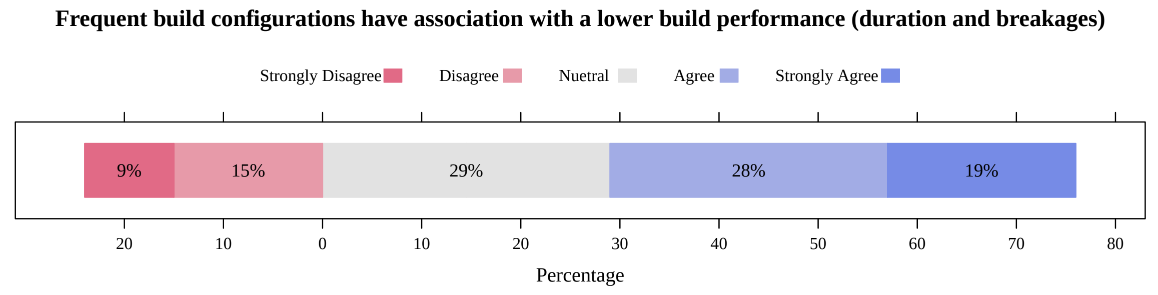


Figure 5.6: Developers' feedback about the association of frequent build configurations with build performance

The more build environments a project has, the more likely for the project to suffer from long and broken builds. We observe that projects in which builds prolong or break frequently have significantly more (a median of 1.7*x* more) build environments than timely and passed builds. Build environments can be difficult to maintain and, thus, make CI builds prolong or break intermittently [43, 44]. Previous research reported that most of the changes to the CI build configuration are related to integration environments [49]; "*Having a large build matrix (e.g., trying to support multiple combinations of Ruby versions and Rails versions)..., or having lots of active development going on in parallel (e.g., having a hackfest with dozens of CI jobs running at once and getting queued because of limited CI capacity)*". Our results indicate that, even if a project maintains single-job builds at a time, frequently changing the build integration environment for that job can negatively affect build performance (in terms of durations and breakages). Survey respondents indicate that build jobs are not always well-maintained due to "*Poor build parallelization, both in the separation of build jobs and in-job parallelization.*". Therefore, developers should be more careful when setting up build environments; e.g., "*Split longer jobs into smaller tasks which can run in parallel. Sometimes change certain time-consuming CI jobs to avoid running on master when they have just ran on a feature branch (which was merged in fast-forward-mode)*".

Large and active projects are likely to have longer and more frequent build breakages. Vasilescu et al. [102] reports that pushed commits lead to more build breakages than pull requests in older projects. However, our model reveals no association between the age of projects and build durations and breakages. We observe that, regardless of the age of projects, having 100 more commits per year increases the odds of having frequently long and broken builds by 34%. This result is also confirmed by survey respondents who indicated that "*The age of the project has no bearing on this whatsoever.*"; "*I'm not sure if I've seen a correlation here*". Other respondents, when asked about project age, tend to refer to project size "*Younger* *projects are usually smaller. A young project is more likely to have a less mature development process*" or team experience "*Probably correlated with experience of the developers with, in general, less experienced developers on younger projects.*". Hence, developers who commit frequently to the code base should be more careful about build breakages while taking into consideration the potential accumulated increase of build durations.

5.3.2 RQ$_{5.2}$: What build-level metrics are associated with build durations and/or breakages?

Motivation. Findings of *RQ$_{5.1}$* suggest that project characteristics can have strong associations with build durations and breakages. However, builds from the same project may have different characteristics. Therefore, in this RQ, we aim to understand which build-level metrics have a strong association with build durations and breakages.

Approach. Similar to *RQ$_{5.1}$*, we fit a multinomial logistic regression model using quadrants a categorical dependent variable, with the *dominantly timely/passed* quadrant as a reference level. We use the 35 build-level metrics that remained after performing correlation and redundancy as independent variables in our model. We also use the stepwise algorithm to identify significant independent variables, measure the χ^2 using the ANOVA test and compute the odds ratios of each variable. We mark direct and inverse relationships using *upward* and *downward* arrows, respectively. Finally, we use the feedback from survey respondents to verify and justify our empirical findings.

Findings. Table 5.4 presents the results obtained from the models of each of the

Table 5.4: Results of the build-level stepwise multinomial model

Project-level metrics	***Overall***		***Timely/Broken***		***Long/Passed***		***Long/Broken***	
	Signf.$^+$	χ^2%	Signf.	Rel.	Signf.	Rel.	Signf.	Rel.
Developer experience: # of days	***	29.7	***	↗			***	↗
Test churn	***	14.3	***	↗	***	↗	***	↗
Installation	***	14.0	***	↗	***	↗	***	↗
Script instructions	***	8.8	***	↗	***	↗	***	↗
Retry times	***	8.2	***	↗	***	↗	***	↗
Is `travis_wait` used	***	4.5	***	↘	**	↘	***	↗
Is `sudo` used	***	3.8	***	↘	***	↗	***	↗
Files changed	***	3.5	***	↗	***	↘	***	↘
OS: *OSX*	***	2.2	***	↘	***	↘	***	↘
OS: *Linux + OSX*	***	2.2	***	↗	***	↗	***	↗
OS: *Linux + Windows*	***	2.2	***	↗	***	↘	***	↘
OS: *Linux + OSX + Windows*	***	2.2	***	↗	***	↘	***	↘
After script instructions	***	1.9	***	↗			***	↗
Developer total contributions	***	1.1	***	↘	***	↘	***	↘
Configuration files changed	***	1.0	***	↗			***	↗
Is `fast_finish` enabled	***	0.3	***	↗	***	↗	***	↗
Developer % of open pull requests	***	0.2	***	↗	***	↘	***	↘

$^+$Significance codes (*p-value*): 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

four quadrants.

The experience and role of developers have significant associations with build performance. A prior study [85] reported that less experienced developers are likely to produce fewer build breakages. The results of our model also show that code changes submitted by more experienced developers increase the odds of having long builds in addition to build breakages. In particular, we observe that *dominantly long/broken* projects have significantly more experienced developers (a median 2.5 years) than *dominantly timely/passed* projects (a median 2 years). However, looking at Figure 5.7, we find that 77% of the survey respondents agree that it is the contrary; "*the opposite is true, novice contributors would make the build failing and that is a main motivation for having tests, ci, etc.*" Still, respondents believe that less experienced developers usually have timely and passed builds when submitting simple or less impactful changes (e.g., documentation) or do not participate in software testing; "*maybe they only commit simple changes that are fundamentally less likely to cause action at a distance*". Other respondents indicate that "*project* *maintainers don't care about their failing builds, but the new contributor does*" and "*experienced developers are less likely to run the tests they've added/changed locally first, so probably more likely to cause failures on CI*". Moreover, we observe that developers with more contributions (e.g., commits, pull requests, and reviews) have a strong association with timely and passed builds. This result suggests that, regardless of development lifespan, more active developers tend to be more careful with CI builds; "*I generally expect frequent core contributors to be the most effective at fixing these kinds of issues*".

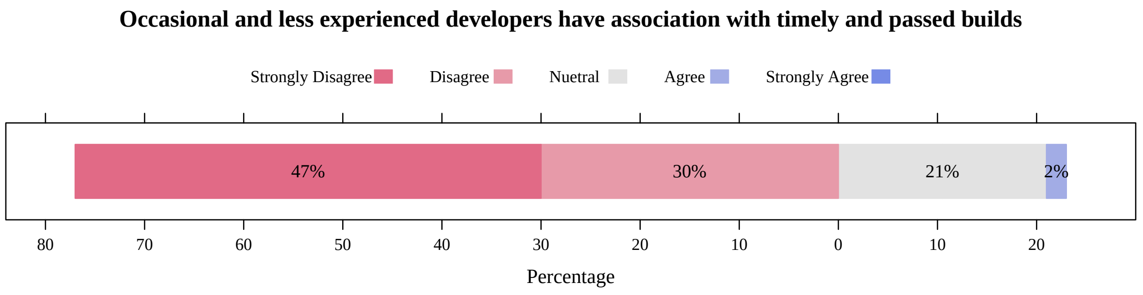


Figure 5.7: Developers' feedback about the association of the experience of developers with build performance

Waiting for long-running tests and retrying failed commands multiple times is strongly associated with long and broken builds. Prefixing build commands with `travis_wait` or `travis_retry` (or built-in `--retry` arguments) is recommended to bypass unexpected build breakages due to timeouts.[6,7] However, such configurations are likely to delay builds [43]. Besides delaying builds, we observe that command retrial does not even guarantee passing builds. Our results show that retrying or waiting for commands or tests are highly associated with long and

[6]https://docs.travis-ci.com/user/common-build-problems/#build-times-out-because-no-output-was-received

[7]https://docs.travis-ci.com/user/common-build-problems/#timeouts-installing-dependencies

frequently broken builds. To investigate the reasons, we analyze all builds in our dataset that use the `travis_wait` configuration. We observe that 90% of the builds do not specify the time to wait for long-running commands (i.e., wait for 20 minutes by default). However, analyzing the logs of the builds that use `travis_wait` shows that breakages occur after command waiting by a median of ten minutes. In addition, we observe that builds may be configured to wait for 120 minutes but eventually break without waiting that long. After investigating reported issues to Travis CI,[8] we find that, if a command generates child processes, then `travis_wait` does not track such processes and timeouts when the main process becomes silent for ten minutes. Nearly half of survey respondents agree that waiting or retrying commands are not always helpful to fix build breakages (Figure 5.8) and should be a last resort; For example, respondents indicate that "*automatic retries are a band-aid on a serious wound. If your tests aren't reliable, fix your tests*"; "*Long tests or flaky builds should be fixed instead of retried*". Considering that command retrial "*is a strategy to address something that's unreliable, so it makes sense that it would often not be successful*". Nevertheless, respondents recommend to use such configurations only in specific situations when failures are due to external factors; "*the 3rd party API happened to suck and be unreliable, so retrying the submission to that API was a necessary evil*". Our analysis of the responses of respondents who disagree with our observation shows that their arguments also confirm that developers should be more careful when using automatic retrials of failing commands. For example, "*If you're retrying and it doesn't fix the build, you should fix your retry logic or stop retrying*"; "*Well I think retrying does help but it's a bad idea for the future*". Hence, developers are encouraged to avoid generalizing automatic retrials for all commands and should

[8]https://github.com/travis-ci/travis-ci/issues/8526

rather consider fixing such commands.

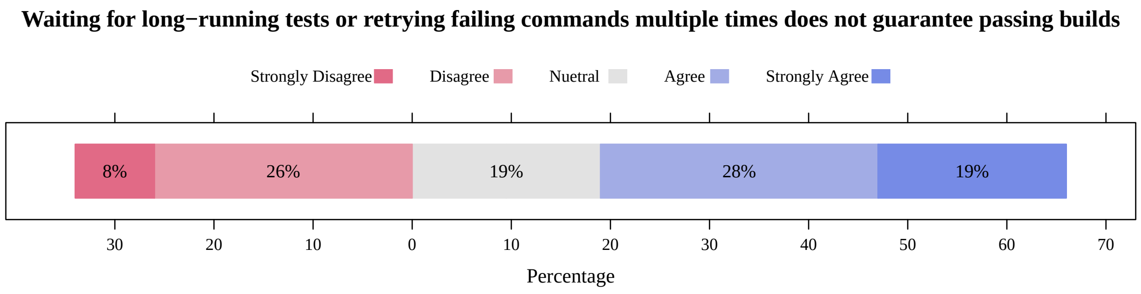


Figure 5.8: Developers' feedback about the association of retrials and waiting for commands/tests with build performance

Using multiple or non-default build architectures (i.e., operating system) has a negative association with build performance. Failure to choose a proper build architecture may impact the overall CI building experience [117]. We observe that running builds on multiple operating systems or changing the default operating system (i.e., Linux) has a strong association with long and frequently broken builds. In particular, we observe that projects that use `OSX` (alone or with other operating systems) have significantly longer and more frequent build breakages than other projects. We investigate the reasons behind such a negative behavior of `OSX`. We find that the documentation[9] of TRAVIS CI indicates that running builds on newer versions of `OSX` is likely to generate unexpected build breakages. In addition, we observe that 95% of the builds that use `OSX` also use `Linux` and/or `Windows`. Yet, coupling `OSX` with other operating systems may introduce conflict in some build commands (e.g., file permission commands). Nearly a third of respondents agree with our observation (Figure 5.9). Yet, 47% of respondents have disagreement with , the majority of disagreeing responses indicate that respondents do not find this association

[9]https://docs.travis-ci.com/user/common-build-problems/#mac-macos-sierra-1012-code-signing-errors

to be obvious with no prior experience of using multiple build architectures. Still, based on the analyzed responses, some operating systems to be more likely to break or delay builds; "*I would expect older and less-popular OSes to have more failures and be slower, because they might indicate a very big matrix (e.g., supporting many OSes) or because there are more dependency failures as an OS drops out of support*". We also find that respondents provide clear reasons why operating systems other than Linux can make builds take longer "*Windows has proven to be a lot more difficult to reduce build times...I believe there are less macOS builders available due to the lack of containers which can result in builds taking longer to start*" or break unexpectedly "*I would say that having reproducible builds for `OSX` and `iOS` is not always as easy, since it requires a level of automation*". Therefore, we suggest that developers should select build architectures carefully to avoid unexpected build performance.

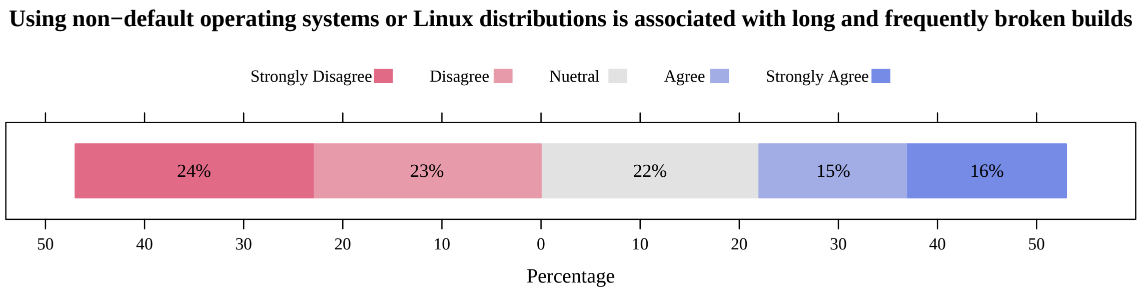


Figure 5.9: Developers' feedback about the association of build architecture with build performance

5.3.3 RQ$_{5.3}$: How should developers act to optimize build performance?

Motivation. *RQ$_{5.2}$* presented the most important build-level metrics that are associated with each quadrant. Yet, it is important for developers to understand the actions that can help projects switch from a certain quadrant (e.g., an undesirable state) to another quadrant (e.g., a desirable state). In this RQ, we study the most important metrics associated with switching builds of a project from one quadrant to another quadrant.

Approach. To understand the switch between quadrants, we use logistic regression to model the difference between each pair of switching quadrants. Of the 12 possible quadrant pairs, we only model six unidirectional switches between quadrants, since modeling the opposite direction would produce the same important metrics but with an opposite effect. In other words, metrics that are positively associated with the switch from quadrant A to quadrant B are negatively associated with the switch from quadrant B to quadrant A. For each project in each quadrant, we keep the builds in which the build status and build duration agree with the label of the quadrant. For example, for the bottom-left (*Dominantly Timely/Passing*) quadrant, we only keep *passed* builds whose durations are under the overall median build duration (i.e., five minutes).

We use mixed-effect logistic regression to model the switches between the four quadrants shown in Figure 5.1. $RQ_{5.1}$ suggests that project characteristics are strongly associated with build performance. Hence, we use Generalized Linear Mixed Models (GLMM) for logistic regression [36] to control the variations between projects in our models. GLMM employs mixed (i.e., fixed and random) effects when modeling the relationship between the dependent and independent variables. In our models, we control the variations between projects by using the '*Project Identifier*' as a random effect in our mixed-effect logistic models. This means that our models assume a different intercept for each project [66]. We fit six models to model the six unidirectional switches between quadrants using the build-level metrics that remained after performing correlation and redundancy as independent variables in our models. Each model maintains a categorical dependent variable that represents the labels of the two quadrants under modeling. Given that we are interested in modeling the switch to a better quadrant (e.g., switching from *Long/Broken* to *Timely/Passing*), we consider the undesirable quadrant as the reference level of the categorical dependent variable. For example, when we model the switch from the *Long/Broken* quadrant to the *Timely/Passing* quadrant, we make *Long/Broken* as a reference level. For the *Long/Passing* and *Timely/Broken* quadrants (in which the switch could be considered positive in either direction), we make *Long/Passing* as a reference level. Similar to $RQ_{5.1}$ and $RQ_{5.2}$, we use the ANOVA and odds ratios to analyze the relationship of the dependent variable with the independent variables of each model. Finally, we use the feedback from survey respondents to verify and justify our empirical findings.

Findings. Table 5.5 presents the results obtained from the six models to switch projects to better build states.

Actions to improve build performance are restricted by the current build performance of a project. We observe that actions associated with the quadrant switching of a project may vary depending on the current quadrant of that project, i.e., "one size does not fit all". For example, actions associated with switching dominantly long/broken projects (e.g., *committing on weekends* or *using fewer command retrials*) are not associated with switching dominantly timely/broken projects. On the other hand, actions associated with switching dominantly timely/broken projects (e.g., `fast_finish`) are not associated with switching dominantly long/broken projects. While 43% of survey respondents agree and 44% neither agree or disagree with this conclusion, as Figure 5.10 depicts, analyzing the response of respondents

Table 5.5: Results obtained from the six switching mixed-effects logistic models

Metric	**L/B $\mapsto$ T/P**			**L/B $\mapsto$ L/P**			**L/B $\mapsto$ T/B**			**L/P $\mapsto$ T/P**			**L/P $\mapsto$ T/B**			**T/B $\mapsto$ T/P**		
	χ^2%	Signf.+	Rel.	χ^2%	Signf.	Rel.	χ^2%	Signf.	Rel.	χ^2%	Signf.	Rel.	χ^2%	Signf.	Rel.	χ^2%	Signf.	Rel.
After script instructions	0.35		↘	0.29		↘	0.01		↘	2.1	**	↘	0		↗	1.1	**	↘
Developer experience: # of days	0.09		↘	0.23		↘	0.08		↗	0.02		↗	0.95		↘	3.14	***	↘
Build day/night	0.53		↗	1.9	*	↘	0.02		↗	0.13		↗	0.14		↗	0.1		↗
Is CI `cache` enabled	**37.7**	***	↗	**22.84**	***	↗	**14.84**	***	↗	**19.94**	***	↗	**8.21**	***	↗	**25.07**	***	↗
Commit day/night	0		↗	0.52		↘	0.42		↗	2.51	**	↗	4.34	**	↗	0.18		↘
Developer % of commits	1.03	*	↗	0.56		↗	1.83	*	↗	**3.64**	***	↗	**7.49**	***	↗	0.92	**	↘
Configuration files changed	0.03		↘	2.63	**	↘	0.11		↗	0.33		↗	**4.53**	**	↗	0.24		↘
Deployment instructions	0.88	*	↗	0.29		↗	1.19	.	↗	**3.77**	***	↗	2.47	*	↗	0.55	*	↗
OS distribution	0.77	.	↗	0.69		↗	**4.13**	***	↗	0.6		↗	2.28	*	↗	2	***	↘
Is `docker` used	0.02		↗	1.99	*	↗	0.13		↗	1.47	*	↘	1.45		↘	0.22		↘
Is `fast_finish` enabled	**14.98**	***	↗	**15.51**	***	↗	**22.41**	***	↗	0.02		↘	**6.05**	***	↗	**5.53**	***	↘
Is core developer	0.96	*	↘	1.13	.	↗	0.03		↘	2.65	***	↘	0.77		↘	0.07		↘
Documentation files changed	0.12		↘	0.66		↗	0.35		↗	0.37		↘	0.02		↗	0.22		↗
Files changed	0		↗	0.81		↗	0.74		↗	0.13		↗	0.12		↗	0.23		↘
Other files changed	0.08		↗	0.14		↘	0.22		↘	0.19		↗	0.07		↗	0		↘
Is pull request (PR)	**5.1**	***	↘	2.75	**	↘	0.84		↗	0.04		↘	**24.68**	***	↗	**9.59**	***	↘
Commits in push	0.43		↘	0.12		↘	0.21		↗	0.15		↘	3.59	*	↗	**5.63**	***	↘
Commits on touched files	0.08		↗	0.46		↗	0.3		↗	0.56		↘	1.75	.	↘	0.59	*	↘
Source churn	0.03		↗	0.61		↘	0.02		↘	0.55		↗	0.54		↗	0.4	.	↘
Test churn	0.11		↘	1.39	.	↘	1		↘	0.37		↘	0.24		↘	0.11		↘
Installation instructions	**3.65**	***	↘	0.55		↗	1.62	*	↗	**45.5**	***	↘	**9.99**	***	↘	1.81	***	↘
Developer % of issues	0.74	.	↗	0.06		↗	0.01		↘	0.06		↗	0.11		↗	0.03		↘
Jobs changed	0.13		↘	0.3		↘	0.06		↘	0.01		↘	0.18		↘	0		↘
Developer % of merged pull requests	**3.96**	***	↗	1.71	*	↗	0.21		↗	0.02		↘	**7.73**	***	↘	2.17	***	↗
Jobs added	1.34	*	↘	2.14	*	↘	0.06		↘	0.63		↘	0.96		↗	1.42	***	↘
Developer % of open pull requests	1.13	*	↘	1.83	*	↘	0.22		↘	0.74	.	↗	0.43		↗	0		↗
Operating system (OS)	0.26		↘	0.07		-	0.02		-	0.08		-	0.97		-	**19.6**	***	-
Retry times	**10.82**	***	-	0		↗	**13.14**	***	↘	0.15		↘	2.3	*	↘	0.55	*	↗
Developer % of reviews	0.01		↗	0		↘	**4.75**	***	↗	0.05		↘	0.23		↗	0.72	*	↘
Script instructions	1.16	*	↘	3.31	**	↘	1.12	.	↘	**9.29**	***	↘	0.74		↘	1.66	***	↘
Is `sudo` enabled	0.04		↗	1.37	.	↘	2.48	**	↗	0.16		↗	2.51	*	↗	3.57	***	↗
Travis wait	0		↗	0		↗	0		↗	2.02	**	↘	3.11	*	↘	**12.53**	***	↘
Developer % of unmerged pull requests	0		↗	0.02		↘	0.16		↘	1.58	**	↗	0.99		↗	0.01		↘
Build weekday/weekend	**13.44**	***	↗	**33.13**	***	↗	**27.29**	***	↗	0.16		↗	0.05		↘	0.04		↘

+Significance codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

reveals that such restriction in options is due to the trade-off between build durations and breakages. For example, "*Slowness and unreliability compound each other: if a build is slow, it's hard to debug because it takes so long to get results; if a build is unreliable, it's hard to debug because the change you made may not be the reason it was failing.*". Therefore, developers are encouraged to monitor the performance of their builds during the past periods before acting towards optimizing it.

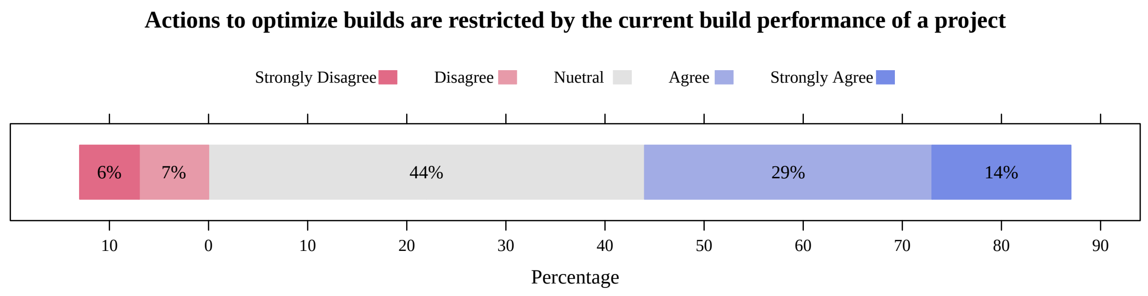


Figure 5.10: Developers' feedback about the restricted actions to improve build performance

Enabling the `fast_finish` build configuration is associated with timely builds only when builds are dominantly broken. Prior research reported that configuring CI builds to finish as soon as the required jobs finish is associated with timely builds [43]. However, our results indicate that such a configuration has lower chances of reducing build durations when builds are dominantly passing. The `fast_finish` configuration is always aligned with the *allow_failures* configuration, which developers use to allow some builds jobs to fail without breaking the entire build. Hence, having both `fast_finish` and *allow_failures* helps builds finish faster as builds are not required to wait for *allow_failures* jobs to finish, since they do not affect the overall build status. Figure 5.11 shows that 59% of survey respondents neither agree or disagree with our observation, while 27% of them agree with our observation. Yet, respondents do not recommend the use of *allow_failures* jobs to allow revealing all possible test failures; "*allow_failures is generally a bad idea. Tests should be fixed, or deleted, not ignored*"; "*perhaps developers don't look at the CI that much when it usually succeeds anyway*". Therefore, developers should carefully define the rationale of either allowing a build to fail or just exclude it from the build.

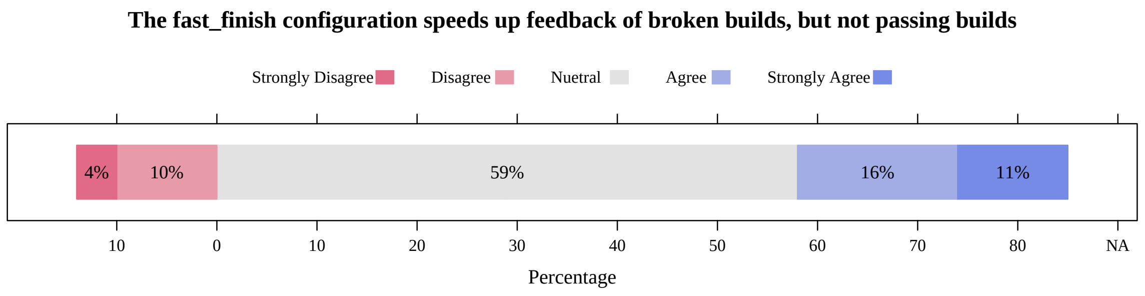


Figure 5.11: Developers' feedback about the association of `fast_finish` with only timely broken builds

CI caching has an inverse association with build breakages in addition to build durations. Prior research reports that caching content that change less often is highly associated with timely builds [43]. Our results reveal that, besides speeding up builds, the odds of having passed builds increases by 139% when caching is adopted. Two-thirds of the projects that adopt CI caching cache dependencies to the CI server to allow future builds to fetch the binaries of the dependencies from the cache and only recently updated dependencies are installed from their sources. Such a process can help builds avoid failures that might occur due to the dependency installation [85]. We observe from Figure 5.12 that 40% of respondents agree with observation; "*This is true. I've often reduced failures by caching more artifacts locally, rather than requiring my builds to go out to the internet and fetch all the tools we need each time.*". However, there is also a relatively higher percentage (43%) of disagreement. Disagreeing responses indicate that respondents do not find this association to be obvious. Yet, our analysis of disagreeing responses shows arguments that indicate a negative side effect of CI caching (i.e., causing occasional build breakages); "*In my experience it is opposite. High level of caching often can cause issues where stale data is used accidentally, causing build failures.*". Hence, developers should better asses their needs for caching in their builds and keep any eye on build breakages that might be caused due to caching.

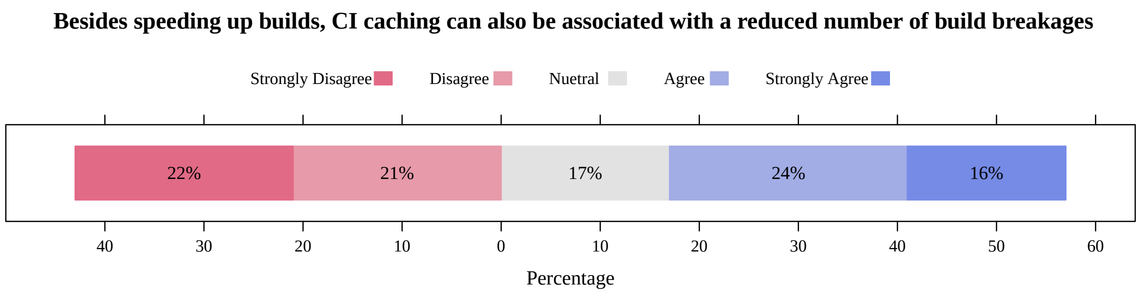


Figure 5.12: Developers' feedback about the association of `caching` with only timely broken builds

The day of week has a stronger association with long and broken builds than any other builds. According to prior studies, the day of week is associated with build durations [43] but not with build breakages [85]. Our models show that the day of week is associated with both build durations and breakages. In particular, we observe that committing on weekdays is 95% more likely to generate prolonged and broken builds. Such an issue has occasionally been reported to the TRAVIS CI team.[10,11]. We find that the TRAVIS CI team is aware of the overhead caused due to triggering builds during rush hours. Although 27% of respondents agreed with out observation, other 30% of respondents have an uncertain opinion. Overall, respondents commonly agree that, on weekends, CI resources are more likely to encounter less load in comparison with working days. Respondents also confirmed that shared CI resources are vital in such a case; " *If shared compute hardware is under reduced load and the tests are running faster, time-dependent (race condition) type non-deterministic test failures may also be reduced*". However, respondents indicated

[10]https://github.com/travis-ci/travis-ci/issues/2072
[11]https://github.com/travis-ci/travis-ci/issues/8489

that this might not always be the case; e.g., "*as a paid travis subscriber, we don't get throttled.*". Hence, developers are recommended to reduce the number of operations in their builds to make their builds both timely and passed regardless of the triggering time.

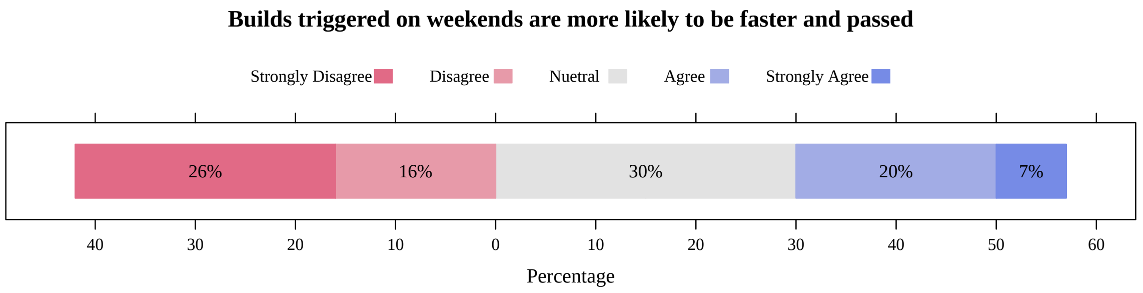


Figure 5.13: Developers' feedback about the association of day of week with build performance

5.3.4 RQ$_{5.4}$: How frequently does build performance change over time?

Motivation. Findings of $RQ_{5.3}$ suggest that changing development and CI practices are associated with build performance. However, it is still unclear what makes projects undergo major ups and downs to build performance during the development lifetime. It is important to understand which practices change when the build performance changes over time. In this RQ, we investigate the patterns, frequency, and metrics of projects switching between different build performance states over time.

Approach. We analyze the evolution of build performance of the studied projects. For each project, we analyze the state-switching patterns using two scenarios, as follows:

- We identify the state switching between every pair of subsequent builds.
- We divide the project lifetime into four quarters using the 1st quantile, median,

and 3^{rd} quantile. For each quarter, we identify the most frequent state of each quarter. Then, we identify the state switching between the quarters of each project.

We use the rules and labels presented in Table 5.6 to tag the state switches between every two subsequent builds and quarters. If two subsequent builds or quarters have the same state label (e.g., both have a *positive* label), we combine their labels into a single label. For example, assume a project with four quarters (S_1, S_2, S_3, and S_4). If the switch from S_1 to S_2 is positive and from S_2 to S_3 is positive, then the switch from S_1 to S_3 is positive. As a result, we identify eight evolutionary trends (16 sub-patterns) for projects to switch from one state to another state or remain in the same state throughout the development lifetime.

We explore the changes in the values of the metrics between every two quarter-to-quarter state switches. To do this, we use nonparametric statistical tests instead of regression modeling, since the small number of builds in each state could lead to overfitting problems [78]. Following the guidelines provided by Sheskin [90] (previously used in a CI context [85]), we perform Pearson's χ^2-tests for categorical metrics and Wilcoxon rank sum (Mann–Whitney U) tests for numeric metrics. We use the standard significance testing of null hypotheses for each metric using the common level of significance $\alpha = 0.05$. For each significant metric (i.e., p-value < 0.05), we compute the effect size using Cramér's V for categorical metrics and rank-biserial correlation r for numeric metrics. The higher the effect size, the more important the metric is for modeling state-switching. Finally, we use the feedback from survey respondents to verify and justify our empirical findings.

Findings. Figure 5.14 shows (a) the ratios of build-to-build state switches and (b)

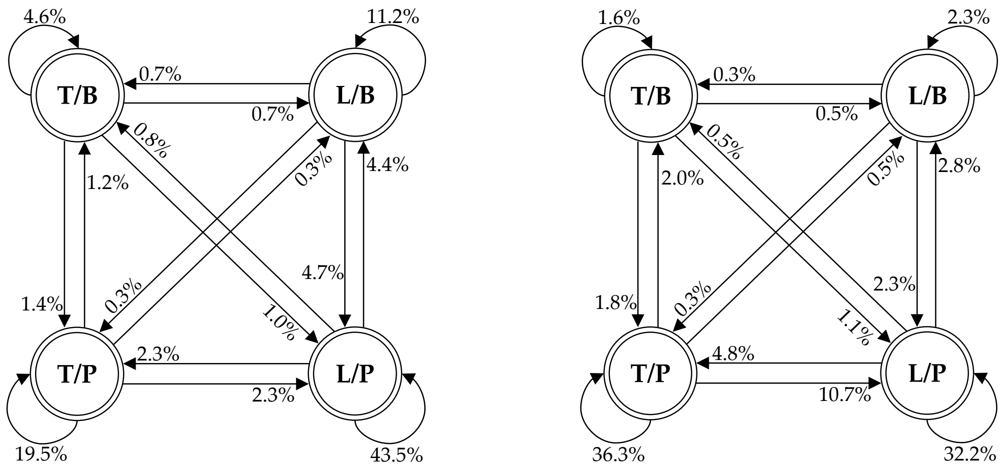
Table 5.6: The rules used to tag the quadrant switching between subsequent builds/quarters of each project

Previous quadrant	Current quadrant	Switch
Any	Timely/Passing	$\oplus$
Any	Long/Broken	$\ominus$
Timely/Passing	Any	$\ominus$
Timely/Broken	Long/Passing	$\ominus$
Long/Passing	Timely/Broken	$\ominus$

the ratios of quarter-to-quarter state switches across the studied projects. Table 5.7 presents eight patterns and 16 sub-patterns in which projects undergo build state-switching over time.

The majority (i.e., 79%) of builds in our dataset perform similarly to former builds. Prior research has reported that build breakages follow former breakages [44, 85]. We observe that not only the breakage but also the duration of CI builds follow former builds. Nevertheless, there 21% of build pairs that encounter different performance. We observe that performance state switching tends to have a similar proportion in both directions. For example, we observe that the proportion of builds that switch from the *Timely/Passed* state to the *Long/Passed* state is analogous (i.w., 2.3%) in both directions. Therefore, it is important for developers to understand how to revert CI builds from an undesired build state to a better state.

Two-thirds of the studied projects have a persistent build performance over time. Prior research has paid little attention to the evolution of both build durations and breakages. Looking at Table 5.7, we observe that approximately 55% of the studied projects maintain a persistent build performance throughout the development lifetime. In particular, most of the projects (i.e., 29.9%) were maintaining undesirable build performance, whereas build states of 25.2% of the projects have



(a) Build to build (b) Quarter to Quarter

Figure 5.14: Percentages of state-switching cases

been steadily positive over time. Such results indicate that there is still room for improvement in the build performance of such projects. Nevertheless, build performance has undergone degradation and improvement in 24.3% and 7.1% of the projects, respectively. Both performance degradation and improvement mostly (i.e., 44% and 43%, respectively) occur at early stages (i.e., after the first quarter) of project lifetime. This result suggests that build performance can become worse easily, which poses a challenge for developers to return it back to its normal state. Moreover, we observe around 13% of the projects in which builds undergo multiple state switches over time (e.g., $\ominus \mapsto \oplus \mapsto \ominus$ or $\oplus \mapsto \ominus \mapsto \oplus \mapsto \ominus$).

State-switching of build performance is accompanied by significant changes to development activities. Overall, there are 264 projects (45% of all projects in our dataset) in which build performance has been unsteady throughout the development lifetime. We identify a total of 619 quarter-to-quarter state switches in those projects. Our statistical analyses of those state switches show that there are significant differences between the characteristics of builds before and after a state

Table 5.7: Patterns of quadrant state-switching over time

Pattern	**Trend**	**Sub-Patterns**	**Projects** #	%
$\ominus$	Negative	$\ominus \mapsto \ominus \mapsto \ominus \mapsto \ominus$	176	29.9%
$\oplus$	Positive	$\oplus \mapsto \oplus \mapsto \oplus \mapsto \oplus$	148	25.2%
$\oplus \mapsto \ominus$	Degrading	$\oplus \mapsto \ominus \mapsto \ominus \mapsto \ominus$	63	10.7%
		$\oplus \mapsto \oplus \mapsto \ominus \mapsto \ominus$	43	7.3%
		$\oplus \mapsto \oplus \mapsto \oplus \mapsto \ominus$	37	6.3%
$\ominus \mapsto \oplus$	Improving	$\ominus \mapsto \oplus \mapsto \oplus \mapsto \oplus$	18	3.1%
		$\ominus \mapsto \ominus \mapsto \ominus \mapsto \oplus$	17	2.9%
		$\ominus \mapsto \ominus \mapsto \oplus \mapsto \oplus$	7	1.2%
$\oplus \mapsto \ominus \mapsto \oplus$	Recovering	$\oplus \mapsto \ominus \mapsto \oplus \mapsto \oplus$	17	2.9%
		$\oplus \mapsto \oplus \mapsto \ominus \mapsto \oplus$	12	2.0%
		$\oplus \mapsto \ominus \mapsto \ominus \mapsto \oplus$	8	1.4%
$\ominus \mapsto \oplus \mapsto \ominus$	Relapsing	$\ominus \mapsto \oplus \mapsto \ominus \mapsto \ominus$	13	2.2%
		$\ominus \mapsto \ominus \mapsto \oplus \mapsto \ominus$	10	1.7%
		$\ominus \mapsto \oplus \mapsto \oplus \mapsto \ominus$	7	1.2%
$\oplus \mapsto \ominus \mapsto \oplus \mapsto \ominus$	Fluctuating	$\oplus \mapsto \ominus \mapsto \oplus \mapsto \ominus$	10	1.7%
$\ominus \mapsto \oplus \mapsto \ominus \mapsto \oplus$	Fluctuating	$\ominus \mapsto \oplus \mapsto \ominus \mapsto \oplus$	2	0.3%

switch (the full list of statistical testing results of state-switching can be found in our online appendix [3]). We observe that the experience of developers (within-project or cross-project) is significantly associated with all state switches. In addition, we observe that the most significant metrics (e.g., the top 20 metrics) of in-project state switches are different from those cross-project switches reported in RQ_3. For example, 21% and 31% of the analyzed state switches are accompanied by changes in CI *fast_finish* and *caching* configurations, respectively. Such results indicate that developers should make use of CI practices (e.g., collaborating with experienced developers from other projects) to improve the build performance in their projects.

5.4 Implications

In this section, we discuss the implications of our findings for developers, researchers, and CI services.

5.4.1 Implications for developers

Reducing both build breakages and durations. Developers rely on the documentation provided by the CI service provider when configuring builds. However, developers may be unaware of the consequences of such configurations on their projects. For example, dependency caching is known to speed up builds. However, developers might not realize that caching could reduce build breakages, since cached binaries of dependencies can be used instead of installing dependencies from sources in subsequent builds. Such a process helps builds avoid many unnecessary installations that may lead to unexpected build breakages.

Assessing where a project stands before making any build optimization decisions. Actions to improve build performance may not fit every project. The characteristics and current build performance of a project can limit the actions to reduce build durations and breakages. Hence, developers should take into consideration such factors when configuring their builds. For example, when a project does not maintain experimental build environments (i.e., build jobs), it is unlikely for that project to benefit from the *fast_finish* build configuration.

Ensuring that fixes to build breakages really resolve the errors causing the breakage. Developers need to be careful when dealing with build breakages. Our findings show that certain practices to fix build breakages may just increase the build duration without reducing build breakages. For example, when a dependency installation frequently fails, developers may configure the build to allow retrying the installation command multiple times. However, our results show that such retrials are unlikely to resolve the installation error but highly likely to increase the build durations. Therefore, developers are encouraged to investigate the reasons behind the frequent failure of a command rather than just rerunning it multiple times.

Constant monitoring of build performance. Build performance may change (i.e., improve or decline) at any time. Hence, developers should frequently monitor build durations and breakages to identify any anomalous increase. Our results indicate that build performance measures most likely decline at early stages and improve at later stages of the development lifetime. Therefore, continuous inspection of recent build configurations is highly encouraged to keep build states persistently positive.

5.4.2 Implications for researchers

Awareness of the importance of project characteristics when reported research findings. Research has been conducted to study build durations and breakages. Yet, only a few studies have considered which builds belong to which projects (e.g., using project-wise statistical testing [85] or mixed-effects regression [43, 44]). Other studies have analyzed builds from different projects together. We encourage researchers to take into account that different project characteristics can impact empirical conclusions. Doing so can help developers identify which practices would fit their projects.

Highlighting side effects when reporting actionable findings. Researchers should take into consideration any possible side effects and trade-offs when reporting actionable findings to developers. When a reported action is leveraged, it is hard for developers to know whether such an action would lead to unexpected results. Exploring all possible aspects of a CI problem can help identify the undesirable effects of using an action to solve that problem.

Approaches for detecting/reporting the inefficient adoption of CI configurations. Our findings suggest that not all projects can benefit from certain code or CI configurations. For example, a project with fewer dependencies is unlikely to benefit from the adoption of CI caching. In addition, single-job projects are unlikely to benefit from the *fast_finish* build configuration. Therefore, developers need approaches to predict or detect whether a certain CI practice achieves its anticipated benefit, since the current build performance of a project may impede such a practice from performing well.

5.4.3 Implications for CI services

CI service providers should continuously improve the documentation of CI configurations. Despite the details/examples provided by CI services (e.g., Travis CI[12]), certain CI configurations may still lead to unexpectedly low build performance. Hence, side effects and misuses of CI configurations could be as a result of unclear, inadequate, or misunderstood CI documentation.[13] Therefore, CI services should maintain clear and updated documentation to avoid misleading developers when configuring builds.

5.5 Threats to Validity

In this section, we discuss the potential threats to the validity of our findings.

[12]https://docs.travis-ci.com
[13]https://github.com/travis-ci/travis-ci/issues/5700

Construct validity. Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. In our study, we rely on the data collected and computed mostly from TravisTorrent and from the Git repositories that we clone from GitHub. Mistakenly computed values may influence our results. However, we carefully filter and test the data to reduce the possibility of wrong computations that may impact the analyses of this study. In addition, we report our findings based on exploratory models that we fit on data collected from actual projects. Still, future studies should consider surveying developers to further understand ways to control the interplay between build durations and build breakages in practice.

Internal validity. Internal threats to validity are concerned with the ability to draw conclusions from the relation between the independent and dependent variables. In our study, we investigate the association between 18 project-level and 40 build-level metrics and four build states. In particular, we use logistic regression to model the differences between project quadrants. However, we are aware that these metrics are not fully comprehensive. Using additional metrics may affect our results. In addition, in the cases where two metrics are highly correlated, deciding which metrics to keep and which metrics to remove in our models may have an impact on the results obtained from the models. To make our results reproducible, we make our choices of selected metrics explicit for all the pairs of highly correlated metrics.

External validity. External threats are concerned with our ability to generalize our results. Our study is based on builds that are collected from 588 GitHub projects that are linked with Travis CI. To mitigate this threat, we include projects that have larger numbers of builds spanning up to ten years. Yet, we cannot generalize our conclusions as more projects with different characteristics should be investigated. For example, a replication of our work using projects written in other programming languages or linked with other CI services is required to reach more general conclusions.

5.6 Summary

In this chapter, we conduct an empirical study to investigate the interplay between mitigating build durations and build breakages. We study (1) which project characteristics are associated with build durations and breakages; (2) the most important build-level metrics that are associated with build durations and breakages; (3) the actions that help developers generate satisfactory builds; and (4) whether software projects undergo build state switches throughout the development lifetime. In addition, we conduct a questionnaire survey with software developers who have contribute to the projects in our dataset to get their feedback on our empirical observations. We summarize the key findings of our study as follows:

- Project characteristics (e.g., project maturity, age, and build configuration ratio) have a significant association with build performance.
- Metrics that are known to be associated with one build measure (e.g., the build breakage) may have positive or negative associations with the other build measure (e.g., the build duration).
- Actions to improve the builds of a project may depend on the current build state of that project.

- Developers should keep an eye on their builds as a build state may change as a result of changes to the development or building practices.

Our findings encourage researchers and CI service providers to supply developers with more tools and guidelines to increase their awareness of the possible interplay between build durations and breakages. We aim in the future to extend our study to include more CI practices from industrial projects to investigate whether our findings would hold.

Chapter 6

A Study of the Considerations for Adopting Caching in Travis CI

This chapter describes our study of the CI caching configuration and what developers should consider when adopting caching in their builds. Section 6.1 presents the research problem and motivation of our study. Section 6.2 presents the study design of our empirical study. Section 6.3 presents the results and findings of our study. Section 6.4 discusses the implications of our findings for researchers and tool builders. Section 6.5 describes the threats to the validity of our results. Finally, Section 6.6 summarizes the chapter and suggests future work.

6.1 Problem and motivation

CI builds may take too long to run, which wastes lots of resources and impedes software developers from engaging in other development activities. CI services offer the capability of caching artifacts that do not change frequently [98]. As reported by prior work, the adoption of CI caching helps to speed up builds [34, 43]. However, caching is not enabled in CI builds by default, i.e., developers need to manually configure builds to specify which artifacts to cache. Yet, developers may (a) lack awareness of which artifacts change less frequently (b) misuse the caching configuration, and (c) invest time/effort to maintain the CI cache during the development lifetime. In addition, it is unclear from prior research whether CI caching is always useful.

According to the Travis CI team, "*There is pros and cons of using the cache. It is up to you [developers] to decide whether you want to use it.*".[1] Although CI caching can speed up builds, the challenges (e.g., maintenance effort) and issues (e.g., unexpected breakages) that might arise when adopting CI caching remain unexplored. In addition, there is lack of empirical research on what software developers should consider to adopt CI caching carefully. Furthermore, prior research has paid little attention to (a) how well CI caching is adopted and maintained in open source projects and (b) whether CI caching brings advantages to builds of every project. Studying the current practices, challenges, and issues of CI caching can help developers understand what to expect when adopting CI caching to avoid any unnecessary development overhead.

In this chapter, we conduct an empirical study on the practices of adopting CI caching in Travis CI. To this end, we analyze 513,384 builds from 1,279 GitHub projects that adopt Travis CI. We investigate the adoption rate of CI caching in software projects to understand the reasons behind the early or proactive adoption of CI caching and why projects still do not adopt or delay the adoption of CI caching. We group the studied projects into five categories based on caching adoption, and model the differences between such categories using logistic regression. To gain further insights on caching adoption, we submit pull requests to enable the CI *cache* in projects that never adopted CI caching. We also comment on the pull requests/commits by

[1]https://github.com/travis-ci/travis-ci/issues/6396#issuecomment-240926751

which CI caching was adopted proactively or belatedly, asking for justifications about such a practice. Moreover, to better understand the overhead that caching adoption may introduce, we mine the process of maintaining the CI cache, the impact of caching on CI builds, and the commonly reported issues about CI caching.

The goal of this study is to (a) analyze the adoption rate of CI caching, (b) understand the common activities to maintain CI caching, (c) analyze the extent to which CI caching reduces the build duration, (d) investigate the overhead of CI caching on the building process, and (e) explore the issues and challenges related to CI caching. Based on the above goals, we address the following RQs:

$RQ_{6.1}$: What urges software projects to adopt CI caching?

CI caching is supposed to speed up builds. However, we observe that CI caching is not adopted in 70% of the projects in our dataset. Our investigation reveals that not adopting CI caching is due to lack of awareness of the CI caching support, not finding caching useful, or moving to other CI services. Developers of such projects incur opportunity costs by disregarding the power of caching.

$RQ_{6.2}$: How do developers maintain CI caching?

Even when caching is enabled, developers need to regularly maintain the caching configuration to reflect the latest code changes. Yet, maintenance activities related to CI caching remain unknown. We mine the process of changing the CI caching configuration in our studied projects and identify five common patterns of activities for maintaining CI caching. Nevertheless, we observe that the majority (76%) of the projects perform no maintenance on CI caching after its adoption. Developers should maintain the cached artifacts regularly to keep the benefit of CI caching.

$RQ_{6.3}$: To what extent does CI caching reduce the build duration?

Prior research has paid little attention to whether CI caching always reduces the build duration. We model the difference in the build duration before and after the adoption of CI caching to elicit whether CI caching is associated with a reduced build duration. We observe that two-thirds of the projects experience no decrease in the build duration after adopting CI caching. Our analysis shows that 80% of those projects do not maintain CI caching regularly. Developers should not consider CI caching as "one size, fits all".

$RQ_{6.4}$: How much overhead does CI caching introduce to builds?

CI caching adds additional building steps, such as downloading and uploading the cache from/to the CI server. However, it is unclear whether such steps delay CI builds. Analyzing build logs reveals that cache uploads occur in 97% of the builds and take $6x$ longer than cache downloads. We find that frequent cache uploads happen due to caching frequently changing artifacts (e.g., installation logs). Developers should monitor build logs to spot any abnormal cache uploads and exclude unnecessary artifacts from the CI cache.

$RQ_{6.5}$: What issues do developers face with CI caching?

Adopting CI caching may bring additional issues to developers, such as occasional build breakages. Highlighting caching issues can help developers avoid such issues and at least decide whether to adopt CI caching or not. In this RQ, we analyze the caching-related issues reported to the TRAVIS CI team. Our results indicate that developers have common issues with cache storage, where artifacts get corrupted or outdated. Recommendations of the CI team suggest that clearing the CI cache regularly helps resolve most caching-related problems.

6.2 Study Design

This section presents the experimental setup of our empirical study. We explain how we collect and prepare the data for our studied RQs.

6.2.1 Data Collection

Figure 6.1 shows an overview of our study. Our study is based on data collected from TravisTorrent [15], a commonly used dataset to study CI builds [14, 68, 76]. TravisTorrent contains builds from $1,283$ projects: 886 Ruby, 393 Java, and 4 JavaScript projects. We exclude the four JavaScript projects, since they cannot be used as a representative sample of the Javascript language. The last build in our dataset was triggered in August 31, 2016. Each project in TravisTorrent contains builds that belong to different repository branches. In this study, we consider only the main (i.e., *master*) branches of the projects, since they are more active and contain more builds than other branches. We identify the main branch of a repository by using the GitHub API.[2] As a result, we obtain $513,384$ builds. For each build, we collect the build logs of the build jobs using the Travis API.[3]

6.2.2 Data Processing

In this section, we explain how we process the data of the $1,279$ projects.

[2]https://docs.github.com

[3]https://docs.travis-ci.com/api

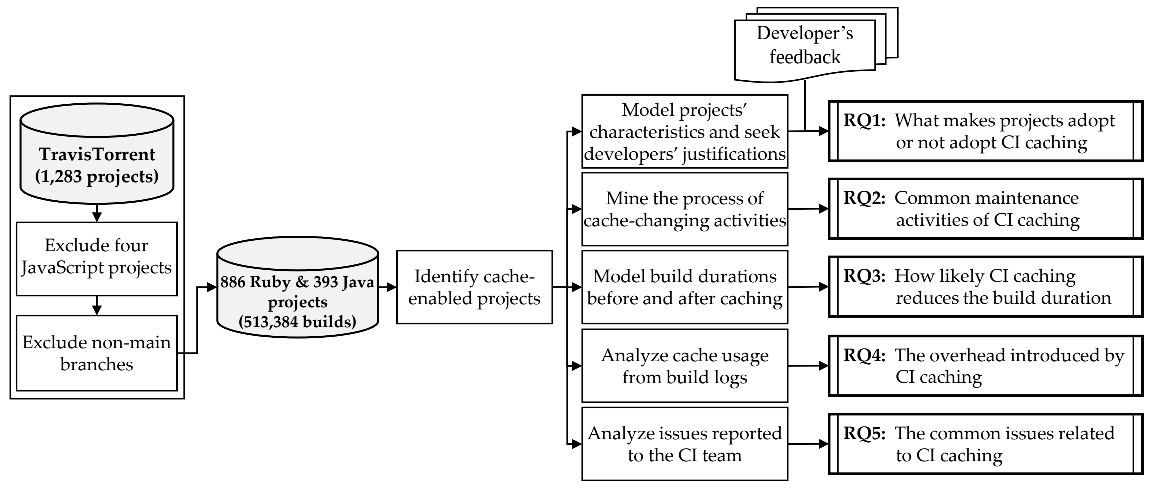

Developer's feedback
TravisTorrent (1,283 projects)
Exclude four JavaScript projects
Exclude non-main branches
886 Ruby & 393 Java projects (513,384 builds)
Identify cache-enabled projects
Model projects' characteristics and seek developers' justifications
Mine the process of cache-changing activities
Model build durations before and after caching
Analyze cache usage from build logs
Analyze issues reported to the CI team
RQ1: What makes projects adopt or not adopt CI caching
RQ2: Common maintenance activities of CI caching
RQ3: How likely CI caching reduces the build duration
RQ4: The overhead introduced by CI caching
RQ5: The common issues related to CI caching

Figure 6.1: Overview of our empirical study of caching in CI

Computing project-level metrics. The analyses performed in our study are project-wise, i.e., we study various aspects related to CI caching adoption at the project level. In particular, we perform the following:

- We identify whether a project adopts CI caching by analyzing the `.travis.yml` configuration file.
- We identify when a project has adopted CI caching by locating the first commit in which CI caching was enabled.
- We identify the characteristics of the projects that adopt and do not adopt CI caching by analyzing the last build of each project before adopting CI caching.

We study the relationship between project characteristics and CI caching adoption by considering the following:

- *Programming Language:* Caching requirements can differ from one language to another. We study whether the CI caching adoption is associated with the programming language of the projects.

Table 6.1: Description of the project-level metrics. All metrics are computed before a project adopts CI caching

Project characteristic	Project-level metric	Description
Programming Language	Language	The GitHub dominant programming language of a project
Code Maturity	Size (SLOC)	Number of source lines of code of a project
	Dependencies	Number of dependencies (i.e., libraries, packages, or gems) a project relies on
	Test density	Median number of test cases per 1,000 SLOC of a project
Development Activity	# of commits per lifetime	Ratio of the number of commits per project age
	Growth rate	Ratio of the relative increase/decrease (i.e., delta) in the lines of code
	Team size	Median number of developers at each build
	# developers	Number of unique developers contributed to a project
CI Activity	CI lifespan	Time (in terms of days) since a project adopted Travis CI
	# of builds	Number of CI builds triggered by a project
	Building frequency	Frequency (in terms of days) of triggering builds in a project
	Configuration ratio	Ratio of commits that change the build configuration file (`.travis.yml` per project lifetime
	Configuration frequency	Frequency (in terms of days) of changing the build configuration file (`.travis.yml` of a project
	Unique job environments	Number of unique integration environments used as build jobs in a project
	Build duration	Median build duration of a project

- *Code Maturity:* Mature projects may vary from projects that are still at the early development stages. We study whether the code maturity (e.g., SLOC, test density, and # of dependencies) relates to CI caching adoption.

- *Development Activity:* Projects have different levels of activity (e.g., committing more frequently). We study whether more active projects adopt CI differently from those projects with fewer activities.

- *CI Activity:* CI builds may need frequent maintenance to cope with code changes. We study whether CI activities (e.g., build configurations or durations) are associated with how projects adopt CI caching.

For each aspect of project characteristics, we compute 14 project-level metrics based on builds triggered before the caching adoption. Table 6.1 shows the description of project-level metrics used as independent variables in our models. We make our dataset available online [4].

Performing correlation and redundancy analyses. We use the project-level metrics as independent explanatory variables to fit our logistic regression models (See Section 6.3.1:$RQ_{6.1}$). We first exclude the highly correlated independent variables, since they can adversely affect regression models [26]. We follow the guidelines provided by Harrell [47] on regression modeling. In particular, we use the Spearman rank ρ clustering analysis [88] (using the `varclus` function from the `rms`[4] *R* package) to identify highly correlated variables. For each pair of correlated variables in which $|\rho| > 0.7$, we keep one variable in our logistic regression models and remove the other variable. According to the principle of parsimony in regression modeling, simple variables should be preferred over complex variables [101]. Considering that our variables are equally simple in terms of computation, we keep the variables that are more informative about the development and building processes. For example, we exclude the '*Team size*' variable, since it is highly correlated with the '*# of unique developers*' variable. We also perform a redundancy analysis of the remaining variables, since they can distort regression models [47]. We use the `redun` function from the `rms` *R* package to identify variables that can be modeled by other variables with $R^2 \geq 0.9$. As a result, we find no redundant variables.

6.3 Experimental Results

In this section, we discuss the motivation, approaches, and findings of our research questions.

[4]https://cran.r-project.org/web/packages/rms/rms.pdf

6.3.1 RQ$_{6.1}$: What urges software projects to adopt CI caching?

Motivation. CI caching allows builds to download software artifacts directly from the CI server rather than installing them from original sources. Despite the potential speed up of caching, projects might neglect the opportunity of adopting CI caching. In this RQ, we (i) investigate the proportion of projects that adopt CI caching, and (ii) analyze the factors that are associated with adopting, not adopting, or delaying the adoption of CI caching.

Approach. To identify which projects adopt CI caching, we analyze the configuration files (i.e., `.travis.yml`) of the builds of the studied projects. We check whether a configuration file contains the '`cache:`' directive. After identifying the projects that adopt CI, we investigate the first date on which CI caching has been enabled. To do so, we identify the first commit in which the '`cache`:' directive has been added to the build configuration file. Then, we calculate the difference in days between the first date of enabling CI caching in a project and the date on which CI caching was announced to be available to open source projects (i.e., *December* 17, 2014). Figure 6.2 shows the distribution of days that passed until caching is adopted (i.e., *adoption delay*) in the studied projects. We use the 1^{st} quantile (41 days) and 3^{rd} quantile (12.2 months) of the caching adoption delay as thresholds to identify *early*, *ordinary*, and *late* projects. We consider projects to be (i) *early* adopters if the delay is between 0 and 41 days, (ii) *ordinary* adopters if the delay is between 41 days and 12.2 months, and (iii) *late* adopters if the delay is between 12.2 and 20 months. As a result, we group the studied projects into five categories, as follows.

- ***Non-adopters:*** projects that never adopt CI caching in their builds (899

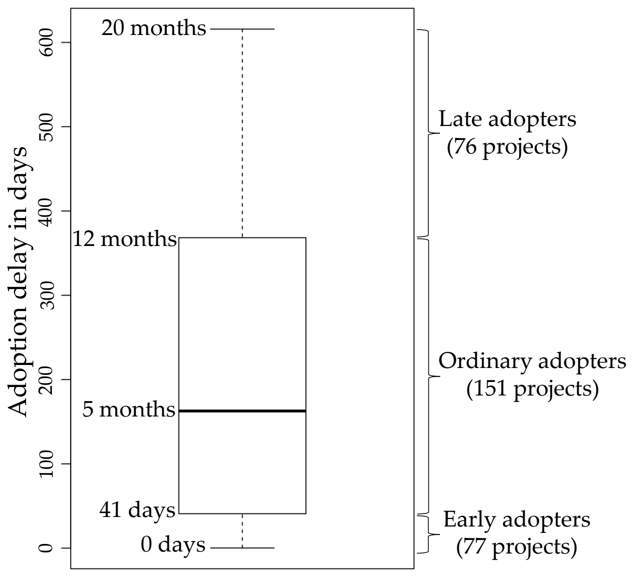


Figure 6.2: Boxplot of the CI caching adoption delay

projects).

- ***Proactive adopters:*** projects that adopt CI caching before it was officially available (76 projects).
- ***Early adopters:*** projects that adopt CI caching within 41 days of being officially available (77 projects).
- ***Ordinary adopters:*** projects that adopt CI caching within a year of being officially available (151 projects).
- ***Late adopters:*** projects that adopt CI caching after a year of being officially available (76 projects).

To uncover the factors behind the none/early/late adoption of CI caching, we model the differences between the five categories of projects using project-level metrics. To this end, we use Generalized Linear Models (GLM) for logistic regression to explore the factors that are associated with delaying the adoption of CI caching or not adopting CI caching at all. In particular, we fit five GLM models, one model per each category, using the `glm` function provided by the `stats`[5] R package. In each model, we explore the differences between projects of a specific category (e.g., *early* adopters) and projects of other categories. Therefore, each model maintains a logical dependent variable: *TRUE* if a project belongs to the category being modeled and *FALSE* otherwise. We use 12 project-level metrics (described in Section 6.2.2) as independent variables in our models.

Alongside our modeling results, developers may have other reasons for not adopting CI caching (e.g., being unaware or careless). To further investigate why CI caching is not adopted, we seek insights from developers who contribute to the projects that never adopt caching on Travis CI. In particular, from the 899 projects that do not adopt CI caching, we manually fork 160 studied projects that (i) have not adopted caching since the data was collected and (iii) have recent commits (i.e., in 2020 or after). We trigger CI builds for all these projects using the last commit of their main repository branches (i.e., without performing any changes). We exclude 71 projects in which CI builds did not pass. We also exclude four projects that have no dependencies. For the remaining 85 projects, we push a commit in which we enable dependency caching. After we make sure that all builds triggered by our commits are *passing* , we submit a pull request to each project. In our submitted pull requests, we ask for clarification as to why CI caching has not been enabled in those projects.

To understand the reasons behind the *proactive* and *late* adoption of CI caching, we raise inquiries to those projects by commenting on the commits or pull requests that enable the CI *cache*. We exclude all projects that are archived or not active

[5]https://www.rdocumentation.org/packages/stats

Table 6.2: Results of modeling the differences between projects based on the caching adoption delay

Project characteristic	**Project-level metric**	***None***		***Proactive***		***Early***		***Ordinary***		***Late***	
		χ^2%	Signf.$^+$	χ^2%	Signf.	χ^2%	Signf.	χ^2%	Signf.	χ^2%	Signf.
Language	Language (Ruby)	1.09		16.11		7.95	.	0.3		16.85	.
Code Maturity	Size (SLOC)	4.51		6.27		1.08		0.71		0.55	
	Test density	0.42		13.33		8.24	.	0.84		3.05	
	Dependencies	8.11	.	3.57		**16.38**	* (↗)	7.33		5.58	
Development Activity	# of commits per lifetime	8.86	.	8.42		**29.15**	*** (↗)	2.27		0.09	
	Growth rate	2.12		0.04		4.46		0.35		0.28	
	# of developers	**34.18**	*** (↘)	19.39	.	8.67	.	14.2		1.12	
CI Activity	CI lifespan	7.37	.	2.94		**17.19**	** (↗)	**56.93**	** (↘)	0.34	
	Building frequency	2.55		9.21		4.52		0.51		4.94	
	Configuration ratio	**17.62**	** (↘)	1.64		0		1.67		**51.81**	** (↗)
	Build environments	0.89		4.52		2.3		0.09		3.7	
	Build duration	**12.28**	* (↘)	14.57		0.05		14.82		11.69	

$^+$Significance codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

anymore. For *proactive* caching adopters, we ask developers from 74 projects about the reasons why caching is enabled before it was made available for open source projects. For *late* caching adopters, we ask developers from 71 projects about the reasons behind the delay in adopting CI caching in their builds. We wait two weeks to receive responses from the developers of these projects.

Findings. Table 6.2 shows the results obtained from our logistic regression models of the differences between the studied projects in adopting CI caching. In Table 6.4, we summarize the significant characteristics associated with CI caching adoption for each category. In Table 6.3, we show a summary of the responses we obtain from developers to our submitted pull requests and comments to none/proactive/late caching adopters. We expect a quick and high acceptance rate for our pull requests regardless of how much speed up CI caching brings to CI builds, since CI caching is less likely to cause issues to CI builds. However, we observe that only 36% of our pull requests have been approved/merged by project maintainers and some pull requests took aver a month to be accepted.

Table 6.3: Summary of the responses obtained from developers regarding the none, proactive, and late adoption of CI caching. Links to pull requests and commits can be found in APPENDIX:TABLES B.1 and B.2

Caching adoption	Action	# (%) of responses	#	Response
Non-adopters (pull requests)	*Approved/Merged*	31/85 (36%)	9/31	No reason given
			9/31	Appreciation
			6/31	Unaware of the caching support
			4/31	Merged as the build passes
			2/31	Assumption that caching was enabled by default
			1/31	Developers are busy or have other priorities
	Open (with response)	7/85 (8%)	5/7	Not useful and/or introduced problems
			2/7	Moving to other CI services
	Closed	9/85 (11%)	9/9	Moving to other CI services
Proactive adopters (comments)		22/74 (30%)	9/22	Caching was available unofficially (pre-release)
			5/22	Developers' misunderstanding
			4/22	Mirrored or copied from another project
			3/22	Developers cannot recall / no specific reason
			2/22	Travis CI is no longer used
Late adopters (comments)		40/71 (56%)	21/40	Unaware of the caching support
			12/40	Developers are busy or have other priorities
			7/40	Developers cannot recall / no specific reason

Table 6.4: A summary of the characteristics of project categories with respect to CI caching adoption

Category	Characteristics
Non-adopters	Have slower builds, fewer build configurations, and fewer developers
Proactive adopters	Have no significant differences from other projects
Early adopters	Have more dependencies, commits, and CI lifespan
Ordinary adopters	Have less CI lifespan
Late adopters	Have more build configurations

Over two-thirds of the studied projects do not adopt CI caching. Despite being available to every GitHub project, we find that CI caching has been adopted by only 30% (380 out of $1,279$) of the projects in our dataset. We observe from Figure 6.2 that about two-thirds of the caching adopters took over a month before enabling caching in their builds. Nevertheless, 20% of the studied projects adopted CI caching proactively. Based on the responses received from developers, 39% of such projects were given an opportunity to use CI caching unofficially (i.e., a preview), whereas 39% of the projects were configured to use caching configuration by mistake (e.g., misunderstanding or a *copy/paste* from other projects). In addition, we observe that projects that do not adopt CI caching have significantly shorter build durations, fewer developers, and fewer build configurations. This result hints that projects that do not adopt CI caching are not as active or mature enough to seek the benefits of CI caching. Our analysis of the responses received from projects that did not adopt CI caching reveals several reasons, such as (a) indifference due to relying on other CI services than Travis CI (21%), (b) lack of awareness of the caching support from Travis CI (19%), or (c) having the impression that caching is not useful or might cause occasional problems (12%). Still, the caching support may attract developers to adopt Travis CI back (after being disabled in the past).[6] These results suggest CI services to devise better ways to communicate updates to developers about the CI features that can improve build performance, such as caching.

Projects with shorter build durations, smaller development teams, and fewer build configurations are unlikely to adopt CI caching. We observe that the build durations of projects that do not adopt CI caching are significantly shorter than other projects. This result is confirmed by our investigation of projects

[6]https://github.com/ccw-ide/ccw/commit/032e253

in which developers indicate that caching is unnecessary because builds run fast already.[7,8,9] In addition, we observe that projects that do not adopt CI caching have significantly fewer developers (median = 10) than those projects that adopt CI caching (median = 14). These findings suggest that (a) having more developers can increase the awareness of the CI caching support and (b) the demand for caching increases when projects have longer build durations (i.e., more developers are expected to wait for builds to finish).

The larger the number of commits and dependencies, the more likely for a project to early adopt CI caching. Caching the binaries of dependencies to be used by future builds gains the maximum benefit of CI caching. Our studied projects have a median of 39 dependencies. Our models show that the number of dependencies is significantly associated with the early adoption of CI caching. An increase of ten dependencies increases the odds of adopting CI caching earlier by 4%. In addition, we observe that projects that do not adopt CI caching have 29% fewer dependencies than all projects that adopt CI caching. This result indicates that developers of projects with fewer dependencies anticipate a minimal speed up to their builds. For example, developers of the `test-unit/test-unit` project, which has ten dependencies, indicated that the speed up introduced by CI caching was insignificant to their builds.[10] It is worth noting that caching can be used for artifacts other than dependencies (e.g., arbitrary directories[11]) in which changes are less frequent.

[7]https://github.com/apache/storm/pull/902
[8]https://github.com/davidmoten/rxjava-jdbc/pull/96
[9]https://github.com/Esri/geometry-api-java/pull/283
[10]https://github.com/test-unit/test-unit/pull/123
[11]https://docs.travis-ci.com/user/caching/#arbitrary-directories

The late adoption of CI caching is strongly associated with more build configuration activities. We observe that projects that have more build configuration activity tend to delay the CI adoption to later stages. For example, the `rubinius/rubinius` project has over 100 build configuration changes during its 5-year TRAVIS CI lifespan. Yet, the `rubinius/rubinius`[12] project has adopted CI caching belatedly (i.e., about 1.5 years after caching was made available for open source projects). We find that, before adopting CI caching, developers performed many build configuration changes, part of which were associated with speeding up builds (e.g., by removing build jobs). Moreover, our investigation of the commits/pull requests in which developers enable CI caching reveals that the late adoption of caching was mainly due to (a) lack of awareness of the caching support from TRAVIS CI (54%), or (b) being busy on other project priorities than caching (31%).

6.3.2 RQ$_{6.2}$: How do developers maintain CI caching?

Motivation. *RQ$_{6.1}$* reports that the majority of projects do not adopt CI caching. Even when caching is enabled, developers need to regularly maintain the caching configuration to reflect the latest code changes (e.g., adding/removing dependencies). While regular maintenance of CI caching is important, it can introduce overhead (i.e., time and effort) to developers, like any other build maintenance activities [70]. In this RQ, we investigate the historical changes of caching configuration to understand the common patterns of CI cache-changing activities.

Approach. We identify the commits that (a) change the caching configuration (i.e., the '`cache`' and '`before_cache`' directives) in the build configuration file (i.e.,

[12]https://github.com/rubinius/rubinius/pull/3660

`.travis.yml`) or (b) contain the '*cache*' or '*caching*' terms in the commit message. We obtain 763 cache-changing commits from the 380 projects that adopt CI caching (with a median of one cache-changing commit per project). We manually analyze those commits to verify whether the changes are related to caching. As a result, we exclude 180 commits in which the change does not affect CI caching (e.g, formatting or reorganizing the build configuration). Finally, we assign a label to each of the 583 remaining commits. These labels represent the type of activity related to CI caching.

We generate a list of chronologically sorted cache-changing activities for each project. To identify the patterns of CI cache-changing activities, we use the ProM 6 framework, a process mining framework [100], in which all the identified cache-changing activities are converted into event logs. Process mining allows to uncover a general process behind events, which are in our cases caching activities. Using ProM 6, we generate the common maintenance patterns, event timeline, and state chart diagrams associated with the provided cache-changing events. We use the *Fuzzy Miner* plugin of ProM6 to abstract the mined process model and generate a corresponding fuzzy graph [45]. We refer to the first and last cache-changing activities of each project as *first event* and *last event*, respectively. Moreover, we extract the type of artifacts being cached by the `cache` directive to investigate what projects cache in common.

Findings.

Only* 24% *of the studied projects perform maintenance activities on the CI caching configuration. Our analysis of the cache-changing commits shows that the majority (76%) of the projects enable CI caching only once with no further maintenance. This result confirms findings of prior research [13] in which over 80% of configuration files, in general, are never changed after creation. We find that projects with a one-time caching configuration tend to cache all the dependencies that are required by the dependency management tool (e.g., *Bundler*, *Maven*, or *Gradle*). While such a practice allows builds to install only newly added dependencies, it can lead to over-caching.[13] As a result, dependencies that are cached but unused anymore by a project (e.g., outdated dependency versions) can be stale in the CI cache. If the CI cache does not change for a certain period, Travis CI clears them automatically.[14] However, caching a frequently updated dependency can leave the cache uncleared for longer periods, which can cause problems to CI builds.[15] Thus, Travis CI allows developers to manually clear CI cache from the CI server. Yet, we could not find evidence that developers perform such a practice, since Travis CI does not keep records of cache storage activities. Therefore, developers should avoid over-caching dependencies in their builds and consider clearing the CI cache regularly.

There are five unique maintenance activities in which developers update the CI caching configuration. Table 6.5 shows the cache-related activities that developers perform (at the commit level). For the 24% of the projects, of which CI caching is maintained, we observe a median of three caching-related activities per project. The most frequent activity after enabling CI caching is updating (i.e., adding/removing) the directories to be considered for caching. Although updating cached directories can improve the build performance, developers should take into consideration that directories that were previously stored in the CI cache are not removed automatically. In addition, only 14% of cache-maintaining projects exclude unnecessary artifacts from the cache. In addition, we observe 50% of cache-changing

[13]https://eng.localytics.com/best-practices-and-common-mistakes-with-travis-ci
[14]https://docs.travis-ci.com/user/caching/#caches-expiration
[15]https://github.com/travis-ci/travis-ci/issues/3758

activities are submitted through pull requests or in response to reported issues. This result may indicate that (a) developers maintain the caching configuration on separate branches before applying such a change to the main repository branch, or (b) the cache configuration is proposed by external developers. Therefore, developers should be more careful when updating cached directories and consider excluding any frequently changing artifacts before uploading the cache to the CI server.

The process of changing the CI cache configuration is unsystematic and repetitive. Figure 6.3 shows a fuzzy graph that corresponds to the cache-changing activities of the studied projects. The fuzzy graph presents the frequency of each cache-changing activity. The thickness and numeric values assigned to an arrow in Figure 6.3 represents the percentage of events (i.e., cache-changing activities) making a transition from one activity to another. We observe that cache-related events can be circular (e.g., enable-disable-enable) or repetitive (e.g., subsequently occurs multiple times). As Figure 6.3 depicts, developers disable CI caching at a certain point of time, but in 85% of the cases, developers enable CI caching again. This result indicates that changing CI caching by developers can be arbitrarily in a 'trial and error' manner. For example, we find that developers might disable CI caching temporarily until a build breakage is resolved.[16] In addition, we observe that developers update the cached directories multiple times (up to eight times per project). This suggests that developers face difficulties to decide which artifacts should be cached. For example, the list of cached directories of the 'SonarSource/Sonarqube' project[17] was updated eight times. The eight cache-updating events involved activities in which developers add artifacts (e.g., `node_modules` and `.sonar/cache`) to the cache and decide later

[16]https://github.com/ms-ati/docile/commit/8c4b9a8
[17]https://github.com/SonarSource/sonarqube

Table 6.5: Statistics of maintenance activities of CI caching

Maintenance Activity	**Freq.**	**First event**	**Last event**
Enable cache	68.2%	99.2%	80.8%
Update cached directories	15.0%	0%	7.9%
Disable cache	7.4%	0.8%	5.3%
Update cache configuration	5.6%	0%	3.2%
Exclude artifacts from cache	2.9%	0%	2.1%
Update directories & Exclude artifacts	0.9%	0%	0.8%

on to remove such dependencies from the list of cached directories. Moreover, we observe that approximately half of the changes of CI caching occurred after a median of 31 days from a previous caching change. We also observe that cache-updating events can happen irregularly at early or later stages of the development lifetime (see the timeline of cache-changing events in Appendix:Figure 6.4). This result encourages (i) developers to pay more attention to the changes that have consequences on the CI cache, (ii) researchers and tool builders to develop tools that keep track of the historical code changes to detect portions of the code that change very often (i.e., should be excluded from the cache) or change less often (i.e., should be cached), and Travis CI to leverage information from build logs to suggest candidate artifacts to be cached.

The majority (67%) of the projects cache all software dependencies. The Travis CI documentation provides examples on how to use CI caching in different programming languages.[18] Although the documentation examples are intended to be simple, we suspect that developers might take such examples as the best caching practices. Our analysis of the `cache` directive in the build configuration files reveals

[18]https://config.travis-ci.com/ref/job/cache

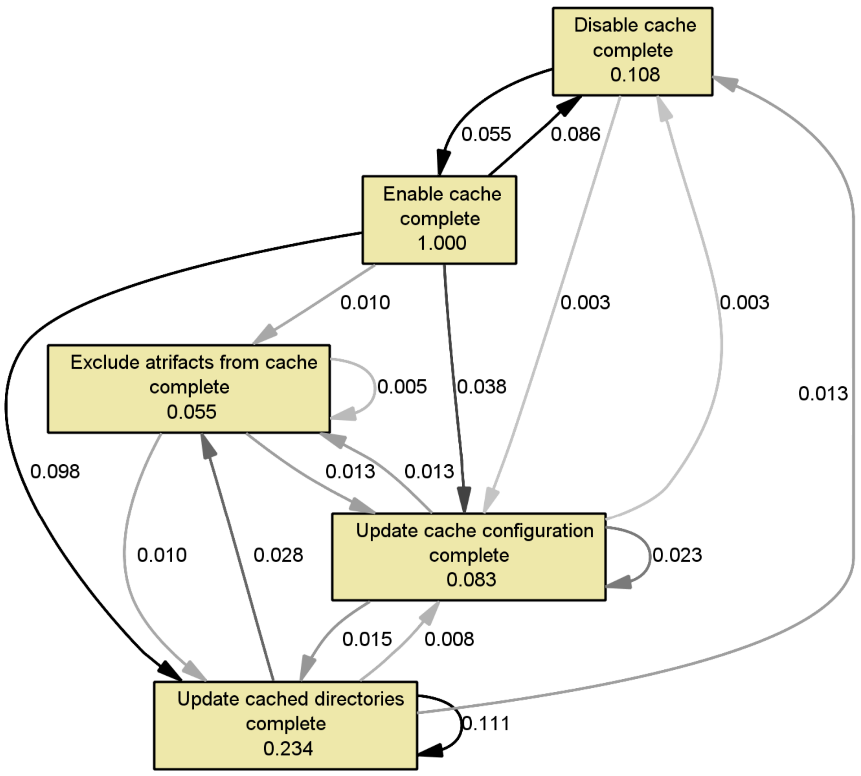

Disable cache complete 0.108
Enable cache complete 1.000
Exclude atrifacts from cache complete 0.055
Update cache configuration complete 0.083
Update cached directories complete 0.234
0.055
0.086
0.010
0.003
0.003
0.005
0.038
0.013
0.098
0.013
0.013
0.010
0.028
0.023
0.015
0.008
0.111

Figure 6.3: A Fuzzy Graph of cache-changing activities

that 80% of Ruby builds and 45% of Java builds cache all build dependencies. The rest of the projects either have a customized caching configuration from the first caching adoption or start with caching all artifacts of a directory and later narrow it down to specific sub-directories. For example, the `cloudfoundry/uaa`[19] project cached all artifacts in the Maven `$HOME/.m2` directory (i.e., depicting the documentation example), but then limited caching to `$HOME/.m2/repository` later on. In addition, developers should take into consideration that dependency management tools, such as *Bundle* or *Maven*, generate log files while compiling the code or installing dependencies. Such log files change for almost every build and, hence, force builds to upload the cache to the CI server [98]. We observe that only a negligible number of the studied projects

[19]https://github.com/cloudfoundry/uaa/commit/53249a7

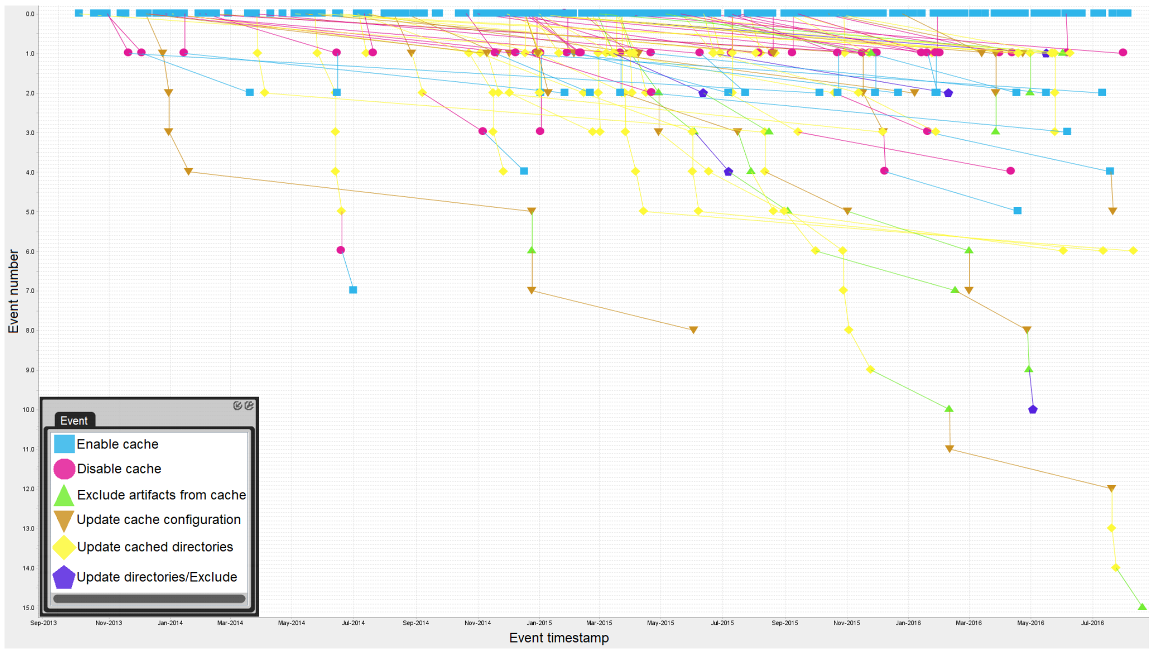


Figure 6.4: An event log timeline of the activities of maintaining the CI cache over time. The y-axis represents the event number, the x-axis represents the time, and the colored shapes represent the type of event.

(i.e., 4%) pre-process artifacts (e.g., excluding certain directories/files) before uploading the cache. For example, the `formtastic/formtastic`[20] project once cached all artifacts in the `vendor/bundle` directory, but then excluded the generated logs (i.e., `gem_make.out,mkmf.log`) from being uploaded to the cache later on. Therefore, we encourage developers to carefully identify and exclude frequently changing artifacts from the CI cache.

6.3.3 RQ$_{6.3}$: To what extent does CI caching reduce the build duration?

Motivation. The number and type of artifacts a project caches can determine how much speed up CI caching can introduce. However, existing research has not studied

[20]https://github.com/formtastic/formtastic/commit/2c13f86

whether the adoption of CI caching has been effective to reduce the build duration in practice. In this RQ, we investigate whether the current practice of CI caching adoption is associated with reducing the build duration.

Approach. We use Regression Discontinuity Design (RDD) [22], a quasi-experimental pretest-posttest design, to model the difference in the build duration before and after the adoption of CI caching. The pretest-posttest design of RDD allows to estimate the causal relations of *interventions* by assigning a *cutoff* above or below which an intervention is assigned [53]. RDD assumes that a trend may continue with no changes if the intervention does not exist. In our case, intervention refers to the adoption of CI caching and a cutoff refers to the time at which CI caching is enabled.

We use RDD to assess whether the adoption of CI caching has an association with the build duration. Therefore, we split the builds of each project in our dataset into two parts: before and after adopting CI caching. We include all the builds before and after the CI adoption. We use the `RDestimate` function in the `rdd` `R` package.[21] In some projects, `RDestimate` may fail to produce results due to internal errors (e.g., insufficient data before or after the caching adoption). In particular, `RDestimate` could successfully model the impact of CI caching adoption on the build duration in 75% of the projects We consider the adoption of CI caching has an impact on the build duration if the RDD estimate is significant (i.e., *p-values* ≤ 0.05). Each RDD model produces a Local Average Treatment Effect (LATE) estimate, which measures the average difference between the expected values of different subsets of builds before and after the adoption of CI caching. We use the LATE estimates to understand whether the adoption of CI caching is positive (i.e., an increase) or negative (i.e., a

[21]https://cran.r-project.org/web/packages/rdd/rdd.pdf

decrease) in the build duration. For example, if modeling the build duration of a project produces a negative LATE estimate, we conclude that adopting CI caching in that project is associated with a decrease in the build duration.

Findings. ***Only a third of the projects have a significant decrease in the build duration after adopting CI caching.*** The results of our project-wise RDD models show that the build duration became significantly shorter after adopting CI caching in 33% of the projects. However, the build duration became significantly longer in 17% of the projects and has insignificant changes in half of the projects. To understand the reasons behind such an anomaly, we analyze the characteristics of projects that have either a significant increase or an insignificant change in the build duration after adopting caching. Table 6.6 shows a list of metrics along with the ratio of projects in which such metrics have changed after adopting CI caching. We observe that $30 - 60\%$ of the projects, in which the CI caching adoption has no association with a decreased build duration, have undergone significant increases in (a) source lines of code, (b) build jobs, and (c) test density. Therefore, besides CI caching, developers are encouraged to investigate other alternatives to speed up builds (e.g., parallelizing test suites or removing duplicate tests).

One-time caching configuration is highly associated with a decrease or insignificant increase to build durations. By analyzing cache-changing commits, we observe that developers of 80% of the projects that have insignificant changes or a significant increase in the build duration configured CI caching only once (i.e., no further maintenance). Analyzing the build logs of those projects reveals that the build *script* phase always dominates the overall build duration. This result hints that, even if caching might perform as expected, it is unlikely to help much with those

Table 6.6: Summary of metric changes before and after caching adoption but non-reduced build durations

Projects with *significantly* increased durations				**Projects with *insignificant* duration changes**			
Metric	Insign. %	Signif. ↗%	Signif. ↘	Metric	Insign. %	Signif. ↗%	Signif. ↘%
SLOC	30	**60**	10	SLOC	28	**41**	31
Test lines/KLOC	25	**55**	20	Build jobs	46	**38**	15
Documentation files changed	50	25	25	Test lines/KLOC	34	**30**	36
Build jobs	44	22	33	Files added	81	19	0
Other files changed	80	20	0	Commits in push	69	15	15
Team Size	38	13	50	Team Size	58	13	29
Test churn	61	11	28	Test churn	79	12	9
Files added	78	11	11	Source churn	82	11	7
Source files changed	80	10	10	Commits on touched files	73	11	16
Commits on touched files	85	5	10	Source files changed	79	10	10
Source churn	85	0	15	Other files changed	89	6	6
Commits in push	80	0	20	Documentation files changed	100	0	0

projects that do not rely heavily on external dependencies.[22] This finding suggests that CI caching should not be considered as "one size, fits all".

6.3.4 RQ$_{6.4}$: How much overhead does CI caching introduce to builds?

Motivation. Caching adoption can improve the performance of CI builds. However, processing (i.e., uploading, downloading, and storing) the CI cache takes time, which might introduce overhead to the building process. Understanding such overhead can help developers estimate whether caching is worth the adoption. In this RQ, we analyze how caches are being processed by CI builds.

Approach. We analyze 611,501 logs of the build jobs of builds in which CI caching is enabled. We identify the dependencies that are downloaded from the CI cache (marked as '`using`') and the dependencies that do not exist in the CI cache (marked as '`installing`') during dependency installation. In addition, we identify the time a build took to download the cache, upload the cache, and install the dependencies

[22]https://github.com/FasterXML/jackson-core/pull/682

that are missed in the cache. Moreover, we identify the dependencies that have been cached but no longer used in subsequent builds. To do so, we analyze the dependencies downloaded and used from the CI cache in a build x_n but are no longer used in the subsequent build x_{n+1}. We compute the cache miss rates as a percentage of dependencies that are downloaded from the CI cache but are no longer used by future builds.

Findings. ***CI builds perform cache uploads very frequently.*** Figure 6.5 shows a beanplot of the time required for downloading the cache, uploading the cache, and installing missed dependencies. Our analysis of build logs shows that 97% of the builds in our dataset perform cache uploads. This result indicates that CI caches most likely include artifacts that change very frequently (almost on every build). We observe that uploading the cache takes a median of 12 seconds (i.e., six times slower than downloading the cache), whereas installing new dependencies takes a median of 8.5 seconds (i.e., 3.25 slower than downloading the cache). Moreover, the longer the duration to upload caches in a build, the more likely for that build to accidentally cache unnecessary files, such as building logs.[23] This result indicates that caching might introduce overhead to the building process if not properly used. Therefore, developers should keep an eye on the generated build logs to spot any abnormal cache uploads and configure their builds to remove any unnecessary files before uploading the CI cache.

One-third of the projects have a cache miss rate of* 33%*. Figure 6.6 shows a beanplot of the median cache miss rates of the studied projects. Our analysis of build logs reveals that two-thirds of the projects have a zero miss rate. However,

[23]https://github.com/twitter-archive/commons/commit/146de30

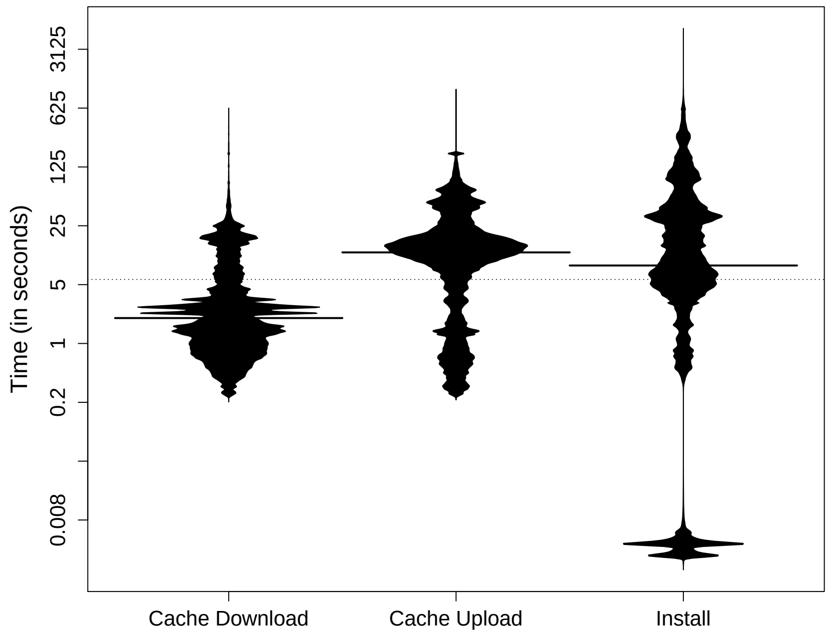


Figure 6.5: A beanplot of the time required to download cache, upload cache, and install missed dependencies

having all dependencies in the CI cache may indicate that those projects have outdated dependencies, which makes such projects prone to security vulnerabilities [77]. Kula et al. [60] found a similar pattern in which more than 80% of projects that heavily rely on dependencies do not update their dependencies. On the other hand, we observe that one-third of the projects in our dataset have frequent dependency updates, which leads to installing new/updated dependencies in the majority of their builds. Although updating dependencies is considered a good practice, developers should carefully identify the most frequently updated dependencies and exclude them from the CI cache. Not excluding such dependencies from the CI caches may introduce delays to the building process, since (a) older dependency versions are downloaded from the cache in every build, and (b) newer dependency versions are installed and

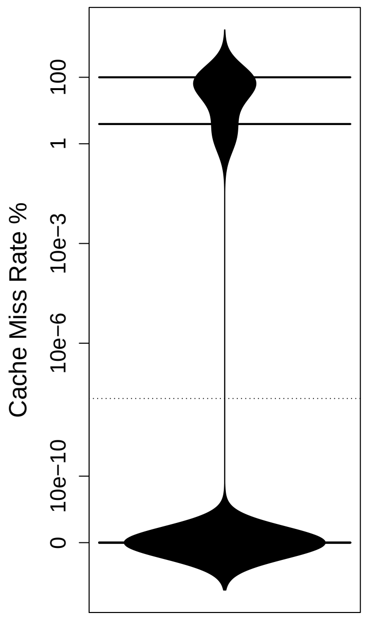


Figure 6.6: A beanplot of the median cache miss rates of the studied projects

then are uploaded again to the cache.

27% ***of the studied projects have unused dependencies in the CI cache***. Caching frequently updated dependencies may lead older dependency versions to stale in the CI cache. Our results show that builds of the majority (i.e., 73%) of the projects use all dependencies stored in the CI cache. This result may indicate that those projects either cache non-updating dependencies only or remove unused dependencies from the cache manually. Yet, there are 27% of the projects in which dependencies that were used by former builds do not appear to be used in subsequent builds. This result indicates that the cache size can keep growing continuously if developers do not consider removing unused artifacts from the cache.[24] For example, developers of two

[24]https://github.com/Cockatrice/Cockatrice/issues/3781

projects in our dataset indicate that having stale cached dependencies is one of the reasons behind not adopting CI caching.[25,26] In addition to the unnecessary delay, unused dependencies in the CI cache can introduce cache storage issues, especially that a single repository may maintain an independent CI cache for every repository branch and language/compiler version.

6.3.5 RQ$_{6.5}$: What issues do developers face with CI caching?

Motivation. CI caching requires regular maintenance. Our findings hint that software developers may encounter issues related to CI caching, which likely urged such maintenance. Prior research has paid little attention to the issues that might arise when using CI caching. Understanding such issues and how to resolve them helps developers better leverage the benefits of caching. In this RQ, we investigate the issues that may arise when CI caching is adopted in a software project.

Approach. We crawl the GitHub caching-related issues reported to the Travis CI team on or before August 31, 2016 (the last build triggering date in our dataset). We use '*cache*' and '*caching*' as key terms to collect only caching-related issues[27] from the Travis CI repository on GitHub. We obtain 373 issues related to caching on Travis CI. These issues represent the common problems that any project may encounter when adopting CI caching with Travis CI. This means that, when developers encounter a caching-related problem, they may not report it to the Travis CI team if already reported before. Developers may also leave comments on existing issue reports to indicate having the same issue/request.[28]

[25]https://github.com/FasterXML/jackson-core/pull/682
[26]https://github.com/airlift/slice/pull/139
[27]https://github.com/travis-ci/travis-ci/issues
[28]https://github.com/travis-ci/travis-ci/issues/4393

Two researchers perform a manual analysis of the issues from GitHub. We use open coding [23] to identify the common caching issues that developers may encounter. We use the negotiated agreement technique [19] to help resolve any disagreements in our manual analysis. Using such a technique, the two co-authors collaborate on coding a sample of 50 issues. We group the codes that emerged from issues into common categories. Then, we take another sample of 50 issues to analyze whether we would encounter new codes. We repeat this process until no new codes are identified.

Findings. ***Outdated or corrupted caches are the most common issues related CI caching.*** Although errors that may occur while downloading or uploading CI caches do not lead to build breakages, users tend to report issues about build breakages that occur when CI caching is adopted. Our analysis of the cache-related issues reported to Travis CI shows that build breakages that are associated with CI caching are mostly because (i) the CI cache is corrupted,[29] (ii) the CI cache does not update dependency versions,[30] or (iii) multiple build jobs or repository branches point to the same cache file (e.g., the cache of one build job is overwritten by another build job).[31] The Travis CI team responds to every issue and asks clarification questions or supporting materials. We find that caching-related issues are commonly fixed in future releases of Travis CI.[32] Yet, the Travis CI team may sometimes be unable to identify the root cause of the issues and, hence, suggest that developers should manually remove the cache from the CI server as an attempt to resolve the issue.[33] We find that manually clearing the cache resolves most (i.e., 62%) of the analyzed

[29]https://github.com/travis-ci/travis-ci/issues/6396
[30]https://github.com/travis-ci/travis-ci/issues/3758
[31]https://github.com/travis-ci/travis-ci/issues/4657
[32]https://github.com/travis-ci/travis-ci/issues/4393#issuecomment-219776423
[33]https://github.com/travis-ci/travis-ci/issues/3758#issuecomment-106989375

issues (based on the responses of developers who reported the issues or those who have encountered and confirmed similar issues in their repositories). We summarize the common caching-related issues reported by developers along with the proposed fixes/workarounds by the Travis CI team in Appendix:Table B.3.

The CI caching support does not satisfy all development needs but is continuously evolving. Our manual analysis of CI caching issues shows that developers request more caching capabilities from the Travis CI team. For example, we find a common issue in which developers request caching support for builds that run under certain environments (e.g., `macOS`[34]). In addition, developers may request a go-to directive that simplifies caching certain artifacts (e.g., `ccache`) instead of manually listing of directories in the build configurations. We observe that the Travis CI team often provides positive responses to user requests (i.e., by offering support in future releases).[35] In some cases, the Travis CI team members propose workarounds to resolve the issues raised[36] or disagree with the raised arguments.[37] This result encourages developers to be up-to-date with the latest capabilities of CI caching, which could have solutions to previously unsolved issues.

6.4 Implications

In this section, we discuss the implications of our findings for developers, researchers, and CI services.

[34]https://github.com/travis-ci/travis-ci/issues/4869
[35]https://github.com/travis-ci/travis-ci/issues/4997#issuecomment-216615467
[36]https://github.com/travis-ci/travis-ci/issues/7456#issuecomment-296505058
[37]https://github.com/travis-ci/travis-ci/issues/4191

6.4.1 Implications for developers

What else to cache? Our analysis of the studied projects as well as the CI caching documentation shows that caching is mostly linked to dependencies. However, there are other software artifacts that might also be good candidates for CI caching. For example, prior research has shown that tests are central to the building process and significantly increase the build duration [14]. Hence, developers should consider caching any static data that might be generated while running software tests (e.g., using test input-data generators [58] or search results[38]). In compiler-based languages (e.g., Java) developers should consider performing an on-demand compilation of class files. To this end, an incremental compilation should be used to allow caching previously generated class files and compile only those source files that are detected to be changed. While such kind of caching is supported by Travis CI for C and C++ projects using the `ccache` directive,[39] we find no similar directive for Java projects.

No pain, no gain! Our process mining of CI cache-changing commits reveals that developers may need to maintain CI caching to cope with the changes in their projects. Our results indicate that 80% of the projects that experience no significant improvement in the duration of CI builds do not maintain CI caching. While caching maintenance is important, it may require dedicated time and effort (i.e., can be costly) [70]. Therefore, developers should keep an eye on the caching behavior in their builds with minimal effort.

[38]https://github.com/thredded/thredded/commit/2f04819
[39]https://docs.travis-ci.com/user/caching/#ccache-cache

6.4.2 Implications for CI services & tool builders

Nominating cacheable artifacts. Our findings suggest that one caching configuration may not fit all software projects. For example, caching large files may not lead to significant changes to the build duration, since downloading such files from the CI cache may take as long as downloading them from the remote repository.[40] However, developers may find it hard to tell whether certain directories are cacheable or not, since such a decision might require performing certain experimentation. Hence, developers may need means to detect (a) which project artifacts are appropriate to be cached and (b) which artifacts exist in the cache but no longer used by builds.

Semantic linters to report any inefficient/inappropriate caching configurations. The TRAVIS CI *linter* accepts any parameters given to the caching directive in the build configuration, as long as the syntax is respected and directories exist. However, such a *linter* does not complain if the provided directories are inefficient for caching (e.g., they change very frequently) or do not benefit from being cached (e.g., files that are easy to install but slow to download, such as *JDK* packages). As such, developers should use semantic linters to highlight CI configuration misuses (e.g., [104]), including caching.

Enriching the documentation about CI caching with more practical examples/consequences about caching adoption. CI services (e.g., TRAVIS CI[41]) provide reasonable details for developers on how to enable CI caching in their builds. However, developers may take the sample caching configurations (e.g., caching Ruby's

[40]https://docs.travis-ci.com/user/caching/#things-not-to-cache
[41]https://docs.travis-ci.com

bundler[42]) given the documentation as a role model. Our results indicate that two-thirds of the studied projects cache all dependencies (i.e., as given in the CI documentation). As a result, those projects are less likely to gain benefit from adopting CI caching. Therefore, CI services should consider leveraging the historical building experience of projects to suggest caching options that fit the context of each project. For example, projects that undergo frequent dependency updates should be provided with a recommendation to perform selective caching to allow caching on those dependencies that update less often.

Raising the awareness about CI caching. Our results show that 70% of the studied projects do not adopt CI caching. Those projects have significantly different characteristics (e.g., fewer developers and shorter build durations) from other projects that adopt CI caching. Still, adopting CI caching could speed up the builds of some if not all projects. Our analysis of the build configuration of those projects shows that those projects have never attempted to enable CI caching in their builds to test its effectiveness. Our investigation of a sample of those projects shows that developers are unaware of CI caching or they had the impression that CI caching is enabled by default. CI services should raise developers' awareness of such unexploited CI features as caching. For example, CI services should (a) suggest potentially useful CI features, such as caching, at the time of adopting CI, or (b) analyze the build history of a project to estimate the time that could be saved should caching be adopted, and provide such feedback to developers.

Making selective caching simpler. Our investigation of the CI caching issues reported to Travis CI shows that developers may find it complicated to exclude

[42]https://docs.travis-ci.com/user/caching/#bundler

certain directories from the CI cache. For example, if certain sub-directories are desired to be excluded from a cached directory, developers would need to configure their builds to (i) list all other sub-directories in the `cache` directive, or (ii) cache the parent directory in the `cache` directive and remove the list of unwanted sub-directories in the `before_cache` directive. Doing so can make CI caching adoption more complicated and error prone. Hence, CI services should provide support to perform different caching scenarios (e.g., the use of wildcards or regular expressions) to minimize the effort required by developers to configure builds.

6.5 Threats to Validity

In this section, we discuss the potential threats to the validity of our findings.

Construct validity. Construct threats to validity are concerned with the degree to which our analyses measure what we claim to analyze. In our study, we rely on the data collected and computed mostly from TravisTorrent and from the Git repositories that we clone from GitHub. Mistakenly computed values may influence our results. However, we carefully filter and test the data to reduce the possibility of wrong computations that may impact our analyses in this study. In addition, to identify the issues related to CI caching, we use certain key terms that are most likely to appear in every caching-related issue. Yet, those terms may give us issues that are unrelated to CI caching. To mitigate this threat, we excluded the issues that do not belong to CI caching during the manual analysis performed collaboratively by two co-authors together using open coding. Moreover, we report our findings based on exploratory models and the manual analyses we perform on the data collected from actual projects. We make our dataset available online [4]. Still, future studies should consider surveying developers to further understand their rationale for not adopting CI caching or more practices to maintain the CI cache.

Internal validity. Internal threats to validity are concerned with the ability to draw conclusions from the relation between the independent and dependent variables. In our study, we use project-level metrics to understand the factors that may be associated with (a) the adoption delay of CI caching and (b) the increase/decrease in the build duration. We are aware that the metrics used in our analyses are not exhaustive, i.e., using additional metrics may affect our results. In the cases where two metrics are highly correlated, deciding which metrics to keep in our models may have an impact on the results obtained from the models. To make our results reproducible, we make our choices of the selected metrics explicit for all the pairs of highly correlated metrics.

External validity. External threats are concerned with the ability to generalize our results. Our study is based on builds that are collected from $1,279$ GitHub projects that adopt Travis CI. To mitigate this threat, we include projects that have larger numbers of builds spanning up to six years (we make our dataset publicly available [4]). Yet, we cannot generalize our conclusions to caching of other CI services, such as CircleCI and GitHub Actions. For example, other CI services provide additional instructions to configure CI caching which can open the opportunity for developers to misuse the CI cache, thus leading to more build issues. Future research should investigate caching in other CI services to assess whether our results hold. In addition, a replication of our work using projects from the industry or written in other programming languages is required to reach more general conclusions.

6.6 Summary

In this chapter, we conduct an empirical study of CI caching practices in Travis CI, a cloud-based CI service. We analyze 513, 384 builds from 1, 279 GitHub projects that are linked to Travis CI. In particular, we study the extent to which CI caching is adopted in software projects to understand why projects may neglect to adopt CI caching despite its potential usefulness. In addition, we study the common issues and maintenance activities of CI caching and how the adoption of CI caching may influence build generation. We summarize the key findings of our study as follows:

- CI caching is not adopted by 70% of the studied projects. Project characteristics and lack of developers' awareness are among the factors association with the non-adoption of CI caching. Developers should assess whether the benefits of caching supersede its drawbacks.
- CI caching is not maintained in 76% of the studied projects. Developers should consider maintaining the CI cache to reflect the evolving software changes as caching is not "one size, fits all".
- Two-thirds of the studied projects have no significant decrease in the build duration after adopting CI caching. Without regular maintenance, caching can have a negligible impact on the build performance (i.e., no pain, no gain).
- The majority (80%) of projects cache all build dependencies (as given in the CI documentation). Developers should pay attention to frequently changing artifacts, such as building/installation logs, since they can lead to unnecessary cache uploads.

- Developers should take into consideration that CI caching (i) may introduce unwanted issues, such as corrupt and outdated caches, and (ii) may not support certain caching scenarios (e.g., the CI cache cannot be cleared automatically).

Our findings encourage software engineering researchers and tool builders to work closely with software developers to provide them with mechanisms that mitigate the burden of maintaining CI caches.

We aim in the future to survey/interview software developers to gain more insights about the level of awareness and actual practices of developers to deal with CI caching over the various CI services available in the market.

Chapter 7

Conclusions and Future Work

This dissertation addresses fulfilled CI promises for providing fast and trustworthy feedback to software developers. To do this, we conduct four empirical studies to understand the reasons behind the unfulfilled CI promises: (1) a study of the factors associated with long CI build durations, (2) a study of noisy CI build breakages and their impact on observations of prior research, (3) a study of the interplay between the durations and breakages of CI builds, and (4) a study of the considerations for adopting caching in CI builds.

7.1 Contributions

The main contributions of this dissertation are as follows.

(1) **A catalogue of the factors associated with long build durations (Chapter 3)**. We model long build durations to identify their frequency and the most important factors associated with them. We find that, in addition to well-known factors (e.g., lines of code and test density), misusing the build configuration, rerunning failed commands or tests multiple times, and waiting for long-running

commands are significantly associated with long build durations. We provide insights for developers, researchers, and CI services to help them optimize the duration of CI builds.

Results show that about 84% of CI builds take more than the acceptable 10 minutes duration to run. In addition to common wisdom factors (e.g., project size, team size, and test density), builds can take longer due to workload on CI servers, command retrials, build misconfigurations.

(2) **A catalogue of noisy build breakages and their impact on modeling results (Chapter 4)**. We propose three criteria to identify noisy build breakages, including environmental, cascading, and allowed breakages. We provide a catalogue of environmental build breakages and their frequency in open source projects. We show that neglecting such kinds of breakages can lead to misleading conclusions regarding the factors associated with broken builds.

Results show that 55% of build breakages are unrelated to developer activities (i.e., noisy). Not removing noisy breakages from build breakage models can lead to misleading conclusions regarding the factors associated with build breakages.

(3) **The dual and side effects of reducing build durations and fixing build breakages (Chapter 5)**. We model the interplay between build durations and build breakages using project-level and build-level metrics and investigate their associations with build performance. In addition, we survey software developers to get their feedback about our empirical observations. Overall, we find metrics that have side effects on build durations or breakages. We also find metrics

that have dual reduction of both the build duration and the frequency of build breakages.

Results show that the context of a project (e.g., being a large or active project) has a strong association with build performance, and 76% of survey respondents acknowledge such association. Developers are encouraged to configure builds only when needed to avoid impacting build performance.

(4) **The important considerations for configuring CI caching (Chapter 6)**. We analyze the CI caching adoption rate and investigate why projects do not adopt or delay the adoption of CI caching. We find that the majority of projects do not adopt CI caching mainly due to lack of awareness. We also identify the common activities to maintain the CI cache. We find that not maintaining the CI cache (e.g., caching all dependencies) may lead to overhead on the building process. We highlight the most common issues related to CI caching along with the workarounds to resolve them.

Results show that only 30% of projects adopt CI caching in their builds. Though configuring CI caching is straightforward, not maintaining the CI cache can introduce occasional delays and breakages to CI builds.

7.2 Future Work

The research work presented in this dissertation highlights the actions and considerations that developers should take into account when adopting CI in software repositories. Below, we suggest potential research opportunities that may advance this line of work in the future.

- **Parallelizing builds.** Builds always have limits on how many jobs they can run in a build in parallel. Some build jobs may be configured to run in serial. Yet, software developers tend to deal with build jobs randomly, i.e., developers do not know how many jobs to run and whether to run jobs in parallel or in serial. Future research should study how open source projects (i.e., free CI subscribers) may become more costly than projects with a paid service based on the gain in terms of build durations. In addition, research should invest in developing techniques to allow developers to automate the process of build job parallelization to enable acquiring the most benefit from CI services.

- **Test optimization and prioritization for CI.** The '*run-them-all*' testing strategy might no help developers in a CI context. Future research should explore ways to identify tests that may perform similarly (i.e., semantic test clones) in order to potentially reduce build durations. Developers add more tests whenever a new system functionality is introduced to the project. Hence, due to the frequent additions of tests, developers may neglect writing efficient test cases. In addition, due to parallel development activities, it could be hard for developers to identify whether a test is a duplicate of another existing test. Therefore, researchers may explore ways to prioritize software tests from a CI perspective.

- **Build duration versus system performance.** Users might get the impression that projects in which builds take longer to run are of low performance. Future research should assess whether the build duration has a potential correlation with the performance of the system at runtime. If such a correlation exists, researchers may leverage existing performance optimization techniques to optimize existing software tests. Doing so may help to reduce the build duration.

- **Auto-configuring CI builds.** Misusing a CI configuration may unintentionally be associated with *long* build durations. For example, developers may configure builds to update dependencies in every run to avoid breaking the build in the case of unexpected dependency updates. Hence, builds should be able to optionally perform such an update only if a dependency is recognized to be not up to update. Therefore, future research should investigate approaches to spot build (re)configuration opportunities and provide feedback to developers about the possible (re)configurations for their builds. Feedback may also incorporate information about the reasons why recently triggered builds have longer build durations than the previously triggered builds.

- **Enriching build outcomes.** Existing CI services provide developers with an abstract build status (i.e., *passed* or *broken*). Future research should consider investing in supporting developers with an enhanced user interface to help them understand (a) the differences between individual job breakages, (b) the types of build errors occurred, (c) whether the build would possibly pass if restarted, (d) whether certain build breakages occurred in the past, and (e) what actions developers have previously made to fix that breakages. Build logs are quite rich of information about the reasons behind build breakages. Therefore, such information may be regularly collected by a tool to build a history of the frequent root causes of build breakages. In addition, if builds are frequently broken due to connection errors, a useful tool may consider increasing the number of times to rerun failing commands. Likewise, if builds are frequently broken due to exceeding time or log size limits, a tool may consider increasing the time required to wait for CI commands to run or removing redundant or unnecessary information from being logged, respectively.

- **Detecting inefficient adoption of CI configurations.** This dissertation shows that misconfiguring CI (e.g., caching) might negatively affect build performance. For example, projects with fewer dependencies are unlikely to benefit from the adoption of CI caching. In addition, single-job projects are unlikely to benefit from the *fast_finish* build configuration. Therefore, future research should work on techniques to predict or detect whether a certain CI practice achieves its anticipated benefit, since the current build performance of a project may impede such a practice from performing well.

Bibliography

[1] Online appendix of Chapter 3. https://taher-ghaleb.github.io/thesis/chapter_3/appendix.html.

[2] Online appendix of Chapter 4. https://taher-ghaleb.github.io/thesis/chapter_4/appendix.html.

[3] Online appendix of Chapter 5. https://taher-ghaleb.github.io/thesis/chapter_5/appendix.html.

[4] Online appendix of Chapter 6. https://taher-ghaleb.github.io/thesis/chapter_6/appendix.html.

[5] Software in the DOE: The hidden overhead of "the build". Technical report.

[6] Rabe Abdalkareem, Suhaib Mujahid, Emad Shihab, and Juergen Rilling. Which commits can be CI skipped? IEEE Transactions on Software Engineering, 2019.

[7] Alan Agresti. Tutorial on modeling ordered categorical response data. Psychological bulletin, 105(2):290, 1989.

[8] Glenn Ammons. Grexmk: Speeding up scripted builds. In Proceedings of the international workshop on Dynamic Systems Analysis, pages 81–87. ACM, 2006.

[9] Dave Astels. One assertion per test. http://www.artima.com/weblogs/viewpost.jsp?thread=35578. Visited on February 05, 2018.

[10] Abigail Atchison et al. A time series analysis of TravisTorrent builds: to everything there is a season. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017), pages 463–466, 2017.

[11] Sebastian Baltes, Jascha Knack, Daniel Anastasiou, Ralf Tymann, and Stephan Diehl. (no) influence of continuous integration on the commit activity in GitHub projects. In Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics, pages 1–7. ACM, 2018.

[12] Kent Beck. Extreme programming explained: Embrace change. addison-wesley professional, 2000.

[13] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. Analyzing the state of static analysis: A large-scale evaluation in open source software. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 470–481. IEEE, 2016.

[14] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017), pages 356–367, 2017.

[15] Moritz Beller, Georgios Gousios, and Andy Zaidman. TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. In Proceedings of the 14th International Conference on Mining Software

Repositories (MSR 2014), pages 447–450, 2017.

[16] João Helis Bernardo, Daniel Alencar da Costa, and Uirá Kulesza. Studying the impact of adopting continuous integration on the delivery time of pull requests. In Proceedings of the 15th International Conference on Mining Software Repositories, pages 131–141. ACM, 2018.

[17] Ekaba Bisong, Eric Tran, and Olga Baysal. Built to last or built too fast?: Evaluating prediction models for build times. In Proceedings of the 14th International Conference on Mining Software Repositories, pages 487–490, 2017.

[18] Graham Brooks. Team pace keeping build times down. In Proceedings of the AGILE Conference, pages 294–297. IEEE, 2008.

[19] John L Campbell, Charles Quincy, Jordan Osserman, and Ove K Pedersen. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. Sociological Methods & Research, 42(3):294–320, 2013.

[20] Norman Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. Psychological Bulletin, 114(3):494, 1993.

[21] Jacob Cohen. A coefficient of agreement for nominal scales. Educational and psychological measurement, 20(1):37–46, 1960.

[22] Thomas D Cook, Donald Thomas Campbell, and Arles Day. Quasi-experimentation: Design & analysis issues for field settings, volume 351. Houghton Mifflin Boston, 1979.

[23] Juliet M Corbin and Anselm Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. Qualitative sociology, 13(1):3–21, 1990.

[24] Elizabeth R DeLong, David M DeLong, and Daniel L Clarke-Pearson. Comparing the areas under two or more correlated receiver operating characteristic curves: a nonparametric approach. Biometrics, 44(3):837–845, 1988.

[25] Naji Dmeiri, David A Tomassi, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. BugSwarm: Mining and continuously growing a dataset of reproducible failures and fixes. In Proceedings of the 41st International Conference on Software Engineering, pages 339–349. IEEE Press, 2019.

[26] Pedro Domingos. A few useful things to know about machine learning. Communications of the ACM, 55(10):78–87, 2012.

[27] Norman R Draper and Harry Smith. Applied regression analysis, volume 326. John Wiley & Sons, 2014.

[28] Olive Jean Dunn. Multiple comparisons among means. Journal of the American Statistical Association, 56(293):52–64, 1961.

[29] Olive Jean Dunn. Multiple comparisons using rank sums. Technometrics, 6(3):241–252, 1964.

[30] Paul M Duvall, Steve Matyas, and Andrew Glover. Continuous Integration: improving software quality and reducing risk. Pearson Education, 2007.

[31] PM Duvall. Continuous delivery patterns and antipatterns in the software lifecycle. WWW], Available (accessed on 28.4. 2015): http://refcardz. dzone. com/refcardz/continuous-delivery-patterns, 2011.

[32] Tore Dybå, Dag IK Sjøberg, and Daniela S Cruzes. What works for whom, where, when, and why? On the role of context in empirical software engineering. In Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, pages 19–28, 2012.

[33] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 235–245. ACM, 2014.

[34] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. CloudBuild: Microsoft's distributed and caching build service. In Proceedings of the 38th International Conference on Software Engineering Companion, pages 11–20, 2016.

[35] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In Proceedings of the 8th Working Conference on Mining Software Repositories, pages 153–162. ACM, 2011.

[36] Julian J Faraway. Extending the linear model with R: Generalized linear, mixed effects and nonparametric regression models, volume 124. CRC press, 2016.

[37] Stuart I Feldman. Make: A program for maintaining computer programs. Software: Practice and experience, 9(4):255–265, 1979.

[38] Wagner Felidré, Leonardo Furtado, Daniel da Costa, Bruno Cartaxo, and Gustavo Pinto. Continuous integration theater. In Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2019), pages 1–10, 2019.

[39] Jennifer Fereday and Eimear Muir-Cochrane. Demonstrating rigor using thematic analysis: A hybrid approach of inductive and deductive coding and theme development. International journal of qualitative methods, 5(1):80–92, 2006.

[40] Ronald Aylmer Fisher. Statistical methods for research workers. Genesis Publishing Pvt Ltd, 1925.

[41] Martin Fowler and Matthew Foemmel. Continuous Integration, 2006.

[42] Aakash Gautam, Saket Vishwasrao, and Francisco Servant. An empirical study of activity, popularity, size, testing, and stability in continuous integration. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 495–498. IEEE, 2017.

[43] Taher Ahmed Ghaleb, Daniel Alencar da Costa, and Ying Zou. An empirical study of the long duration of continuous integration builds. Empirical Software Engineering, pages 1–38, 2019.

[44] Taher Ahmed Ghaleb, Daniel Alencar da Costa, Ying Zou, and Ahmed E. Hassan. Studying the impact of noises in build breakage data. IEEE Transactions on Software Engineering, pages 1–14, 2019.

[45] Christian W Günther and Wil MP Van Der Aalst. Fuzzy mining: Adaptive process simplification based on multi-perspective metrics. In International conference on business process management, pages 328–343. Springer, 2007.

[46] James A Hanley and Barbara J McNeil. The meaning and use of the area under a receiver operating characteristic (ROC) curve. Radiology, 143(1):29–36, 1982.

[47] Frank E Harrell. Regression modeling strategies, with applications to linear models, survival analysis and logistic regression. GET ADDRESS: Springer, 2001.

[48] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. Trade-offs in continuous integration: Assurance, security, and flexibility. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE 2017), pages 197–207. ACM, 2017.

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

[50] Sture Holm. A simple sequentially rejective multiple test procedure. Scandinavian journal of statistics, pages 65–70, 1979.

[51] David C Howell. Median absolute deviation. Wiley StatsRef: Statistics Reference Online, 2014.

[52] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In Proceedings of the 22nd ACM SIGSOFT

International Symposium on Foundations of Software Engineering, pages 621–631. ACM, 2014.

[53] Guido W Imbens and Thomas Lemieux. Regression discontinuity designs: A guide to practice. Journal of econometrics, 142(2):615–635, 2008.

[54] Md Rakibul Islam and Minhaz F Zibran. Insights into continuous integration build failures. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 467–470. IEEE, 2017.

[55] Romit Jain, Saket Kumar Singh, and Bharavi Mishra. A brief study on build failures in continuous integration: Causation and effect. In Progress in Advanced Computing and Intelligent Engineering, pages 17–27. Springer, 2019.

[56] Peter Kampstra et al. Beanplot: A boxplot alternative for visual comparison of distributions. Journal of Statistical Software, 28, 2008.

[57] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. Why do automated builds break? An empirical study. In 2014 IEEE International Conference on Software Maintenance and Evolution, pages 41–50. IEEE, 2014.

[58] Bogdan Korel. Automated software test data generation. IEEE Transactions on software engineering, 16(8):870–879, 1990.

[59] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. Journal of the American statistical Association, 47(260):583–621, 1952.

[60] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? Empirical Software Engineering, 23(1):384–417, 2018.

[61] Irwin Kwan, Adrian Schroter, and Daniela Damian. Does socio-technical congruence have an effect on software build success? A study of coordination in a software project. IEEE Transactions on Software Engineering, 37(3):307–324, 2011.

[62] Eero Laukkanen and Mika V Mäntylä. Build waiting time in continuous integration: An initial interdisciplinary literature review. In Proceedings of the Second International Workshop on Rapid Continuous Software Engineering, pages 1–4, 2015.

[63] Yong Lei and James H Andrews. Minimization of randomized unit test cases. In Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on, pages 10–pp. IEEE, 2005.

[64] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pages 417–420. ACM, 2007.

[65] Marko Leppänen, Simo Mäkinen, Max Pagels, Veli-Pekka Eloranta, Juha Itkonen, Mika V Mäntylä, and Tomi Männistö. The highways and country roads to continuous deployment. Ieee software, 32(2):64–72, 2015.

[66] Andrew J Lewis. Mixed effects models and extensions in ecology with R. Springer, 2009.

[67] Jingjing Liang, Sebastian Elbaum, and Gregg Rothermel. Redefining prioritization: continuous prioritization for continuous integration. In Proceedings of the 40th International Conference on Software Engineering, pages 688–698. ACM, 2018.

[68] Yang Luo, Yangyang Zhao, Wanwangying Ma, and Lin Chen. What are the factors impacting build breakage? In Proceedings of the 14th Conference on Web Information Systems and Applications Conference (WISA 2017), pages 139–142. IEEE, 2017.

[69] Robert Malouf. A comparison of algorithms for maximum entropy parameter estimation. In COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002), 2002.

[70] Shane McIntosh, Bram Adams, Thanh HD Nguyen, Yasutaka Kamei, and Ahmed E Hassan. An empirical study of build maintenance effort. In 2011 33rd International Conference on Software Engineering (ICSE), pages 141–150. IEEE, 2011.

[71] Gerard Meszaros. xUnit test patterns: Refactoring test code. Pearson Education, 2007.

[72] Mathias Meyer. Continuous integration and its tools. IEEE software, 31(3):14–16, 2014.

[73] Ade Miller. A hundred days of continuous integration. In Agile, 2008. AGILE'08. Conference, pages 289–293. IEEE, 2008.

[74] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. Non-recursive make considered harmful: Build systems at scale. In Proceedings of the 9th International Symposium on Haskell, pages 170–181. ACM, 2016.

[75] Shinichi Nakagawa and Holger Schielzeth. A general and simple method for obtaining R2 from generalized linear mixed-effects models. Methods in Ecology and Evolution, 4(2):133–142, 2013.

[76] Ansong Ni and Ming Li. Cost-effective build outcome prediction using cascaded classifiers. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR 2017), pages 455–458, 2017.

[77] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pages 1–10, 2018.

[78] Peter Peduzzi, John Concato, Elizabeth Kemper, Theodore R Holford, and Alvan R Feinstein. A simulation study of the number of events per variable in logistic regression analysis. Journal of clinical epidemiology, 49(12):1373–1379, 1996.

[79] P Pinheiro. Linear and nonlinear mixed effects models. R package version 3.1-97. http://cran. r-project. org/web/packages/nlme, 2010.

[80] Gustavo Pinto, Fernando Castor, Rodrigo Bonifacio, and Marcel Rebouças. Work practices and challenges in continuous integration: A survey with Travis CI users. Software: Practice and Experience, 48(12):2223–2236, 2018.

[81] Akond Rahman, Amritanshu Agrawal, Rahul Krishna, and Alexander Sobran. Characterizing the influence of continuous integration: Empirical results from 250+ open source and proprietary projects. In Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics, pages 8–14. ACM, 2018.

[82] Md Tajmilur Rahman and Peter C Rigby. The impact of failing, flaky, and high failure tests on the number of crash reports associated with Firefox builds. In Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 857–862. ACM, 2018.

[83] Mohammad Masudur Rahman and Chanchal K Roy. Impact of continuous integration on code reviews. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 499–502. IEEE, 2017.

[84] Jonathan Rasmusson. Long build trouble shooting guide. Proceedings of the Extreme Programming and Agile Methods-XP/Agile Universe Conference, pages 557–574, 2004.

[85] Thomas Rausch, Waldemar Hummer, Philipp Leitner, and Stefan Schulte. 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 (MSR 2017), pages 345–355, 2017.

[86] R Owen Rogers. Scaling continuous integration. In International Conference on Extreme Programming and Agile Processes in Software Engineering, pages 68–76. Springer, 2004.

[87] J Romano, JD Kromrey, J Coraggio, and J Skowronek. Should we really be using t-test and Cohen's d for evaluating group differences on the NSSE and other surveys. In Annual meeting of the Florida association of institutional research, 2006.

[88] WS Sarle. The VARCLUS procedure. SAS/STAT User's Guide, 1990.

[89] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: A case study (at Google). In Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), pages 724–734. ACM, 2014.

[90] David J Sheskin. Handbook of parametric and nonparametric statistical procedures. crc Press, 2020.

[91] Puneet Kaur Sidhu, Gunter Mussbacher, and Shane McIntosh. Reuse (or lack thereof) in travis ci specifications: An empirical study of CI phases and commands. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 524–533. IEEE, 2019.

[92] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In ACM sigsoft software engineering notes, volume 30(4), pages 1–5. ACM, 2005.

[93] Peter Smith. Software build systems: Principles and experience. Addison-Wesley Professional, 2011.

[94] Rodrigo Souza and Bruno Silva. Sentiment analysis of Travis CI builds. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 459–462. IEEE, 2017.

[95] Daniel Ståhl and Jan Bosch. Modeling continuous integration practice differences in industry software development. Journal of Systems and Software, 87:48–59, 2014.

[96] Ewout W Steyerberg, Frank E Harrell Jr, Gerard JJM Borsboom, MJC Eijkemans, Yvonne Vergouwe, and J Dik F Habbema. Internal validation of predictive models: efficiency of some procedures for logistic regression analysis. Journal of clinical epidemiology, 54(8):774–781, 2001.

[97] Sean Stolberg. Enabling agile testing through continuous integration. In Agile Conference, 2009. AGILE'09, pages 369–374. IEEE, 2009.

[98] Travis CI. Caching dependencies and directories. https://docs.travis-ci.com/user/caching. Visited on February 13, 2021.

[99] Arie Van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. In Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001), pages 92–95, 2001.

[100] Boudewijn F Van Dongen, Ana Karla A de Medeiros, HMW Verbeek, AJMM Weijters, and Wil MP van Der Aalst. The ProM framework: A new era in

process mining tool support. In International conference on application and theory of petri nets, pages 444–454. Springer, 2005.

[101] Joachim Vandekerckhove, Dora Matzke, and Eric-Jan Wagenmakers. Model comparison and the principle. In The Oxford handbook of computational and mathematical psychology, volume 300. Oxford Library of Psychology, 2015.

[102] Bogdan Vasilescu, Stef Van Schuylenburg, Jules Wulms, Alexander Serebrenik, and Mark GJ 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 2014), pages 401–405. IEEE, 2014.

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

[104] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C Gall, and Massimiliano Di Penta. Configuration smells in continuous delivery pipelines: A linter and a six-month study on gitlab. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 327–337, 2020.

[105] David Gray Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. A conceptual replication of continuous integration pain points in the context of travis CI. In Proceedings of the 2019 27th ACM Joint Meeting on

European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pages 647–658. ACM, 2019.

[106] Bodo Winter. A very basic tutorial for performing linear mixed effects analyses. arXiv preprint arXiv:1308.5499, 2013.

[107] Jing Xia and Yanhui Li. Could we predict the result of a continuous integration build? An empirical study. In Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion (QRS 2017), pages 311–315, 2017.

[108] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 52–63. ACM, 2014.

[109] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, and Naoyasu Ubayashi. Magnet or sticky? An OSS project-by-project typology. In Proceedings of the 11th working conference on mining software repositories, pages 344–347, 2014.

[110] Yue Yu, Gang Yin, Tao Wang, Cheng Yang, and Huaimin Wang. Determinants of pull-based development in the context of continuous integration. Science China Information Sciences, 59(8):080104, 2016.

[111] H Mehmet Yuksel, Eray Tuzun, Erdogan Gelirli, Emrah Biyikli, and Buyurman Baykal. Using continuous integration and automated test techniques for a robust C4ISR system. In Computer and Information Sciences, 2009. ISCIS 2009. 24th International Symposium on, pages 743–748. IEEE, 2009.

[112] Fiorella Zampetti, Gabriele Bavota, Gerardo Canfora, and Massimiliano Di Penta. A study on the interplay between pull request review and continuous integration builds. In 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), pages 38–48. IEEE, 2019.

[113] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. How open source projects use static code analysis tools in continuous integration pipelines. In 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), pages 334–344. IEEE, 2017.

[114] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. A large-scale empirical study of compiler errors in continuous integration. In Proceedings of the 2019 13th Joint Meeting on Foundations of Software Engineering (FSE 2019), pages 176–187. ACM, 2019.

[115] Yangyang Zhao, Alexander Serebrenik, Yuming Zhou, Vladimir Filkov, and Bogdan Vasilescu. 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, pages 60–71. IEEE Press, 2017.

[116] Celal Ziftci and Jim Reardon. Who broke the build? Automatically identifying changes that induce test failures in continuous integration at google scale. In Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP 2017), pages 113–122. IEEE, 2017.

[117] Mahdis Zolfagharinia, Bram Adams, and Yann-Gaël Guéhéneuc. A study of build inflation in 30 million CPAN builds on 13 Perl versions and 10 operating systems. Empirical Software Engineering, pages 1–39, 2019.

Appendix A

Appendix A: Chapter 5: Developers' Survey

Below, we present the invitation letter we sent to software developers to participate in our survey study. We use [*Participant*] as a placeholder for the first name of every software developer we invite. The first page of the survey includes a letter of information and consent form that developers need to read and give consent to participate in the survey.

Table A.1 (Parts 1, 2, and 3) contain the questions we include in our survey We split our questions into four sections asking for demographics, development and CI practices, opinions about our empirical observations, and additional insights from developers into our study. Each section in the survey appears in a separate web page. In each survey section, we show the questions asked and the type of response expected from survey participants.

Survey recruitment letter

Dear [Participant],
I hope this email finds you well.

My name is Taher Ghaleb. I am a Ph.D. candidate at Queen's University, Canada under the supervision of Prof. Ying Zou (ying.zou@queensu.ca). I am inviting you to participate in a survey study about Continuous Integration (CI) practices, since you have been contributing to open source GitHub projects that adopt Travis CI. We have used the GitHub API to obtain the public email address associated with the commits you pushed to public GitHub repositories. We apologize in advance if our email is unwanted or wasting your time.

This study helps us better understand two major CI build problems in which builds may (1) take too long to generate, i.e., long build durations, or (2) fail frequently. We want to understand how software developers maintain a balance between mitigating long CI build durations and frequent build failures. The outcome of this study is to provide recommendations to software practitioners (i.e., developers) regarding the most appropriate actions to control the possible side effects between reducing CI build durations and frequent build failures.

There are **no mandatory questions** in our survey and it will **approximately take 15-20 minutes** of your time. There are no known risks to this study. To participate in our survey, please **click on the following link**: https://queensu.qualtrics.com/jfe/form/SV_ahpQLcMb2VOOKIS

To compensate you for your time, you may win a $10 Amazon gift card if you complete the full questionnaire. We are offering these gift cards to 30% randomly drawn participants who complete the questionnaire and share their contact information to receive the prize if they win. You will also be asked to share your contact information if you are interested in a follow-up interview study. We would also be happy to share the results of this research with you, so if you are interested, you may need to check the website of our research lab (https://seal-queensu.github.io) on which information about the research findings can be found.

If you have any questions about this survey, or difficulty in accessing the site or completing the survey, please contact Taher Ghaleb at taher.ghaleb@queensu.ca. Thank you in advance for your time and for providing important feedback to our study.

Any ethical concerns about the study may be directed to the Chair of the General Research Ethics Board at chair.greb@queensu.ca or 1(844)535-2988 (Toll-free in North America). Use 1(613)533-2988 if outside North America. Please note that GREB communicates in English only.

DISCLAIMER: We have no affiliation with any social, business, or commercial entities. This is a one-time request for your voluntary participation in an academic survey. We will not be contacting you again.

Best regards,
Taher Ghaleb
PhD Candidate
Queen's University
Canada

Table A.1: Survey Questionnaire (Part 1)

#	Question	Answer choices
Section 1: Demographics		
$Q_{1.1}$	What is your gender?	[Female, Male, Other, Prefer not to answer]
$Q_{1.2}$	Which country do you work from?	open-ended response
$Q_{1.3}$	What is your age range?	[Under 25, 25-35, 36-45, 46-55, Over 55]
$Q_{1.4}$	What are your overall years of experience with software development?	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10+] years
$Q_{1.5}$	What are your overall years of experience with TravisCI?	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10+] years
$Q_{1.6}$	What is your role(s) in the repository or the technology associated with TravisCI?	[user, maintainer, active contributor, occasional contributor, other]
$Q_{1.7}$	What other CI services have you used in addition to TravisCI? (Select all that apply)	[CircleCI, Appveyor, TeamCity, other]
Section 2: Development and CI practices In this section, we wish to know more about the development and continuous integration practices you normally perform to deal with long build durations (i.e., the time it takes TravisCI to generate build) and build failures (including *errored* and *failed* build status).		
$Q_{2.1}$	To what extent do you think that project characteristics (e.g., project age, number of branches, number of developers, number of tests, etc.) are associated with long build durations and frequent build failures?	Likert Scale [1, 2, 3, 4, 5] (1: strongly disagree, 5: strongly agree)
$Q_{2.2}$	Based on your experience, what project characteristics have an association with the build generation the most?	open-ended response
$Q_{2.3}$	Based on the projects you contribute to and are linked with TravisCI, to what extent do you think that the build duration and frequency of build failures change over time?	Likert Scale [1, 2, 3, 4, 5] (1: very unlikely, 5: very likely)
$Q_{2.4}$	How often do you act towards reducing build durations?	Likert Scale [1, 2, 3, 4, 5] (1: rarely, 5: very frequently)
$Q_{2.5}$	What actions do you usually take to reduce build durations?	open-ended response
$Q_{2.6}$	Have you ever configured CI builds to reduce build durations despite its negative impact on build failures?	[Yes, No, Unsure]
$Q_{2.7}$	How often do you act towards fixing build failures?	Likert Scale [1, 2, 3, 4, 5] (1: rarely, 5: very frequently)
$Q_{2.8}$	What actions do you usually take to reduce build failures?	open-ended response

Table A.1: Survey Questionnaire (Part 2)

$Q_{2.9}$	Have you ever configured CI builds to avoid build failures despite its negative impact on build durations?	[Yes, No, Unsure]
$Q_{2.10}$	Would you give more priority to resolving long build durations or frequent build failures?	[Durations, Failures, None, Both, Other issues]
$Q_{2.11}$	Are there build failures that you consider unnecessary or unwanted? Please explain?	[Yes, No, Unsure]
$Q_{2.12}$	How would you define an unnecessary or unwanted build failure? Can you give us an example of build failures that you usually consider as unnecessary?	open-ended response
$Q_{2.13}$	Based on your experience with TravisCI builds, what are the practices or build configurations that would **reduce build failures** but may likely lead to a slower build generation?	open-ended response
$Q_{2.14}$	Based on your experience with TravisCI builds, what are the practices or build configurations that would reduce the build duration but may likely lead to more frequent build failures?	open-ended response
$Q_{2.15}$	Based on your CI experience, what are the practices or build configurations that would affect (either positively or negatively) both the build duration and status?	open-ended response

Section 3: Opinion about our empirical observations
In this section, we wish to get your generous feedback on the observations that we obtain from conducting statistical analyses using an empirical study on 500+ GitHub projects that are linked with Travis CI.

Below, you will find observations from our empirical analysis. Please let us know the extent to which you Agree or Disagree with each finding and how you would justify your answer or explain the possible reasons for each finding, based on your experience.

#	Observation	Likert Scale [1: strongly disagree, 2, 3, 4, 5: strongly agree]	Justification/Explanation open-ended response
$Q_{3.1}$	Younger projects (i.e., those that are at their early development stages) are likely to have more build failures but faster builds.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{3.2}$	The more branches a project has, the better control the project has over build duration and build failures.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{3.3}$	Modifying the build configuration more frequently can be associated with build durations and failures.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{3.4}$	Allowing builds to wait for long-running tests or to retry failing commands multiple times is unlikely to help generate passed builds.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{3.5}$	Pull requests and multi-commit pushes are likely to generate builds that fail fast.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	
$Q_{3.6}$	Besides making builds faster, CI caching can also be associated with a reduced number of build failures.	1 ○ 2 ○ 3 ○ 4 ○ 5 ○	

Table A.1: Survey Questionnaire (Part 3)

Q$_{3.7}$	CI builds triggered on weekends are more likely to be faster and passed.	1 2 3 4 5 ○ ○ ○ ○ ○	
Q$_{3.8}$	Actions to reduce build durations or failures are restricted by the current state of the project builds (i.e., whether a project suffers from either slower or failed builds, or both)? For example, using fewer command retrials may improve builds that take a longer time to fail, but builds that fail fast are unlikely to benefit from such a practice.	1 2 3 4 5 ○ ○ ○ ○ ○	
Q$_{3.9}$	Running builds on certain Linux distributions (e.g., trusty) or operating systems (e.g., OSx) have an association with long build durations and frequent build failures.	1 2 3 4 5 ○ ○ ○ ○ ○	
Q$_{3.10}$	Commits submitted by occasional and less experienced developers are associated with making builds faster and passing.	1 2 3 4 5 ○ ○ ○ ○ ○	
Q$_{3.11}$	The *'fast_finish'* configuration allows builds to finish without waiting for the *'allow_failures'* jobs. However, *'fast_finish'* may only help when builds fail frequently (i.e., builds that usually pass do not benefit from *'fast_finish'*).	1 2 3 4 5 ○ ○ ○ ○ ○	
Q$_{3.12}$	Build durations and build failures tend to follow a similar pattern from one build to another (i.e., if a previous build is slow or failed, the following build will most likely be slow/failed).	1 2 3 4 5 ○ ○ ○ ○ ○	
Section 4: Additional insights In this section, we wish to get your generous insights and recommendations on how developers should deal with different alternatives to improve the build generation.			
Q$_{4.1}$	What should developers take into consideration when acting towards reducing build durations and/or failures?		open-ended response
Q$_{4.2}$	What tools or approaches do you think developers still need to improve the build generation? What can researchers do in this regard?		open-ended response
Q$_{4.3}$	What facilities should TravisCI (or any other CI service) provide to improve the overall experience of CI building?		open-ended response
Q$_{4.4}$	Are you willing to have a quick follow-up interview to help our research on reducing build durations and failures? If yes, please leave your Email in the pop-up form.		[Separate Form] open-ended response
Q$_{4.5}$	Would you be interested in receiving an Amazon gift card? If yes, please leave your Email in the pop-up form.		[Separate Form] open-ended response
Q$_{4.6}$	Finally, if you have any comments on this survey, please let us know.		open-ended response

Appendix B

Appendix B: Chapter 6: Pull Requests, Commits, and Issues

Table B.1: List of responses obtained from developers regarding the non-adoption of CI caching

Category	**Link**	**Type**	**Response**
Caching Non-adopters	woorea/openstack-java-sdk/pull/226	*Approved*	No reason given
	concerto/concerto/pull/1556	*Merged*	No reason given
	awestruct/awestruct/pull/557	*Merged*	
	Albacore/albacore/pull/253	*Merged*	
	PebbleTemplates/pebble/pull/583	*Merged*	
	Azure/azure-sdk-for-ruby/pull/2825	*Merged*	
	ctran/annotate_models/pull/853	*Merged*	
	neuland/jade4j/pull/196	*Merged*	
	michelson/lazy_high_charts/pull/260	*Merged*	Appreciation
	ledermann/unread/pull/124	*Merged*	
	modelmapper/modelmapper/pull/585	*Merged*	
	remi/teamocil/pull/140	*Merged*	
	redpen-cc/redpen/pull/891	*Merged*	
	w3c/epubcheck/pull/1215	*Merged*	
	pedrovgs/Renderers/pull/57	*Merged*	
	ruby-ldap/ruby-net-ldap/pull/390	*Merged*	
	SlatherOrg/slather/pull/479	*Merged*	
	nathanl/searchlight/pull/35	*Merged*	Unaware of the caching support
	frab/frab/pull/798	*Merged*	
	twilio/twilio-java/pull/624	*Merged*	
	joelittlejohn/embedmongo-maven-plugin/pull/88	*Merged*	
	grosser/fast_gettext/pull/130	*Merged*	
	magnusvk/counter_culture/pull/310	*Merged*	
	drewnoakes/metadata-extractor/pull/530	*Merged*	
	davidmoten/rxjava-jdbc/pull/96	*Merged*	Merged as the build passes
	gma/nesta/pull/185	*Merged*	
	spree/spree_gateway/pull/374	*Merged*	
	zquestz/omniauth-google-oauth2/pull/402	*Merged*	
	faye/faye-websocket-ruby/pull/130	*Merged*	Assumption that caching was enabled by default
	spullara/mustache.java/pull/260	*Merged*	
	tom-lord/regexp-examples/pull/38	*Merged*	Developers are busy or have other priorities
	FasterXML/jackson-core/pull/682	*Open*	Not useful and/or introduced problems
	JavaMoney/jsr354-api/pull/133	*Open*	
	airlift/slice/pull/139	*Open*	
	Esri/geometry-api-java/pull/283	*Open*	
	tpitale/staccato/pull/96	*Open*	
	zombocom/wicked/pull/276	*Open*	Moving to other CI services
	fluent/fluent-plugin-s3/pull/370	*Closed*	Moving to other CI services
	tigrish/devise-i18n/pull/291	*Closed*	
	languagetool-org/languagetool/pull/4575	*Closed*	
	ruby2js/ruby2js/pull/104	*Closed*	
	SpongePowered/Sponge/pull/3323	*Closed*	
	FasterXML/jackson-annotations/pull/185	*Closed*	
	bkeepers/dotenv/pull/427	*Closed*	
	NoBrainerORM/nobrainer/pull/276	*Closed*	

Table B.2: List of responses obtained from developers regarding the proactive and late adoption of CI caching

Category	Link	Adoption	Response
Caching Proactive-adopters	opal/opal/commit/a029afa	Commit	Caching was available unofficially (pre-release)
	middleman/middleman/commit/c1ef5fc	Commit	
	merit-gem/merit/commit/810de5d	Commit	
	ruby-rdf/json-ld/commit/4b5413d	Commit	
	refinery/refinerycms-blog/commit/382a379	Commit	
	vcr/vcr/commit/c5e90d0	Commit	
	errbit/errbit/commit/dcc5223	Commit	
	travis-ci/travis.rb/commit/e68823d	Commit	
	gavinlaking/vedeu/commit/af56779	Commit	
	zxing/zxing/commit/699471b	Commit	Developer's misunderstanding
	OryxProject/oryx/commit/2ba60d3	Commit	
	ms-ati/docile/commit/3aef43a	Commit	
	ms-ati/docile/commit/6450ece	Commit	
	fphilipe/premailer-rails/commit/ec08c65	Commit	
	praxis/praxis/commit/4df5c7a	Commit	Developers cannot recall
	lenskit/lenskit/commit/13cb1dd	Commit	
	thoughtbot/shoulda-matchers/commit/b845451	Commit	
	skylightio/skylight-ruby/commit/c115aee	Commit	Copied from another project
	dblock/rspec-rerun/commit/5a417df	Commit	
	honeybadger-io/honeybadger-ruby/commit/5d93118	Commit	
	tdiary/tdiary-core/commit/4121404	Commit	Travis CI is no longer used
	spring-cloud/spring-cloud-config/commit/574c4d5	Commit	
Caching Late-adopters	premailer/premailer/pull/294	Pull request	Unaware of the caching support
	bolshakov/stoplight/pull/90	Pull request	
	google/guice/pull/975	Pull request	
	grosser/parallel/pull/159	Pull request	
	guard/guard/pull/836	Pull request	
	mgomes/api_auth/pull/108	Pull request	
	inaturalist/inaturalist/pull/805	Pull request	
	celluloid/celluloid/pull/711	Pull request	
	gregorym/bump/pull/50	Pull request	
	MaximeD/gem_updater/pull/64	Pull request	
	openfoodfoundation/openfoodnetwork/commit/a95727b	Pull request	
	swipely/docker-api/pull/324	Pull request	
	eapache/starscope/pull/145	Pull request	
	attr-encrypted/attr_encrypted/pull/182	Pull request	
	scambra/devise_invitable/commit/c74eaba	Commit	
	Parallels/vagrant-parallels/commit/7f2fdae	Commit	
	countries/countries/commit/9d37cd3	Commit	
	zdavatz/spreadsheet/commit/a9ac996	Commit	
	zeisler/active_mocker/commit/e6cd3de	Commit	
	spacecowboy/NotePad/commit/258b077	Commit	
	JetBrains/idea-gitignore/commit/7e919da	Commit	
	apache/storm/pull/902	Pull request	Developers are busy or have other priorities
	matschaffer/knife-solo/pull/463	Pull request	
	square/keywhiz/pull/176	Pull request	
	celluloid/celluloid-zmq/pull/60	Pull request	
	voxpupuli/puppet-mongodb/pull/259	Pull request	
	voxpupuli/puppet-rabbitmq/pull/427	Pull request	
	puppetlabs/puppetlabs-firewall/pull/599	Pull request	
	newrelic/newrelic-ruby-agent/commit/db1b1fd	Commit	
	pboling/sanitize_email/commit/cdb7073	Commit	
	typhoeus/ethon/commit/29bb100	Commit	
	weppos/whois/commit/17c467c	Commit	
	shadabahmed/logstasher/commit/af12db4	Commit	
	mikel/mail/pull/999	Pull request	Developers cannot recall / no specific reason
	toy/image_optim/issues/113	Pull request	
	airlift/airlift/pull/359	Pull request	
	jmettraux/rufus-scheduler/commit/9a9514c	Commit	
	adelevie/parse-ruby-client/commit/898606a	Commit	
	SoftInstigate/restheart/commit/51467d8	Commit	
	facebook/buck/commit/bc01418	Commit	

Table B.3: Common issues of CI caching along with the proposed fixes/workarounds

Caching issue	**Fix/workaround**
Artifacts not being cached	Check for unsupported platforms
	Avoid using wildcards/spaces in the cache path
	Customized `install` might change the default directory
Changes detected in a removed directory	Cache might sync directories before removal. List all directories (except that one)
Seeking a special caching directive	List all directories manually until a special directive is supported
Caching popular dependencies/libraries	Suggest them to the TRAVIS CI team to be added them to the base images
Suspecting dependency issues to be cache-related	Install dependencies locally or using a different runtime environment
Corrupted/outdated cache (e.g., wrong dependency versions)	Clear the cache manually from the CI server